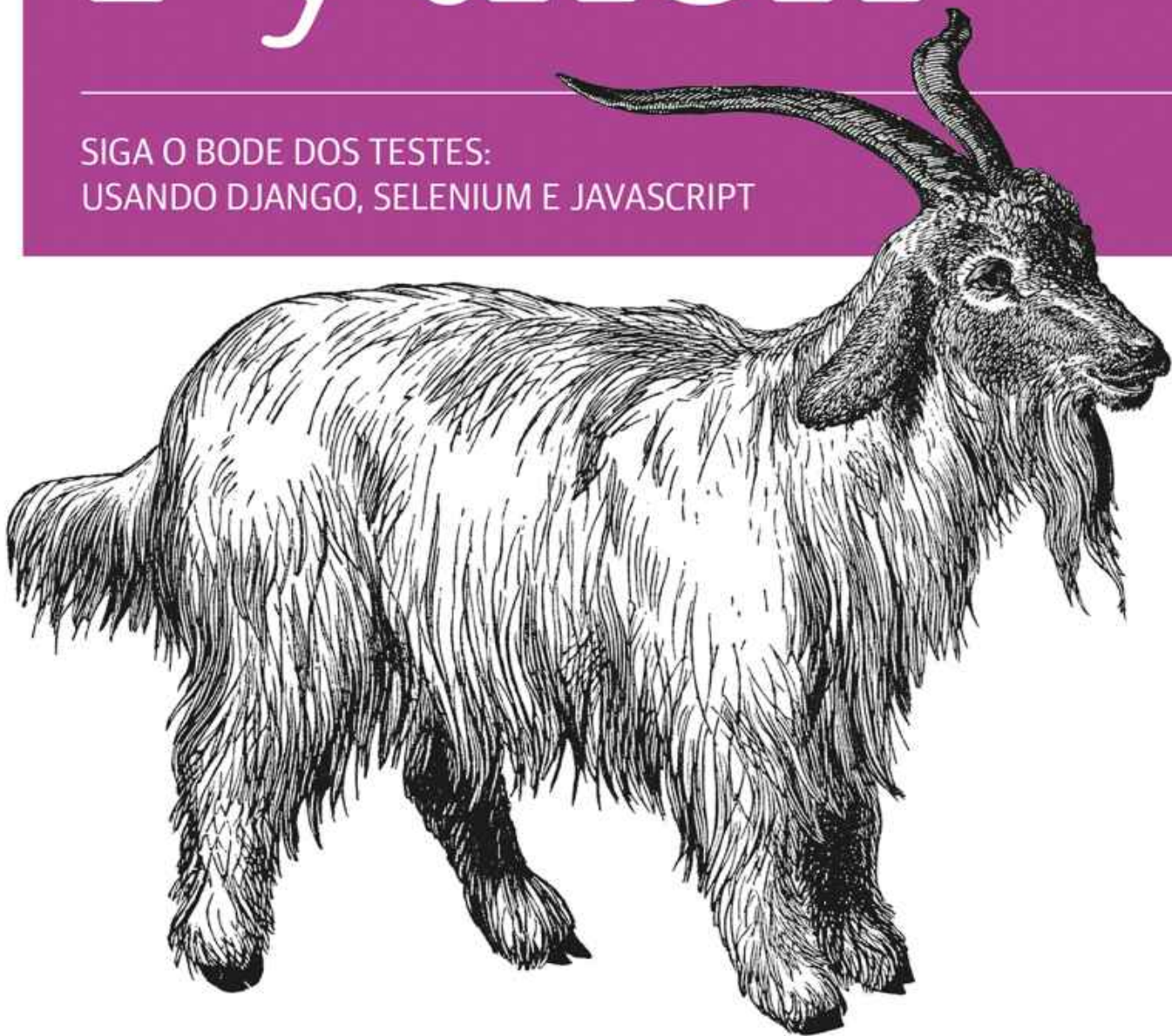


O'REILLY®

TDD com Python

SIGA O BODE DOS TESTES:
USANDO DJANGO, SELENIUM E JAVASCRIPT



novatec

Harry J.W. Percival

Elogios para TDD com Python

“Neste livro, Harry nos leva a uma aventura de descoberta com Python e testes. É um livro excelente, divertido de ler e cheio de informações vitais. Tem minhas melhores recomendações para qualquer pessoa interessada em testes com Python, que queira conhecer o Django ou usar o Selenium. Testar é essencial para a sanidade dos desenvolvedores, e é um campo notoriamente difícil, repleto de negociações. Harry faz um trabalho fantástico para prender a nossa atenção enquanto explora as práticas de testes do mundo real.”

– MICHAEL FOORD,
DESENVOLVEDOR DO NÚCLEO DE PYTHON & MANTENEDOR DO UNITTEST

“Este livro é muito mais que uma introdução ao desenvolvimento orientado a testes: é um curso rápido completo, do início ao fim, sobre as melhores práticas para o desenvolvimento moderno de aplicações web com Python. Todo desenvolvedor web precisa deste livro.”

– KENNETH REITZ,
MEMBRO DA PYTHON SOFTWARE FOUNDATION

“O livro de Harry é o que queríamos que existisse quando estávamos conhecendo o Django. Em um ritmo acessível, embora agradavelmente desafiador, a obra contém explicações excelentes para o Django e várias práticas de testes. Só o conteúdo sobre o Selenium faz com que valha a pena comprar o livro, mas há muito mais nele!”

– DANIEL E AUDREY ROY GREENFELD,
AUTORES DE *TWO SCOOPS OF DJANGO* (TWO SCOOPS PRESS)

TDD com Python

Harry J.W. Percival

O'REILLY
Novatec
São Paulo | 2019

Authorized Portuguese translation of the English edition of Test-Driven Development with Python, 2nd Edition, ISBN 9781491958704 © 2017 Harry J. W. Percival. This translation is published and sold by permission of O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

Tradução em português autorizada da edição em inglês da obra Test-Driven Development with Python, 2nd Edition, ISBN 9781491958704 © 2017 Harry J. W. Percival. Esta tradução é publicada e vendida com a permissão da O'Reilly Media, Inc., detentora de todos os direitos para publicação e venda desta obra.

© Novatec Editora Ltda. [2017].

Todos os direitos reservados e protegidos pela Lei 9.610 de 19/02/1998. É proibida a reprodução desta obra, mesmo parcial, por qualquer processo, sem prévia autorização, por escrito, do autor e da Editora.

Editor: Rubens Prates

Tradução: Lúcia A. Kinoshita

Revisão gramatical: Tássia Carvalho

Editoração eletrônica: Carolina Kuwabata

ISBN: 978-85-7522-768-8

Histórico de edições impressas:

Novembro/2017 Primeira edição

Novatec Editora Ltda.

Rua Luís Antônio dos Santos 110

02460-000 – São Paulo, SP – Brasil

Tel.: +55 11 2959-6529

Email: novatec@novatec.com.br

Site: www.novatec.com.br

Twitter: twitter.com/novateceditora

Facebook: facebook.com/novatec

LinkedIn: linkedin.com/in/novatec

Sumário

[Elogios para TDD com Python](#)

[Prefácio](#)

[Pré-requisitos e suposições](#)

[Vídeo complementar](#)

[Agradecimentos](#)

[Parte I ■ Básico sobre TDD e Django](#)

[Capítulo 1 ■ Configurando o Django com um teste funcional](#)

[Obedeça ao Testing Goat! Não faça nada até ter um teste](#)

[Deixando o Django pronto para funcionar](#)

[Iniciando um repositório no Git](#)

[Capítulo 2 ■ Estendendo nosso teste funcional usando unittest](#)

[Usando teste funcional para definir o escopo de uma aplicação mínima viável](#)

[Módulo unittest da biblioteca-padrão de Python](#)

[Commit](#)

[Capítulo 3 ■ Testando uma página inicial simples com testes de unidade](#)

[Nossa primeira aplicação Django e nosso primeiro teste de unidade](#)

[Testes de unidade e como eles diferem dos testes funcionais](#)

[Testes de unidade no Django](#)

[MVC de Django, URLs e funções de view](#)

[Finalmente! Vamos realmente escrever um pouco de código de aplicação!](#)

[urls.py](#)

[Fazendo testes de unidade em uma view](#)

[O ciclo de testes de unidade/código](#)

Capítulo 4 ■ O que estamos fazendo com todos esses testes?

(E a refatoração)

[Programar é como puxar um balde de água de um poço](#)

[Usando o Selenium para testar interações com o usuário](#)

[A regra “Não teste constantes”, e os templates que vêm para nos salvar](#)

[Refatorando de modo a usar um template](#)

[Django Test Client](#)

[Sobre a refatoração](#)

[Um pouco mais sobre a nossa página inicial](#)

[Revisão: o processo de TDD](#)

Capítulo 5 ■ Salvando a entrada do usuário: testando o banco de dados

[Preparando o nosso formulário para enviar uma requisição POST](#)

[Processando uma requisição POST no servidor](#)

[Passando variáveis Python para serem renderizadas no template](#)

[Três acertos e refatorar](#)

[ORM do Django e o nosso primeiro modelo](#)

[Nossa primeira migração de banco de dados](#)

[Os testes vão surpreendentemente longe](#)

[Um novo campo implica uma nova migração](#)

[Salvando o POST no banco de dados](#)

[Redirecionar após um POST](#)

[Melhor prática para testes de unidade: cada teste deve testar somente um aspecto](#)

[Renderizando itens no template](#)

[Criando nosso banco de dados de produção com migrate](#)

Revisão

Capítulo 6 ■ Melhorando os testes funcionais: garantindo o isolamento e removendo sleeps vodus

Garantindo o isolamento dos testes em testes funcionais

Executando somente os testes de unidade

Informação extra: atualizando o Selenium e o Geckodriver

Sobre esperas implícitas e explícitas, e time.sleeps vodus

Capítulo 7 ■ Trabalhando de forma incremental

Design pequeno quando necessário

Sem um grande design logo no início

YAGNI!

(No estilo) REST

Implementando o novo design de forma incremental usando TDD

Garantindo que teremos um teste de regressão

Iterando em direção ao novo design

Dando um primeiro passo autocontido: um novo URL

Um novo URL

Uma nova função de view

Verde? Refatorar

Outro pequeno passo: um template separado para visualizar listas

Um terceiro passo pequeno: um URL para adicionar itens de lista

Uma classe de testes para a criação de uma nova lista

Um URL e uma view para a criação de uma nova lista

Removendo código e testes não redundantes

Uma regressão! Apontando nossos formulários para o novo URL

Encarando a situação de frente: ajustando nossos modelos

Um relacionamento de chave estrangeira

Adaptando o restante do mundo aos novos modelos

Cada lista deve ter o próprio URL

Capturando parâmetros de URLs

Adaptando new_list para o novo mundo

Os testes funcionais detectam outra regressão

Mais uma view para tratar a adição de itens em uma lista existente

[Tome cuidado com expressões regulares gulosas!](#)
[Último URL novo](#)
[Última nova view](#)
[Testando os objetos de contexto de resposta diretamente](#)
[Uma última refatoração usando inclusões de URL](#)

Parte II ■ Sine qua nons do desenvolvimento web

Capítulo 8 ■ Embelezamento: layout e estilização, e o que testar sobre eles

[O que testar funcionalmente quanto ao layout e ao estilo](#)
[Embelezamento: usando um framework CSS](#)
[Herança de templates do Django](#)
[Integrando o Bootstrap](#)
[Linhas e colunas](#)
[Arquivos estáticos no Django](#)
[Mudando para StaticLiveServerTestCase](#)
[Usando componentes do Bootstrap para melhorar a aparência do site](#)
[Jumbotron!](#)
[Entradas maiores](#)
[Estilização de tabelas](#)
[Usando o nosso próprio CSS](#)
[O que não havíamos revelado: collectstatic e outros diretórios estáticos](#)
[Alguns pontos que ficaram de fora](#)

Capítulo 9 ■ Testando a implantação usando um site de staging

[TDD e as áreas perigosas da implantação](#)
[Como sempre, comece com um teste](#)
[Obtendo um nome de domínio](#)
[Provisionamento manual de um servidor para hospedar o nosso site](#)
[Escolhendo o lugar para hospedar o nosso site](#)
[Iniciando um servidor](#)
[Contas de usuário, SSH e privilégios](#)

[Instalando o Nginx](#)

[Instalando o Python 3.6](#)

[Configurando domínios para o ambiente de staging e o ambiente live](#)

[Usando o FT para confirmar que o domínio funciona e que o Nginx está executando](#)

[Implantando o nosso código manualmente](#)

[Acertando a localização do banco de dados](#)

[Criando um virtualenv manualmente e usando requirements.txt](#)

[Configuração simples do Nginx](#)

[Criando o banco de dados com migrate](#)

[Sucesso! Nossa implantação hack funciona](#)

Capítulo 10 ■ Chegando a uma implantação pronta para produção

[Passando a usar o Gunicorn](#)

[Fazendo o Nginx servir arquivos estáticos](#)

[Passando a usar sockets Unix](#)

[Alterando DEBUG para false e configurando ALLOWED_HOSTS](#)

[Usando Systemd para garantir que o Gunicorn inicie no boot](#)

[Salvando nossas alterações: adicionando o Gunicorn ao nosso requirements.txt](#)

[Pensando em automatizar](#)

[Salvando templates para os arquivos de configuração de nosso provisionamento](#)

[Salvando o nosso progresso](#)

Capítulo 11 ■ Automatizando a implantação com o Fabric

[Detalhando um script do Fabric para a nossa implantação](#)

[Criando a estrutura de diretórios](#)

[Obtendo nosso código-fonte com o Git](#)

[Atualizando settings.py](#)

[Atualizando o virtualenv](#)

[Migrando o banco de dados se for necessário](#)

[Testando](#)

[Implantação no ambiente live](#)
[Configuração de Nginx e de Gunicorn usando sed](#)
[Atribua uma tag do Git à versão](#)
[Leituras complementares](#)

Capítulo 12 ■ Separando testes em vários arquivos e criando método auxiliar genérico para espera

[Comece com um FT de validação: evitando itens em branco](#)
[Ignorando um teste](#)
[Separando os testes funcionais em vários arquivos](#)
[Executando um único arquivo de teste](#)
[Uma nova ferramenta para testes funcionais: um método auxiliar genérico para espera explícita](#)
[Finalizando o FT](#)
[Refatorando os testes de unidade em vários arquivos](#)

Capítulo 13 ■ Validação na camada do banco de dados

[Validação na camada do modelo](#)
[Gerenciador de contexto self.assertRaises](#)
[Idiossincrasia do Django: salvar o modelo não faz a validação ser executada](#)
[Mostrando erros de validação do modelo na view](#)
[Verificando se uma entrada inválida não é salva no banco de dados](#)
[Padrão Django: processando requisições POST na mesma view em que o formulário é renderizado](#)
[Refatorar: transferindo a funcionalidade new_item para view_list](#)
[Impondo a validação do modelo em view_list](#)
[Refatorar: removendo os URLs fixos no código](#)
[Tag de template {% url %}](#)
[Usando get_absolute_url para redirecionamentos](#)

Capítulo 14 ■ Um formulário simples

[Passando a lógica de validação para um formulário](#)

[Explorando a API dos formulários com um teste de unidade](#)
[Passando para um ModelForm do Django](#)
[Testando e personalizando a validação de formulário](#)
[Usando o formulário em nossas views](#)
[Usando o formulário em uma view com uma requisição GET](#)
[Uma grande operação de localizar e substituir](#)
[Usando o formulário em uma view que aceita requisições POST](#)
[Adaptando os testes de unidade para a view new_list](#)
[Usando o formulário na view](#)
[Usando o formulário para exibir erros no template](#)
[Usando o formulário na outra view](#)
[Um método auxiliar para vários testes pequenos](#)
[Um benefício inesperado: validação gratuita no lado cliente pelo HTML5](#)
[Um tapinha nas costas](#)
[No entanto, desperdiçamos muito tempo?](#)
[Usando o método do próprio formulário para salvar dados](#)

Capítulo 15 ■ Formulários mais sofisticados

[Outro FT para itens duplicados](#)
[Evitando duplicações na camada do modelo](#)
[Uma pequena digressão sobre ordenação de querysets e representações de string](#)
[Reescrevendo o teste antigo do modelo](#)
[Alguns erros de integridade aparecem quando salvamos os dados](#)
[Fazendo experimentos com validação de itens duplicados na camada de views](#)
[Um formulário mais complexo para tratar a validação de unicidade](#)
[Usando o formulário de item de lista existente na view de lista](#)
[Conclusão: o que aprendemos sobre testar o Django](#)

Capítulo 16 ■ Mergulhando os pés, cautelosamente, no JavaScript

[Começando com um FT](#)
[Configurando um executor de testes básico de JavaScript](#)
[Usando a jQuery e a div de fixture](#)

[Construindo um teste de unidade de JavaScript para a funcionalidade desejada](#)

[Fixtures, ordem de execução e o estado global: principais desafios nos testes de JavaScript](#)

[console.log para prints de depuração](#)

[Usando uma função de inicialização para ter mais controle sobre o tempo de execução](#)

[Boilerplate para onload e namespacing](#)

[Testes de JavaScript no ciclo de TDD](#)

[Alguns tópicos que não puderam ser abordados](#)

Capítulo 17 ■ Implantando o nosso novo código

[Implantação no servidor de staging](#)

[Implantação no servidor live](#)

[O que fazer se você vir um erro de banco de dados](#)

[Conclusão: atribua uma tag no git para a nova versão](#)

Parte III ■ Tópicos mais avançados sobre testes

Capítulo 18 ■ Autenticação de usuário, spiking e de-spiking

[Autenticação sem senha](#)

[Programação exploratória, também conhecida como “spiking”](#)

[Iniciando um branch para o spike](#)

[UI de login no frontend](#)

[Enviando emails a partir do Django](#)

[Usando variáveis de ambiente para evitar segredos no código-fonte](#)

[Armazenando tokens no banco de dados](#)

[Modelos personalizados para autenticação](#)

[Finalizando a autenticação personalizada do Django](#)

[De-spiking](#)

[Revertendo o nosso código de spiking](#)

[Um modelo de usuário personalizado mínimo](#)

[Testes como documentação](#)

[Um modelo de token para associar emails a um ID único](#)

Capítulo 19 ■ Usando mocks para testar dependências externas ou reduzir a duplicação

Antes de começar: definindo a infraestrutura básica

Fazendo uma simulação manual, também conhecida como monkeypatching

Biblioteca Mock de Python

Usando unittest.patch

Avançando um pouco mais no FT

Testando o framework de mensagens do Django

Adicionando mensagens em nosso HTML

Começando pelo URL de login

Verificando se enviamos um link com um token para o usuário

De-spiking de nosso backend personalizado de autenticação

Um if = mais um teste

Método get_user

Usando o nosso backend de autenticação na view de login

Uma razão alternativa para usar mocks: reduzir a duplicação

Usando mock.return_value

Patching no nível de classe

Hora da verdade: o FT passará?

Teoricamente o FT funciona! Ele funciona na prática?

Terminando o nosso FT, testando o logout

Capítulo 20 ■ Fixtures de teste e um decorador para esperas explícitas

Pulando o processo de login e criando previamente uma sessão

Verificando se o código funciona

Nossa função auxiliar final de espera explícita: um decorador wait

Capítulo 21 ■ Depuração no lado do servidor

Prova definitiva: usando o servidor de staging para capturar os últimos bugs

Configurando o logging

Definindo variáveis de ambiente secretas no servidor

Adaptando o nosso FT para que possamos testar emails de verdade

via POP3

Administrando o banco de dados de testes no ambiente de staging

Um comando de gerenciamento do Django para criar sessões

Fazendo o FT executar o comando de gerenciamento no servidor

Usando o Fabric diretamente de Python

Revisão: criando sessões localmente *versus* no ambiente de staging

Incluindo o nosso código de logging

Conclusão

Capítulo 22 ■ Finalizando “My Lists”: TDD Outside-In

A alternativa: “Inside-Out”

Por que dar preferência para a abordagem “Outside-In”?

FT para “My Lists”

Camada mais externa: apresentação e templates

Descendo uma camada, em direção às funções de view (o controlador)

Outro teste que passa usando a abordagem Outside-In

Uma reestruturação rápida da hierarquia de herança dos templates

Fazendo o design de nossa API usando o template

Descendo para a próxima camada: o que a view passa para o template

Próximo “requisito” da camada de views: novas listas devem registrar o proprietário

Ponto de decisão: devemos prosseguir para a próxima camada com um teste em falha?

Descendo para a camada de modelo

Último passo: passando dados pela API .name do template

Capítulo 23 ■ Isolamento de testes e “Ouvindo os seus testes”

Voltando ao nosso ponto de decisão: a camada de views depende de um código não escrito para modelos

Primeira tentativa de usar mocks para isolamento

Usando side_effects de mocks para verificar a sequência de

eventos

Ouçã os seus testes: testes feios sinalizam a necessidade de refatorar

Reescrevendo nossos testes para a view de modo totalmente isolado

Mantenha a antiga suíte de testes integrados presente, como verificação de sanidade

Uma nova suíte de testes com total isolamento

Pensando em termos de colaboradores

Descendo para a camada de formulários

Continue ouvindo os seus testes: removendo o código de ORM de nossa aplicação

Finalmente descendo para a camada de modelos

De volta às views

A hora da verdade (e os riscos da simulação)

Pensando nas interações entre as camadas como sendo “contratos”

Identificando contratos implícitos

Corrigindo o que deixamos passar

Mais um teste

Organizando o código: o que deve ser mantido em nossa suíte de testes integrados

Removendo código redundante da camada de formulários

Removendo a antiga implementação da view

Removendo código redundante da camada de formulários

Conclusões: quando escrever testes isolados versus testes integrados

Deixe que a complexidade seja o seu guia

Você deve ter ambos?

Em frente!

Capítulo 24 ■ Integração Contínua (CI)

Instalando o Jenkins

Configurando o Jenkins

Desbloqueio inicial

Plugins sugeridos, por enquanto

Configurando o usuário administrador

[Adicionando plugins](#)

[Informando ao Jenkins o local para encontrar o Python 3 e o Xvfb](#)

[Concluindo com HTTPS](#)

[Configurando o nosso projeto](#)

[Primeira construção!](#)

[Configurando um display virtual para que FTs possam executar em modo headless](#)

[Capturando imagens de tela](#)

[Na dúvida, tente aumentar o timeout!](#)

[Executando nossos testes de JavaScript com QUnit no Jenkins com o PhantomJS](#)

[Instalando o node](#)

[Acrescentando os passos de construção no Jenkins](#)

[Outras tarefas que um servidor de CI pode fazer](#)

Capítulo 25 ■ Aspecto social, padrão Page e exercício para o leitor

[Um FT com vários usuários, e a função addCleanup](#)

[Padrão Page](#)

[Estendendo o FT para um segundo usuário, e a página “My Lists”](#)

[Um exercício para o leitor](#)

Capítulo 26 ■ Testes rápidos, testes lentos e Lava Quente

[Tese: testes de unidade são super-rápidos e bons, além de tudo](#)

[Testes mais rápidos significam desenvolvimento mais rápido](#)

[O sagrado estado de fluxo](#)

[Testes lentos não são executados com tanta frequência, o que resulta em código ruim](#)

[Não há problemas agora, mas os testes integrados se tornam mais lentos com o passar do tempo](#)

[Não sou eu quem está falando](#)

[E os testes de unidade levam a um bom design](#)

[Os problemas com os testes de unidade “puros”](#)

[Testes isolados podem ser mais difíceis de ler e de escrever](#)

[Testes isolados não testam automaticamente a integração](#)

Testes de unidade raramente capturam bugs inesperados
Testes com simulação podem se tornar extremamente vinculados
à implementação

Entretanto todos os problemas podem ser superados

Síntese: afinal de contas, o que queremos de nossos testes?

Aplicação correta

Código limpo, possível de manter

Fluxo de trabalho produtivo

Avalie seus testes em relação aos benefícios que você quer que
eles proporcionem

Soluções arquitetônicas

Portas e adaptadores/arquitetura hexagonal/limpa

Functional Core, Imperative Shell

Conclusão

Leituras complementares

Epílogo ■ Obedeça ao Testing Goat!

Testar é difícil

Mantenha suas construções no CI com sinal verde

Tenha orgulho de seus testes, assim como você tem de seu código

Lembre-se de dar gorjetas para o pessoal do bar

Não seja um desconhecido!

Apêndice A ■ PythonAnywhere

Apêndice B ■ Views baseadas em classe do Django

Apêndice C ■ Provisionamento com o Ansible

Apêndice D ■ Testando migrações de banco de dados

Apêndice E ■ Desenvolvimento orientado a comportamento (BDD)

Apêndice F ■ Construindo uma API REST: JSON,

[Ajax e simulação com JavaScript](#)

[Apêndice G ■ Django-Rest-Framework](#)

[Apêndice H ■ Folha de cola](#)

[Apêndice I ■ O que fazer em seguida](#)

[Apêndice J ■ Código-fonte dos exemplos](#)

[Bibliografia](#)

[Sobre o autor](#)

[Colofão](#)

Prefácio

Este livro é minha tentativa de compartilhar com o mundo a jornada que empreendi de “hacking” para “engenharia de software”. É, acima de tudo, sobre testes, mas há muito mais nele, como você verá em breve.

Gostaria de agradecer a você por lê-lo.

Se comprou uma cópia, sou muito grato. Se estiver lendo a versão online gratuita (em inglês), *continuo* sendo grato por ter decidido que vale a pena gastar parte de seu tempo com ele. Quem sabe, depois que chegar ao final, você decida que o livro é bom o suficiente a ponto de você comprar uma cópia de verdade para si ou para um amigo.

Se tiver comentários, perguntas ou sugestões, eu adoraria ouvi-los. Entre em contato comigo diretamente usando obeythetestinggoat@gmail.com, ou pelo Twitter em [@hjwp](https://www.twitter.com/hjwp) (<https://www.twitter.com/hjwp>). Também é possível consultar o site e meu blog (<http://www.obeythetestinggoat.com>), e há também uma lista de discussão (<https://groups.google.com/forum/#!forum/obey-the-testing-goat-book>).

Espero que aprecie a leitura deste livro tanto quanto eu gostei de escrevê-lo.

Por que escrevi um livro sobre TDD (Test-Driven Development)

“Quem é você, por que está escrevendo este livro e por que eu deveria lê-lo?” – posso ouvir você perguntando.

Ainda estou bem no início de minha carreira como programador. Dizem que, em qualquer área, vamos de aprendiz a alguém que

empreende uma jornada e, às vezes, em algum momento, nos tornamos um mestre. Eu diria que, no melhor caso, sou um programador empreendendo uma jornada. No entanto, no início de minha carreira, tive a sorte de acabar trabalhando com um bando de fanáticos por TDD, e isso causou um impacto tão grande em meu modo de programação que estou morrendo de vontade de compartilhá-lo com todos. Você pode dizer que tenho o entusiasmo de um recém-convertido e que a experiência de aprendizado ainda é uma lembrança recente para mim, portanto espero ter empatia com os iniciantes.

Quando comecei a aprender Python (com o excelente livro *Dive Into Python*, de Mark Pilgrim), deparei com o conceito de TDD e pensei: “Sim. Definitivamente, posso ver o sentido disso’.’ Quem sabe você não tenha tido uma reação semelhante na primeira vez que ouviu falar de TDD? Parece uma abordagem realmente sensata, um verdadeiro bom hábito a ser adquirido – como passar fio dental regularmente.

Então, participei de meu primeiro grande projeto, e você pode adivinhar o que aconteceu – havia um cliente, havia prazos, muita coisa para fazer, e qualquer boa intenção relacionada ao TDD foi direto para o ralo.

E, na verdade, tudo estava bem. Eu estava bem.

No início.

No início, eu sabia que não precisava de fato do TDD, pois era um site pequeno, e eu poderia facilmente testar se tudo estava funcionando apenas fazendo verificações manuais. Clique neste link [aqui](#), selecione aquele item no menu suspenso *ali* e *isso* deve acontecer. Fácil. Toda essa história de escrever testes dava a impressão de que demoraria *anos* e, além disso, eu me via, do topo de minhas três semanas de experiência adulta em programação, como um programador muito bom. Eu poderia dar conta do recado. Fácil.

Então a temida deusa da Complexidade chegou. Sem demora, ela

me mostrou os limites de minha experiência.

O projeto cresceu. Partes do sistema começaram a depender de outras partes. Fiz o melhor que pude para seguir bons princípios, como DRY (Don't Repeat Yourself, ou Não se Repita), mas isso somente me levou a alguns territórios bem perigosos. Logo eu estava lidando com herança múltipla. Com hierarquias de classe com oito níveis de profundidade. Instruções *eval*.

Passei a ficar com medo de fazer alterações em meu código. Já não tinha mais certeza do que dependia do quê, e o que poderia acontecer se eu mudasse esse código *aqui* – oh, céus, acho que aquela parte ali herda disso – não, não herda, está sobrescrito. Ah, mas isso depende da variável daquela classe. Certo, bem, desde que eu sobrescreva o que está sobrescrito, não deve haver problemas. Vou simplesmente testar – mas testar estava ficando muito mais difícil. Havia muitas seções no site agora, e clicar em todas elas manualmente estava começando a se tornar impraticável. Melhor não mexer muito, esquecer a refatoração, apenas fazer funcionar.

Em breve eu tinha uma confusão de código horrível e deselegante. Novos desenvolvimentos passaram a ser um sofrimento.

Não muito tempo depois, tive a sorte de conseguir um emprego em uma empresa chamada Resolver Systems (atualmente é a PythonAnywhere (<https://www.pythonanywhere.com>)), em que o XP (Extreme Programming, ou Programação Extrema) era a norma. Eles me apresentaram ao TDD rigoroso.

Embora minha experiência anterior certamente tivesse aberto minha mente para as possíveis vantagens dos testes automatizados, eu ainda continuava relutante em cada etapa. “Quero dizer, testar, em geral, pode ser uma boa ideia, mas *fala sério!* Todos esses testes? Alguns parecem uma total perda de tempo... O quê? Testes funcionais, *além* dos testes de unidade? Ora bolas, é um exagero! E esse ciclo de teste/alteração mínima de código/teste do TDD? É simplesmente uma tolice! Não precisamos de todos esses

passinhos de criança! Qual é, podemos ver a resposta correta, por que não vamos simplesmente direto para o final?”

Acredite em mim, eu lançava dúvidas sobre cada regra, sugeria todo tipo de atalho, exigia justificativas para cada aspecto aparentemente sem sentido do TDD, e acabei percebendo toda a sabedoria que havia por trás dele. Perdi a conta do número de vezes que pensei: “Muito obrigado, testes” – quando um teste funcional revelava uma regressão que jamais teríamos previsto ou um teste de unidade evitava que eu cometesse um erro de lógica realmente tolo. Do ponto de vista psicológico, essa abordagem fez com que o processo de desenvolvimento se tornasse muito menos estressante. Ela gera um código com o qual é um prazer trabalhar.

Então, permita-me dizer *tudo* sobre ela!

Objetivos deste livro

Meu objetivo principal é divulgar uma metodologia – uma forma de fazer desenvolvimento web que acho que resulta em aplicações web melhores e em desenvolvedores mais felizes. Não há muito sentido em ter um livro que apenas inclua um conteúdo que você poderia encontrar pesquisando no Google, portanto este livro não é um guia para a sintaxe de Python nem um tutorial sobre o desenvolvimento web *per se*. Em vez disso, espero ensinar você a usar TDD a fim de atingir o nosso objetivo comum e sagrado, de modo mais confiável: *ter um código limpo e que funcione*.

Dito isso, vou me referir constantemente a um exemplo real prático, construindo uma aplicação web do zero com ferramentas como Django, Selenium, jQuery e Mock. Não estou pressupondo nenhum conhecimento prévio deles, portanto você deverá terminar este livro com uma introdução decente sobre essas ferramentas, assim como da disciplina do TDD.

Na Extreme Programming, sempre fazemos programas aos pares, portanto pensei em escrever este livro como se estivesse ao lado de meu eu anterior, tendo que explicar como as ferramentas funcionam

e responder a perguntas sobre por que programamos dessa maneira em particular. Desse modo, se em alguma ocasião eu assumir um tom um pouco paternalista, é porque não sou tão inteligente assim e tive que ser muito paciente comigo mesmo. Se, em algum momento, eu soar como se estivesse na defensiva, é porque sou o tipo de pessoa irritante que sistematicamente discorda de tudo o que os outros dizem e, às vezes, é preciso muitas justificativas para me convencer de algo.

Organização do livro

Dividi este livro em três partes:

Parte I (Capítulos 1-7): O básico

Mergulha diretamente na construção de uma aplicação web simples usando TDD. Começaremos escrevendo um teste funcional (com o Selenium) e então descreveremos o básico sobre Django – modelos, views, templates – com testes de unidade rigorosos em cada etapa. Também apresentarei o *Testing Goat*¹.

Parte II (Capítulo 8-17): Essencial sobre o desenvolvimento web

Aborda alguns dos aspectos mais intrincados, porém inevitáveis, do desenvolvimento web, e mostra como os testes podem nos ajudar com eles: arquivos estáticos, implantação em ambiente de produção, validação de dados de formulários, migrações de banco de dados e o temido JavaScript.

Parte III (Capítulos 18-26): Tópicos mais avançados sobre testes

Simulação (mocking), integração de um sistema de terceiros, fixtures de teste, Outside-In TDD e CI (Continuous Integration, ou Integração Contínua).

Em seguida, vamos pôr um pouco de ordem na casa...

Convenções usadas neste livro

As seguintes convenções tipográficas são usadas neste livro:

Itálico

Indica termos novos, URLs, endereços de email, nomes e extensões de arquivos.

Largura constante

Usada para listagens de programas, assim como em parágrafos para se referir a elementos de programas, como nomes de variáveis ou de funções, bancos de dados, tipos de dados, variáveis de ambiente, comandos e palavras-chave.

Largura constante em negrito

Mostra comandos ou outros textos que devam ser digitados literalmente pelo usuário.

Ocasionalmente utilizarei o símbolo:

[...]

para indicar que parte do conteúdo foi omitido, abreviar porções longas da saída ou ir direto para uma seção relevante.



Este elemento significa uma dica ou sugestão.



Este elemento significa uma observação geral ou uma informação extra.



Este elemento significa um aviso ou uma precaução.

Submissão de erratas (site do autor, em inglês)

Você identificou um equívoco ou um erro de digitação? Os códigos-fontes deste livro estão disponíveis no GitHub, e sempre fico feliz em receber problemas e pull requests: <https://github.com/hjwp/Book-TDD-Web-Dev-Python/>.

Uso de exemplos de código de acordo com a política da O'Reilly

Os códigos de exemplo estão disponíveis em <https://github.com/hjwp/book-example/>; lá você encontrará branches para cada capítulo (por exemplo, https://github.com/hjwp/book-example/tree/chapter_unit_test_first_view). Você verá uma lista completa, além de algumas sugestões sobre modos de trabalhar com esse repositório, no Apêndice J.

Este livro está aqui para ajudá-lo a fazer seu trabalho. De modo geral, se este livro incluir exemplos de código, você poderá usar o código em seus programas e em sua documentação. Você não precisa nos contatar para pedir permissão, a menos que esteja reproduzindo uma parte significativa do código. Por exemplo, escrever um programa que use diversas partes de código deste livro não requer permissão. No entanto, vender ou distribuir um CD-ROM de exemplos de livros da O'Reilly requer permissão. Responder a uma pergunta mencionando este livro e citar o código de exemplo não requer permissão. Em contrapartida, incluir uma quantidade significativa de código de exemplos deste livro na documentação de seu produto requer permissão.

Agradecemos, mas não exigimos, atribuição. Uma atribuição geralmente inclui o título, o autor, a editora e o ISBN. Por exemplo: “*Test-Driven Development with Python*, segunda edição, de Harry J. W. Percival (O'Reilly). Copyright 2017 Harry Percival, 978-1-491-95870-4”.

Se você achar que o seu uso dos exemplos de código está além do razoável ou da permissão concedida, sinta-se à vontade para nos contatar em permissions@oreilly.com.

Como entrar em contato conosco

Envie seus comentários e suas dúvidas sobre este livro à editora pelo email: novatec@novatec.com.br.

Temos uma página web para este livro, na qual incluímos erratas, exemplos e quaisquer outras informações adicionais.

- Página da edição traduzida para o português

<http://www.novatec.com.br/livros/tdd-com-python>

- Página da edição original em inglês

http://bit.ly/tdd_py_2e

Para obter mais informações sobre os livros da Novatec, acesse nosso site em: *<http://www.novatec.com.br>*.

¹ O autor do livro pensou numa estratégia de marketing que envolveu escolher o animal da capa (um bode) e difundir esse slogan “obey the testing goat”; em uma tradução livre “siga o bode dos testes”. Optamos por manter o termo em inglês na edição brasileira.

Pré-requisitos e suposições

Eis uma descrição do que estou supondo sobre você e sobre o que já sabe, assim como dos softwares que deverá ter prontos e instalados em seu computador.

Python 3 e programação

Tentei escrever este livro com os iniciantes em mente, mas, mesmo que a programação lhe seja uma novidade, estou supondo que já conheceu o básico sobre Python. Então, se você ainda não o fez, leia um tutorial de Python para iniciantes ou adquira um livro introdutório, como *Dive Into Python* (<http://www.diveintopython.net/>) ou *Learn Python the Hard Way* (<http://learnpythonthehardway.org/>), ou, somente por diversão, *Invent Your Own Computer Games with Python* (<http://inventwithpython.com/>) – todos são introduções excelentes para o assunto.

Se você é um programador experiente, porém sem experiência com Python, deverá sair-se bem. Python é agradavelmente simples de entender.

Estou usando *Python 3* neste livro. Quando escrevi a primeira edição em 2013-14, o Python 3 já existia há vários anos, e o mundo estava próximo do ponto de inflexão em que essa linguagem era a opção preferida. Você poderá acompanhar este livro no Mac, no Windows ou no Linux. Instruções detalhadas de instalação para cada sistema operacional serão apresentadas a seguir.



Este livro foi testado com o Python 3.6. Se estiver usando uma versão mais antiga, você verá pequenas diferenças (a sintaxe de f-string, por exemplo), portanto será melhor fazer um upgrade, se puder.

Não recomendaria tentar usar o Python 2, pois as diferenças são mais significativas. Você ainda será capaz de pôr em prática todas as lições aprendidas neste livro caso o seu próximo projeto seja em Python 2. No entanto, gastar tempo para descobrir se o motivo pelo qual a saída de seu programa parece diferente da minha é o Python 2 ou se é porque você cometeu um erro não será um tempo gasto de forma produtiva.

Se você estiver pensando em usar PythonAnywhere (<http://www.pythonanywhere.com>) (a startup de PaaS na qual eu trabalho), em vez de utilizar um Python instalado localmente, dê uma olhada rápida no Apêndice A antes de começar.

Qualquer que seja o caso, espero que você tenha acesso ao Python, saiba como iniciá-lo a partir de uma linha de comando, como editar um arquivo Python e executá-lo. Novamente, dê uma olhada nos três livros que recomendei antes, caso tenha alguma dúvida.



Se você já tem o Python 2 instalado e está preocupado com o fato de que instalar o Python 3 vá causar problemas de alguma forma, não fique! O Python 3 e o 2 podem coexistir pacificamente no mesmo sistema, em particular se você estiver usando um virtualenv – e é o que faremos.

Como o HTML funciona

Também estou supondo que você tem um domínio básico sobre como a web funciona – o que é HTML, o que é uma requisição POST, e assim por diante. Se não estiver certo disso, será necessário encontrar um tutorial básico sobre HTML; há alguns em <http://www.webplatform.org/>. Se souber como criar uma página HTML em seu computador, vê-la em seu navegador, além de entender o que é um formulário e como ele pode funcionar, provavelmente você não terá problemas.

Django

O livro utiliza o framework Django, que (provavelmente) é o framework web mais consagrado no mundo Python. Escrevi o livro supondo que o leitor não tem nenhum conhecimento prévio de Django, mas, se Python e desenvolvimento web e testes forem novidade para você, ocasionalmente poderá achar que há assuntos e conjuntos de conceitos demais para testar e absorver. Se for esse o caso, recomendo que faça uma pausa no livro e dê uma olhada em um tutorial de Django. DjangoGirls (<https://tutorial.djangogirls.org/>) é o melhor e mais agradável tutorial para iniciantes que conheço. O tutorial oficial (<https://docs.djangoproject.com/en/1.11/intro/tutorial01/>) também é excelente para programadores mais experientes.

Continue lendo para ver instruções sobre a instalação do Django.

JavaScript

Há um pouco de JavaScript na segunda metade do livro. Se você não o conhece, não se preocupe com ele até lá; se você se vir um pouco confuso, recomendarei alguns guias na ocasião.

Uma observação sobre IDEs

Se você vem do mundo Java ou de .NET, talvez esteja inclinado a usar um IDE para sua programação com Python. Os IDEs têm todo tipo de ferramentas úteis, incluindo integração com VCS (Version Control System, ou Sistema de Controle de Versões), e há alguns excelentes por aí para Python. Eu mesmo já usei um IDE quando estava começando e o achei muito útil em meus primeiros projetos.

Posso sugerir (e é apenas uma sugestão) que você *não* utilize um IDE, pelo menos durante este tutorial? Os IDEs são muito menos necessários no mundo Python, e escrevi este livro todo partindo do pressuposto de que você estaria usando apenas um editor de texto básico e uma linha de comando. Às vezes é tudo que você terá – quando estiver trabalhando em um servidor, por exemplo –, portanto sempre vale a pena aprender a usar as ferramentas básicas antes e entender como elas funcionam. Será um conhecimento que você sempre terá, mesmo que decida voltar para o seu IDE e todas as suas ferramentas

úteis, depois que terminar este livro.

Instalações de softwares necessários

Além do Python, você precisará do seguinte:

O navegador web Firefox

O Selenium, na verdade, pode controlar qualquer um dos principais navegadores, porém o Firefox é o melhor para ser usado como exemplo, pois pode ser utilizado em diversas plataformas de forma confiável e, como bônus, está menos atrelado a interesses corporativos.

O sistema de controle de versões Git

Está disponível para qualquer plataforma em <http://git-scm.com/>. No Windows, ele vem com a linha de comando *Bash*, necessária no livro.

Um virtualenv com Python 3, Django 1.11 e Selenium 3

As ferramentas virtualenv e pip do Python agora estão incluídas no Python 3.4+ (nem sempre costumavam estar, portanto esse fato merece uma grande comemoração). Instruções detalhadas sobre como preparar o seu virtualenv serão apresentadas a seguir.

Geckodriver

É o driver que nos permitirá controlar remotamente o Firefox por meio do Selenium. Recomendarei um link para download em “Instalando o Firefox e o Geckodriver”.

Observações sobre o Windows

Os usuários de Windows às vezes poderão se sentir um pouco negligenciados no mundo do código aberto, pois o MacOS e o Linux são predominantes, fazendo com que seja fácil esquecer que há um mundo fora do paradigma Unix. Barras invertidas como separadores de diretório? Letras para drives? O quê? Apesar disso, ainda é totalmente possível acompanhar este livro no

Windows. Eis algumas dicas:

1. Ao instalar o Git para Windows, não se esqueça de selecionar “*Run Git and included Unix tools from the Windows command prompt*” (Executar o Git e as ferramentas Unix incluídas a partir do prompt de comandos do Windows). Então você terá acesso a um programa chamado “Git Bash”. Utilize-o como seu principal prompt de comandos, e você terá todas as ferramentas úteis de linha de comando do GNU, como ls, touch e grep, além de barras para frente como separadores de diretório.
2. Também no instalador do Git, selecione “*Use Windows’ default console*” (Usar o console default do Windows); caso contrário, o Python não funcionará de forma apropriada na janela do Git-Bash.
3. Quando instalar o Python 3, a menos que você já tenha o Python 2 e queira mantê-lo como o seu default, marque a opção “*Add Python 3.6 to PATH*” (Adicionar Python 3.6 ao PATH), como vemos na Figura P.1, para que você execute o Python facilmente a partir da linha de comando.



Figura P.1 – Adicione Python ao path do sistema no instalador.



O teste para tudo isso é ver se você é capaz de acessar um prompt de comandos do Git-Bash e simplesmente executar python ou pip a partir de qualquer pasta.

Observações sobre o MacOS

O MacOS é um pouco mais razoável que o Windows, embora ter o pip instalado ainda fosse um tanto quanto desafiador até recentemente. Desde o lançamento da versão 3.4, a situação agora se tornou mais simples:

- O Python 3.6 deve ser instalado sem muita dificuldade a partir de seu instalador, que pode ser baixado de <http://www.python.org>. O pip será instalado automaticamente também.
- O instalador do Git também deverá “simplesmente funcionar”.

De modo semelhante ao Windows, o teste para tudo isso deve garantir que você abra um terminal e simplesmente execute git, python3 ou pip de qualquer lugar. Se você deparar com qualquer problema, os termos de pesquisa “system path” (path do sistema) e “command not found” (comando não encontrado) deverão fornecer recursos bons para a resolução de problemas.



Talvez você queira dar uma olhada também no Homebrew (<http://brew.sh/>). Essa costumava ser a única maneira confiável de instalar várias ferramentas do tipo Unix (incluindo Python 3) em um Mac¹. Embora o instalador usual de Python atualmente não apresente problemas, talvez você ache o Homebrew útil no futuro. Ele exige que você faça download de todos os 1.1 GB de Xcode, mas você também terá um compilador C, o que será um efeito colateral útil.

Editor default e outras configurações básicas do Git

Apresentarei instruções passo a passo para o Git, mas talvez seja uma boa ideia definir algumas configurações agora. Por exemplo, quando executar o seu primeiro commit, o *vi* será apresentado por default, e, nesse ponto, talvez você não tenha a mínima ideia do que fazer com ele. Bem, assim como o *vi* tem dois modos, você terá duas opções. Uma é aprender um mínimo de comandos do *vi* (*pressione a tecla i para entrar em modo de inserção, digite seu texto, teclre <Esc> para voltar ao modo normal e então grave o*

arquivo e saia com : wq<Enter>). Com isso, você terá se associado à grande fraternidade de pessoas que conhecem esse editor de texto antigo e reverenciado.

Ou você pode enfaticamente se recusar a ser envolvido em um retrocesso ridículo aos anos 1970 e configurar o Git para usar um editor de sua preferência. Saia do vi usando <Esc>, seguido de :q! e então altere o editor default do Git. Consulte a documentação sobre a sua configuração básica (<http://git-scm.com/book/en/Customizing-Git-Git-Configuration>).

Instalando o Firefox e o GeckoDriver

O Firefox está disponível como um download para Windows e MacOS em <https://www.mozilla.org/firefox/>. No Linux, é provável que você já o tenha instalado; caso contrário, seu gerenciador de pacotes o terá.

O GeckoDriver está disponível em <https://github.com/mozilla/geckodriver/releases>. Você precisa fazer o download, extraí-lo e colocá-lo em algum lugar no path de seu sistema.

- no MacOS e no Linux, um local conveniente para colocá-lo é `~/.local/bin`;
- no Windows, coloque-o na pasta *Scripts* de Python.

Para testar se isso está funcionando, abra um console do Bash; você deverá ser capaz de executar:

```
geckodriver --version  
geckodriver 0.17.0
```

The source code of this program is available at
<https://github.com/mozilla/geckodriver>.

This program is subject to the terms of the Mozilla Public License 2.0.
You can obtain a copy of the license at <https://mozilla.org/MPL/2.0/>.

Se esse comando não funcionar, talvez seja porque `~/.local/bin` não

está em seu PATH (poderá se aplicar a alguns sistemas Mac e Linux). É uma boa ideia ter essa pasta em seu path porque é o local em que Python instalará seus itens quando você usar `pip install --user`. Eis o modo de adicioná-lo em seu `.bashrc`:²

```
echo 'PATH=~/.local/bin:$PATH' >> ~/.bashrc
```

Feche o seu terminal e abra-o novamente para ver se `geckodriver --version` agora funciona.

Configurando o seu virtualenv

Um virtualenv (abreviatura de *virtual environment*, isto é, ambiente virtual) de Python é o modo de configurar seu ambiente para diferentes projetos Python. Ele permite que você utilize pacotes diferentes (por exemplo, versões distintas de Django e até mesmo de Python) em cada projeto. Além do mais, como você não está instalando recursos válidos para todo o sistema, isso significa que não será necessário ter permissões de root.

O virtualenv foi incluído em Python a partir da versão 3.4, mas sempre recomendo uma ferramenta auxiliar chamada “virtualenvwrapper”. Vamos instalá-la antes (não importa para qual versão de Python você a instalará):

```
# no Windows
pip install virtualenvwrapper
# No MacOS / Linux
pip install --user virtualenvwrapper
# então faça o Bash carregar o virtualenvwrapper automaticamente
echo "source virtualenvwrapper.sh" >> ~/.bashrc
source ~/.bashrc
```



No Windows, virtualenvwrapper só funcionará no shell de “Git-Bash”, e não da linha de comando normal.

virtualenvwrapper mantém todos os seus virtualenvs em um só lugar, além de disponibilizar ferramentas convenientes para ativá-los e desativá-los.

Vamos criar um virtualenv chamado “superlists”³ com Python 3 instalado:

```
# No MacOS/Linux:  
mkvirtualenv --python=python3.6 superlists  
# No Windows  
mkvirtualenv --python=py -3.6 -c"import sys; print(sys.executable)"  
superlists  
# (um pequeno hack para garantir que tenhamos um virtualenv python 3.6)
```

Ativando e desativando o virtualenv

Sempre que trabalhar com o livro, você deve garantir que seu virtualenv esteja “ativo”. Geralmente você poderá inferir isso, pois verá (superlists) entre parênteses em seu prompt. Algo como:

```
$  
(superlists) $
```

Logo depois de criar seu virtualenv, ele deverá estar ativo. Você pode conferir executando which python:

```
(superlists) $ which python  
/home/harry/.virtualenvs/superlists/bin/python  
# (no Windows, será algo como  
# /C/Users/IEUser/.virtualenvs/superlists/Scripts/python)
```

```
(superlists) $ deactivate  
$ which python  
/usr/bin/python  
$ python --version  
Python 2.7.12 # para mim, fora de meu virtualenv, o default de "python"  
# é Python 2.
```

```
$ workon superlists  
(superlists) $ which python  
/home/harry/.virtualenvs/superlists/bin/python  
(superlists) $ python --version  
Python 3.6.0
```



Para ativar seu virtualenv, utilize workon superlists. Para verificar se ele está ativo, observe se (superlists) \$ está em seu prompt de comandos, ou

execute which python.

Instalando Django e Selenium

Instalaremos o Django 1.11 e a versão mais recente do Selenium, que é o Selenium 3:

```
(superlists) $ pip install "django<1.12" "selenium<4"
Collecting django==1.11.3
  Using cached Django-1.11.3-py2.py3-none-any.whl
Collecting selenium<4
  Using cached selenium-3.4.3-py2.py3-none-any.whl
Installing collected packages: django, selenium
Successfully installed django-1.11.3 selenium-3.4.3
```

Algumas mensagens de erro prováveis de ver quando você inevitavelmente se esquecer de ativar o seu virtualenv

Caso os virtualenvs sejam novidade para você – ou mesmo se não forem, para sermos francos – em algum momento, é *certo* que você se esquecerá de ativá-los, e então ficará encarando uma mensagem de erro. Acontece comigo o tempo todo. Eis algumas mensagens em que você deverá prestar atenção:

```
ImportError: No module named selenium
```

Ou:

```
ImportError: No module named django.core.management
```

Como sempre, procure aquele (superlists) em seu prompt de comandos, e um rápido `workon superlists` provavelmente é o que será necessário para que ele funcione de novo.

A seguir apresentaremos mais algumas mensagens, por garantia:

```
bash: workon: command not found
```

Isso significa que você pulou um passo anterior e não adicionou o `virtualenvwrapper` em seu `.bashrc`. Encontre os comandos `echo source virtualenvwrapper.sh` anteriores e execute-os novamente.

```
'workon' is not recognized as an internal or external command,
```

operable program or batch file.

Isso significa que você iniciou o prompt de comandos default do Windows, isto é, o cmd, em vez do Git-Bash. Feche-o e abra esse último.

Boa programação!



Essas instruções não serviram para você? Ou você tem instruções melhores? Entre em contato por meio de obeythetestinggoat@gmail.com!

- 1 Apesar disso, não recomendaria instalar o Firefox usando o Homebrew: o brew coloca o binário do Firefox em um local estranho e confunde o Selenium. Você pode contornar o problema, mas é simplesmente mais fácil instalar o Firefox da forma usual.
- 2 `.bashrc` é um arquivo de inicialização do Bash, que está em sua pasta home. É executado sempre que você iniciar o Bash.
- 3 Posso ouvir você perguntando: por que superlists? Sem spoilers! Você descobrirá no capítulo 1.

Vídeo complementar

Gravei uma série de vídeos (em inglês) com dez partes para acompanhar este livro (<http://oreil.ly/1svTFqB>)¹. Os vídeos abordam o conteúdo da Parte I. Se você acha que aprende bem com um conteúdo em vídeo, incentivo a dar uma olhada neles. Complementando o que está no livro, os vídeos devem dar uma noção de como é o “fluxo” do TDD, com a alternância entre testes e código, e explicando o processo de raciocínio à medida que prosseguimos.

Além do mais, estou usando uma camiseta amarela maravilhosa.



¹ O vídeo não foi atualizado para a segunda edição (em inglês), porém o conteúdo é praticamente o mesmo.

Agradecimentos

Tenho muitas pessoas a agradecer, e sem elas este livro jamais teria se tornado realidade, e/ou teria sido até pior.

Agradeço inicialmente a “Greg” de \$OUTRA_EDITORA, que foi a primeira pessoa a me incentivar e a me fazer acreditar que o livro realmente poderia ser escrito. Apesar de seus empregadores acabarem mostrando um ponto de vista demasiadamente retrógrado sobre copyright, sou eternamente grato por você ter acreditado em mim.

Agradeço a Michael Foord, outro ex-funcionário da Resolver Systems, por ter originalmente me inspirado, ao escrever, ele mesmo, um livro, e obrigado pelo seu apoio constante ao projeto. Agradeço também ao meu chefe Giles Thomas por ingenuamente permitir que outro de seus funcionários escrevesse um livro (embora creio que agora ele tenha alterado o contrato padrão de contratação de funcionários para dizer que “livros não são permitidos”). Obrigado também por sua sabedoria constante e por me colocar no caminho dos testes.

Agradeço aos meus outros colegas Glenn Jones e Hansel Dunlop o apoio de valor inestimável e a paciência com minha conversa repetitiva durante o último ano.

Agradeço à minha esposa Clementine e às minhas duas famílias; sem seu apoio e paciência, eu jamais teria conseguido. Peço desculpas por todo o tempo que passei com o nariz no computador, nos momentos que deveriam ter sido ocasiões familiares memoráveis. Quando me propus a escrever o livro, não tinha a menor ideia do que ele faria com a minha vida (“Você quer dizer, escrever em meu tempo livre? Parece razoável...”). Não poderia ter feito isso sem você.

Agradeço aos meus revisores técnicos, Jonathan Hartley, Nicholas Tollervey e Emily Bache, os incentivos e o feedback de valor inestimável. Especialmente a Emily, que realmente leu todos os capítulos de forma minuciosa. Dou crédito parcial a Nick e Jon, mas isso ainda deve ser interpretado como gratidão eterna. Ter todos vocês por perto deixou todo esse empreendimento menos solitário. Sem vocês, o livro teria sido apenas um pouco mais do que as divagações sem sentido de um idiota.

Obrigado a todas as outras pessoas que concederam parte de seu tempo para me dar algum feedback sobre o livro, somente pela bondade em seus corações: Gary Bernhardt, Mark Lavin, Matt O'Donnell, Michael Foord, Hynek Schlawack, Russell Keith-Magee, Andrew Godwin, Kenneth Reitz e Nathan Stocks. Obrigado por serem muito mais inteligentes do que eu e por evitar que eu dissesse muitas coisas estúpidas. Naturalmente ainda restaram muitas coisas estúpidas no livro, pelas quais vocês não têm absolutamente nenhuma responsabilidade.

Agradeço à minha editora Meghan Blanchette por conduzir um serviço de forma muito amigável e simpática, e por manter o livro no caminho certo, tanto no que diz respeito a prazos quanto para refrear minhas ideias mais tolas. Agradeço a todos os demais na O'Reilly a ajuda, incluindo Sarah Schneider, Kara Ebrahim e Dan Fauxsmith por terem me deixado manter o meu inglês britânico. Obrigado a Charles Roumeliotis a ajuda com estilo e gramática. Talvez jamais tenhamos o mesmo ponto de vista sobre os méritos das regras para aspas/pontuação da Chicago School, mas certamente estou feliz por ter tido você por perto. Obrigado também ao departamento de design por ter conseguido um bode (goat) para a capa!

Em especial, obrigado a todos os leitores da Versão Preliminar por toda a ajuda na identificação de erros de digitação, os feedbacks e as sugestões, por todas as maneiras pelas quais vocês ajudaram a suavizar a curva de aprendizado no livro e, acima de tudo, as

palavras gentis de incentivo e apoio, que fizeram com que eu continuasse o trabalho. Obrigado a Jason Wirth, Dave Pawson, Jeff Orr, Kevin De Baere, crainbf, dsisson, Galeran, Michael Allan, James O'Donnell, Marek Turnovec, SoonerBourne, julz, Cody Farmer, William Vincent, Trey Hunner, David Souther, Tom Perkin, Sorchia Bowler, Jon Poler, Charles Quast, Siddhartha Naithani, Steve Young, Roger Camargo, Wesley Hansen, Johansen Christian Vermeer, Ian Laurain, Sean Robertson, Hari Jayaram, Bayard Randel, Konrad Korżel, Matthew Waller, Julian Harley, Barry McClendon, Simon Jakobi, Angelo Cordon, Jyrki Kajala, Manish Jain, Mahadevan Sreenivasan, Konrad Korżel, Deric Crago, Cosmo Smith, Markus Kemmerling, Andrea Costantini, Daniel Patrick, Ryan Allen, Jason Selby, Greg Vaughan, Jonathan Sundqvist, Richard Bailey, Diane Soini, Dale Stewart, Mark Keaton, Johan Wärlander, Simon Scarfe, Eric Grannan, Marc-Anthony Taylor, Maria McKinley, John McKenna, Rafał Szymański, Roel van der Goot, Ignacio Reguero, TJ Tolton, Jonathan Means, Theodor Nolte, Jungsoo Moon, Craig Cook, Gabriel Ewilazarus, Vincenzo Pandolfo, David "farbish2", Nico Coetzee, Daniel Gonzalez, Jared Contrascere, Zhao 赵亮, e a vários outros. Se eu pulei seu nome, você tem toda razão de ficar ofendido; sou extremamente grato a você também, portanto escreva para mim, e tentarei compensá-lo do modo que puder.

Por fim, agradeço a você, meu leitor mais recente, por ter decidido dar uma olhada no livro! Espero que goste dele.

Agradecimentos adicionais para a segunda edição

Agradeço a segunda edição (em inglês) à minha maravilhosa editora Nan Barber, e a Susan Conant, Kristen Brown e a toda a equipe da O'Reilly. Obrigado novamente a Emily e a Jonathan pela revisão técnica, assim como a Edward Wong pelas suas observações bastante minuciosas. Qualquer erro ou informação inadequada restantes são de minha total responsabilidade.

Agradeço também aos leitores da edição gratuita que colaboraram com comentários, sugestões e até mesmo com alguns pull requests. Com certeza devo ter me esquecido de alguns na lista a seguir, portanto peço desculpas se o seu nome não estiver aqui, mas obrigado a Emre Gonulates, Jésus Gómez, Jordon Birk, James Evans, Iain Houston, Jason DeWitt, Ronnie Raney, Spencer Ogden, Suresh Nimbalkar, Darius, Caco, LeBodro, Jeff, wasabigeek, joegnis, Lars, Mustafa, Jared, Craig, Sorcha, TJ, Ignacio, Roel, Justyna, Nathan, Andrea, Alexandr, bilyanhadzhi, mosegontar, sfarzy, henziger, hunterji, das-g, juanriaza, GeoWill, Windsooon, gonulate, e a vários outros.

PARTE I

Básico sobre TDD e Django

Na primeira parte, apresentarei o básico sobre TDD (*Test-Driven Development*, ou Desenvolvimento Orientado a Testes). Construiremos uma verdadeira aplicação web do zero, escrevendo os testes antes, em cada etapa.

Abordaremos os testes funcionais com o Selenium, assim como os testes de unidade, e veremos a diferença entre ambos. Apresentarei o fluxo de trabalho do TDD, que chamo de ciclo de testes de unidade/código. Também faremos um pouco de refatoração e veremos como ela se encaixa no TDD. Pelo fato de ser absolutamente essencial em uma engenharia de software séria, também utilizarei um sistema de controle de versões (Git). Discutiremos como e quando fazer commits e integrá-los ao TDD e ao fluxo de trabalho do desenvolvimento web.

Usaremos o Django, (provavelmente) o framework web para Python mais popular do mundo. Tentei apresentar os conceitos do Django lentamente, um de cada vez, e disponibilizar vários links para leituras complementares. Se você é totalmente inexperiente com o Django, recomendo fortemente que invista tempo nessas leituras. Se achar que está um pouco perdido, reserve algumas horas para ler o tutorial oficial do Django e então retorne ao livro.

Você também conhecerá o Testing Goat (bode dos testes)...



Tome cuidado com a operação de Copiar e Colar

Se estiver trabalhando com uma versão digital do livro, é natural que queira copiar e colar listagens de código do livro quando estiver trabalhando com

ele. Será muito melhor se você não fizer isso: digitar o código manualmente faz com que as informações entrem em sua memória, e elas parecerão muito mais reais. Inevitavelmente você cometerá os ocasionais erros de digitação, e depurá-los é importante para aprender.

Além disso, você perceberá que as idiossincrasias do formato PDF implicam frequentes ocorrências estranhas quando você tenta copiar/colar a partir desse formato...

CAPÍTULO 1

Configurando o Django com um teste funcional

O TDD não é algo que surge naturalmente. É uma disciplina, como uma arte marcial, e, assim como em um filme de Kung Fu, você precisa de um mestre mal-humorado e prepotente para obrigá-lo a ter disciplina. O nosso mestre é o Testing Goat (bode dos testes).

Obedeça ao Testing Goat! Não faça nada até ter um teste

O Testing Goat é o mascote não oficial de TDD da comunidade de testes de Python. Provavelmente ele apresenta diferentes significados para pessoas diferentes, mas, para mim, o Testing Goat é uma voz em minha mente que me mantém no Verdadeiro Caminho dos Testes – como um daqueles anjinhos ou diabinhos que aparecem em cima de seu ombro nos desenhos animados, mas com um conjunto muito específico de preocupações. Com este livro, espero colocar o Testing Goat em sua mente também.

Decidimos construir um site, apesar de ainda não sabermos muito bem o que ele fará. Geralmente, o primeiro passo em um desenvolvimento web é ter seu framework web instalado e configurado. *Faça download disso, instale aquilo, configure mais isso, execute o script...*; porém, o TDD exige uma mentalidade diferente. Quando estiver fazendo TDD, tenha sempre o Testing Goat dentro de você – considerando como os caprinos são determinados – balindo: “Teste antes, teste antes!”.

Em TDD o primeiro passo é sempre o mesmo: *escreva um teste*.

Em primeiro lugar, escrevemos o teste, e *depois* o executamos e verificamos que ele falha conforme esperado. *Somente então* prosseguimos e desenvolvemos parte de nossa aplicação. Repita isso para si mesmo como se estivesse balindo. É o que eu faço.

Outro fato sobre os caprinos é que eles dão um passo de cada vez. Veja bem, é por isso que eles raramente caem das montanhas, independentemente do quão íngremes elas sejam, como podemos ver na Figura 1.1.



Figura 1.1 – Os caprinos são mais ágeis do que pensamos. (Fonte: Caitlin Stewart, no Flickr

(<http://www.flickr.com/photos/caitlinstewart/2846642630/>)

Continuaremos com pequenos passinhos; usaremos o *Django*, um framework web popular para Python, a fim de construir nossa aplicação.

Nossa primeira atitude será nos certificarmos de que o Django está instalado e pronto para que possamos trabalhar com ele. O *modo*

como verificaremos isso é confirmando que podemos iniciar o servidor de desenvolvimento do Django e realmente o ver servindo uma página web no navegador em nosso microcomputador local. Usaremos a ferramenta de automação de navegação *Selenium* para isso.

Crie um novo arquivo Python chamado *functional_tests.py* no local em que você quer manter o código de seu projeto e insira o código exibido a seguir. Se sentir vontade de dar alguns balidos enquanto faz isso, talvez ajude:

functional_tests.py

```
from selenium import webdriver

browser = webdriver.Firefox()
browser.get('http://localhost:8000')

assert 'Django' in browser.title
```

Esse é o nosso primeiro *FT* (Functional Test, ou Teste Funcional); em breve, falarei mais sobre o que quero dizer com testes funcionais e como eles se comparam aos testes de unidade. Por enquanto, é suficiente garantir que compreendemos o que esse teste faz:

- inicia um “webdriver” Selenium para apresentar uma verdadeira janela do navegador Firefox;
- usa essa janela para abrir uma página web que esperamos que vá ser servida pelo microcomputador local;
- verifica (fazendo uma asserção de teste) se a página tem a palavra “Django” no título.

Vamos tentar executar este teste:

```
$ python functional_tests.py
File ".../selenium/webdriver/remote/webdriver.py", line 268, in get
  self.execute(Command.GET, {'url': url})
File ".../selenium/webdriver/remote/webdriver.py", line 256, in execute
  self.error_handler.check_response(response)
```

```
File ".../selenium/webdriver/remote/errorhandler.py", line 194, in
check_response
    raise exception_class(message, screen, stacktrace)
selenium.common.exceptions.WebDriverException: Message: Reached error
page: abo
ut:neterror?e=connectionFailure&u=http%3A//localhost%3A8000/[...]
```

Você deverá ver uma janela do navegador aparecer e tentar abrir *localhost:8000*; a página de erro “Unable to connect” (Incapaz de se conectar) será exibida. Se você retornar ao seu console, verá a mensagem de erro enorme e horrível informando-nos que o Selenium acessou uma página de erro. E então você provavelmente ficará irritado com o fato de a janela do Firefox ter sido deixada lá em seu desktop para que você faça a limpeza. Corrigiremos isso mais tarde!



Se, em vez disso, você vir um erro de tentativa de importar o Selenium ou de encontrar o “geckodriver”, talvez seja necessário retornar e consultar novamente a seção “Pré-requisitos e suposições”.

Por enquanto, porém, temos um *teste com falha*, e isso significa que podemos começar a desenvolver nossa aplicação.

Adeus aos números romanos!

Há tantas introduções ao TDD que usam números romanos como exemplo que passou a ser uma piada corrente – eu mesmo comecei a escrever um. Se estiver curioso, poderá vê-lo em minha página do GitHub (<https://github.com/hjwp/tdd-roman-numeral-calculator/>).

Os números romanos, como exemplo, são, ao mesmo tempo, bons e ruins. É um bom problema “lúdico”, razoavelmente limitado em escopo, e podemos explicar o TDD muito bem com ele.

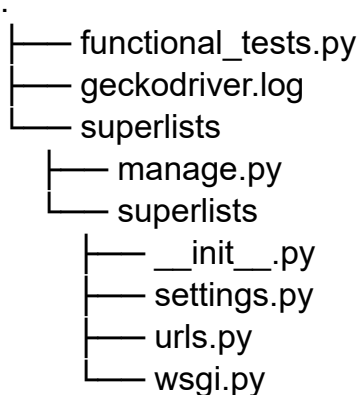
O problema é que pode ser difícil relacioná-lo com o mundo real. É por isso que decidi usar a construção de uma aplicação web real, partindo do zero, como exemplo. Embora seja uma aplicação web simples, minha esperança é que isso facilite a transição para o próximo projeto de verdade.

Deixando o Django pronto para funcionar

Como você certamente já leu a seção “Pré-requisitos e suposições” a essa altura, o Django deve estar instalado. O primeiro passo para que o Django esteja pronto e executando é criar um *projeto*, que será o contêiner principal para o nosso site. O Django disponibiliza uma pequena ferramenta de linha de comando para isso:

```
$ django-admin.py startproject superlists
```

Esse comando criará uma pasta chamada *superlists*, e um conjunto de arquivos e subpastas contidos aí:



Sim, há uma pasta chamada *superlists* dentro de uma pasta chamada *superlists*. Embora um pouco confuso, é apenas um daqueles fatos para os quais há bons motivos quando olhamos para trás e vemos a história de Django. Por enquanto, o importante é saber que a pasta *superlists/superlists* serve para itens que se aplicam ao projeto como um todo – como *settings.py*, que é usado para armazenar informações sobre configurações globais para o site.

Você também deve ter percebido a presença de *manage.py*. É o canivete suíço de Django, e uma das tarefas que ele consegue fazer é executar um servidor de desenvolvimento. Vamos testar isso agora. Execute um `cd superlists` para acessar a pasta *superlists* de nível mais alto (trabalharemos bastante a partir dessa pasta) e então execute:

```
$ python manage.py runserver
```

Performing system checks...

System check identified no issues (0 silenced).

You have 13 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, contenttypes, sessions. Run 'python manage.py migrate' to apply them.

Django version 1.11.3, using settings 'superlists.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.



É seguro ignorar a mensagem sobre “unapplied migrations” (migrações não aplicadas) por enquanto. Veremos as migrações no Capítulo 5.

Esse é o nosso servidor de desenvolvimento Django funcionando em nossa máquina. Deixe-o lá e abra outro shell de comandos. Nesse shell, podemos tentar executar nosso teste novamente (a partir da pasta em que começamos):

```
$ python functional_tests.py  
$
```



Como acabamos de abrir uma nova janela do terminal, será necessário ativar o seu virtualenv com workon superlists para que isso funcione.

Não há muita ação na linha de comando, mas você deve prestar atenção em dois pontos: em primeiro lugar, não houve nenhum AssertionError horrível e, em segundo, a janela do Firefox apresentada pelo Selenium continha uma página com aspecto diferente.

Bem, pode não parecer muito, mas esse foi o nosso primeiro teste em que passou! Viva!

Se tudo parecer mágico demais, como se não fosse muito real, por que não damos uma olhada no servidor de desenvolvimento manualmente, abrindo um navegador web por conta própria e acessando *http://localhost:8000?* Você deverá ver algo semelhante à Figura 1.2.

Se quiser, poderá encerrar agora o servidor de desenvolvimento e retornar ao shell original usando Ctrl-C.

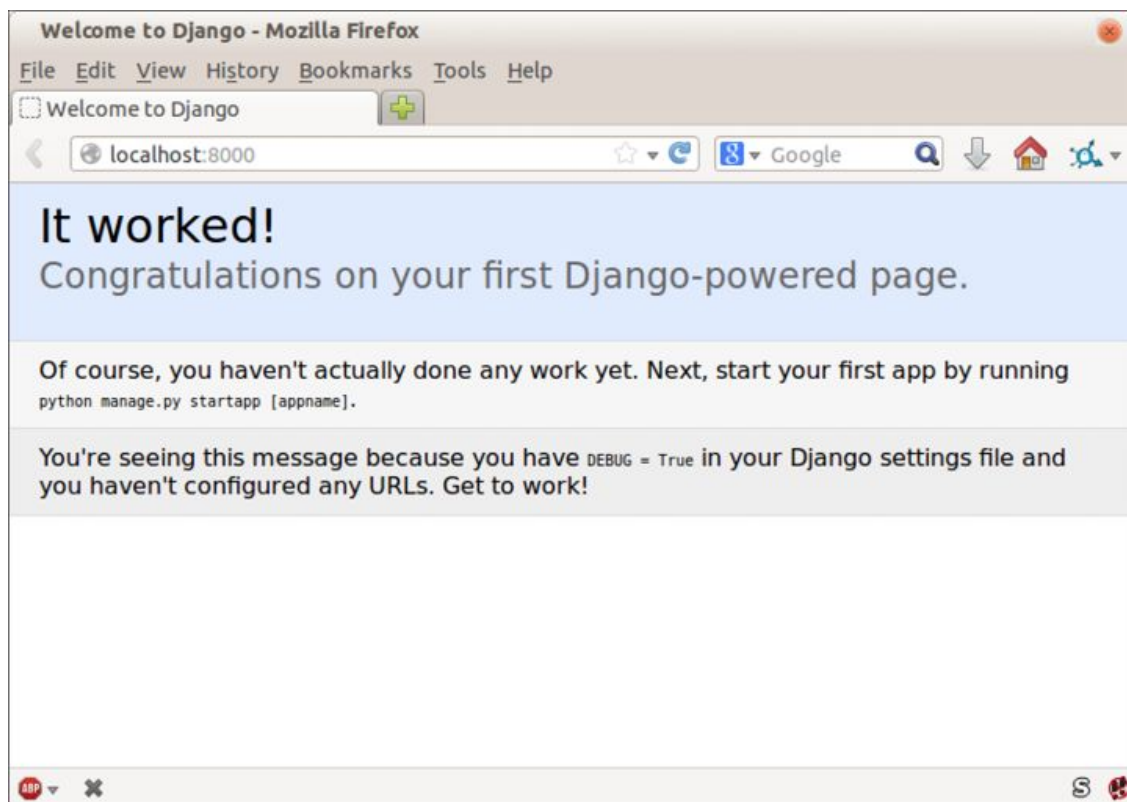


Figura 1.2 – Funcionou!

Iniciando um repositório no Git

Há uma última tarefa a ser feita antes de encerrarmos o capítulo: começar a fazer commit de nosso trabalho em um VCS (Version Control System, ou Sistema de Controle de Versões). Se você é um programador experiente, não precisará me ouvir fazendo pregações sobre controle de versões, mas se esse assunto lhe for uma novidade, por favor, acredite em mim quando digo que ter um VCS é obrigatório. Assim que seu projeto tiver mais que algumas semanas e algumas linhas de código, ter uma ferramenta disponível para rever versões antigas de código, reverter mudanças, explorar novas ideias com segurança ou mesmo ter o código como um backup... não tem preço. O TDD anda de mãos dadas com o controle de versões, portanto quero ter certeza de que vou explicar como esse

sistema se encaixa no fluxo de trabalho.

Então, vamos ao nosso primeiro commit! Apesar de já estarmos um pouco atrasados – que vergonha! Vamos usar o *Git* como nosso VCS, pois ele é o melhor.

Começaremos passando *functional_tests.py* para a pasta *superlists* e executando `git init` para iniciar o repositório:

```
$ ls
superlists functional_tests.py geckodriver.log
$ mv functional_tests.py superlists/
$ cd superlists
$ git init .
Initialised empty Git repository in /.../superlists/.git/
```

Nosso diretório de trabalho a partir de agora será a pasta *superlists* de nível mais alto

A partir de agora, a pasta *superlists* de nível mais alto será o nosso diretório de trabalho.

(Por questões de simplicidade, em minhas listagens de comandos, sempre a mostrarei como `/.../superlists/`, embora é provável que ela vá ser, na verdade, algo como `/home/kind-reader-username/my-python-projects/superlists/`.)

Sempre que eu mostrar um comando a ser digitado, vou supor que estaremos nesse diretório. De modo semelhante, se eu mencionar um path para um arquivo, esse será relativo ao diretório de nível mais alto. Portanto, *superlists/settings.py* quer dizer o *settings.py* que está na pasta *superlists* no segundo nível.

Claro como lama? Se estiver em dúvida, procure *manage.py*; você deverá estar no mesmo diretório em que está *manage.py*.

Vamos ver agora para quais arquivos queremos fazer commit:

```
$ ls
db.sqlite3 manage.py superlists functional_tests.py
```

db.sqlite3 é um arquivo de banco de dados. Não queremos que ele esteja sujeito ao controle de versões. Anteriormente também vimos *geckodriver.log*, um arquivo de log do Selenium, para o qual tampouco queremos controlar as mudanças. Adicionaremos ambos em um arquivo especial chamado *.gitignore*, que diz ao Git o que deve ser ignorado:

```
$ echo "db.sqlite3" >> .gitignore
$ echo "geckodriver.log" >> .gitignore
```

A seguir podemos adicionar o restante do conteúdo da pasta atual, ".":

```
$ git add .
$ git status
On branch master
```

Initial commit

Changes to be committed:
(use "git rm --cached <file>..." to unstage)

```
new file: .gitignore
new file: functional_tests.py
new file: manage.py
new file: superlists/__init__.py
new file: superlists/__pycache__/__init__.cpython-36.pyc
new file: superlists/__pycache__/settings.cpython-36.pyc
new file: superlists/__pycache__/urls.cpython-36.pyc
new file: superlists/__pycache__/wsgi.cpython-36.pyc
new file: superlists/settings.py
new file: superlists/urls.py
new file: superlists/wsgi.py
```

Droga! Acabamos ficando com uma porção de arquivos *.pyc*; não tem sentido fazer commit deles. Vamos removê-los do Git e acrescentá-los ao *.gitignore* também:

```
$ git rm -r --cached superlists/__pycache__
rm 'superlists/__pycache__/__init__.cpython-36.pyc'
rm 'superlists/__pycache__/settings.cpython-36.pyc'
rm 'superlists/__pycache__/urls.cpython-36.pyc'
```

```
rm 'superlists/__pycache__/_wsgi.cpython-36.pyc'  
$ echo "__pycache__" >> .gitignore  
$ echo "*.pyc" >> .gitignore
```

Vamos ver agora qual é a nossa situação... (Você verá que uso bastante o comando `git status` – uso tanto que, com frequência, crio um alias `git st` para ele... mas não vou lhe dizer como fazer isso; deixarei que você mesmo descubra os segredos dos aliases de Git!):

```
$ git status
```

```
On branch master
```

```
Initial commit
```

```
Changes to be committed:
```

```
(use "git rm --cached <file>..." to unstage)
```

```
new file: .gitignore  
new file: functional_tests.py  
new file: manage.py  
new file: superlists/__init__.py  
new file: superlists/settings.py  
new file: superlists/urls.py  
new file: superlists/wsgi.py
```

```
Changes not staged for commit:
```

```
(use "git add <file>..." to update what will be committed)
```

```
(use "git checkout -- <file>..." to discard changes in working directory)
```

```
modified: .gitignore
```

Parece bom – estamos prontos para fazer o nosso primeiro commit!

```
$ git add .gitignore
```

```
$ git commit
```

Quando digitar `git commit`, uma janela de editor será apresentada a você para escrever aí a sua mensagem de commit. A minha se parece com aquela exibida na Figura 1.3.¹



Se você quiser realmente entrar de cabeça no Git, essa também é a hora de saber como enviar seu trabalho para um serviço de hospedagem de VCS em nuvem, como o GitHub ou o Bitbucket. Eles serão convenientes se você achar que vai seguir este livro usando microcomputadores diferentes. Deixarei a seu encargo descobrir como eles funcionam; a documentação deles é excelente. Como alternativa, você poderá esperar até o Capítulo 9, quando usaremos um desses serviços para uma implantação.

```
COMMIT_EDITMSG + (/workspace/superlists/.git) - VIM
File Edit View Search Terminal Help
1 First commit: First FT and basic Django config
2 # Please enter the commit message for your changes. Lines starting
3 # with '#' will be ignored, and an empty message aborts the commit.
4 # On branch master
5 #
6 # Initial commit
7 #
8 # Changes to be committed:
9 #   (use "git rm --cached <file>..." to unstage)
10 #
11 #       new file:   .gitignore
12 #       new file:   functional_tests.py
13 #       new file:   manage.py
14 #       new file:   superlists/__init__.py
15 #       new file:   superlists/settings.py
16 #       new file:   superlists/urls.py
17 #       new file:   superlists/wsgi.py
18 #
~
~
~
~
.git/COMMIT_EDITMSG [ + ] [ tabs ] gitcommit 103,0x67 46,1/18
```

Figura 1.3 – O primeiro commit no Git.

Isso é o que temos como lição sobre o VCS. Parabéns! Você escreveu um teste funcional usando o Selenium, além de ter conseguido instalar e executar o Django, usando TDD de forma comprovada, com um teste antes e a aprovação do Testing Goat (bode dos testes). Dê a si mesmo um bem merecido tapinha nas costas antes de passar para o Capítulo 2.

¹ O vi apareceu e você não tinha a mínima ideia do que fazer? Ou você viu uma mensagem sobre identidade de conta e `git config --global user.username`? Reveja a seção “Pré-requisitos e suposições”; há algumas instruções rápidas ali.

CAPÍTULO 2

Estendendo nosso teste funcional usando unittest

Vamos adaptar nosso teste, que atualmente verifica a página “it worked” (funcionou) default de Django, e conferir alguns pontos que queremos ver na verdadeira página inicial de nosso site.

É hora de revelar o tipo de aplicação web que estamos construindo: um site com listas de tarefas (to-do lists)! Ao fazer isso, estamos basicamente seguindo a moda: há alguns anos, todos os tutoriais web tinham a ver com a construção de um blog. Em seguida, foi a vez de fóruns e pesquisas; atualmente, são listas de tarefas.

A razão para isso é o fato de uma lista de tarefas ser realmente um bom exemplo. Em seu aspecto básico, a lista é muito simples – é somente uma lista de strings de texto –, portanto é fácil ter uma aplicação de lista “mínima viável” ativa e executando. No entanto, ela pode ser estendida de várias maneiras – com diferentes modelos de persistência, acréscimo de prazos finais, lembretes, compartilhamento com outros usuários e melhorias na UI do lado do cliente. Também não há motivos para que estejamos limitados apenas às listas de “tarefas a fazer”; elas podem ser de qualquer tipo. O ponto importante, porém, é que devem permitir que eu demonstre todos os principais aspectos da programação web e como podemos aplicar aí o TDD.

Usando teste funcional para definir o escopo de uma aplicação mínima viável

Os testes que usam o Selenium nos permitem controlar um verdadeiro navegador web, portanto eles realmente possibilitam ver como a aplicação *funciona* do ponto de vista do usuário. É por isso que são chamados de *testes funcionais* (FT, ou Functional Tests).

Isso significa que um FT pode ser uma espécie de especificação para a sua aplicação. O teste tende a registrar o que chamaríamos de uma *História de Usuário* (User Story), e descreve como o usuário trabalharia com uma funcionalidade em particular e como a aplicação deverá responder a ela.

Terminologia: teste funcional == teste de aceitação == teste fim a fim

O que eu chamo de testes funcionais, algumas pessoas preferem chamar de *testes de aceitação* (acceptance tests) ou *testes fim a fim* (end-to-end tests). O ponto principal é que esses tipos de testes olham para o modo como a aplicação toda funciona, a partir do lado externo. Outro termo usado é *teste caixa-preta* (black box test), pois o teste não sabe nada sobre a parte interna do sistema em teste.

Os FTs devem conter uma história legível por um ser humano, ou seja, que possamos compreender. Nós deixamos o teste explícito usando comentários que acompanham o código de teste. Quando criamos um novo FT, podemos escrever os comentários antes a fim de capturar os pontos principais da História do Usuário. Por ser legível aos seres humanos, podemos até mesmo compartilhá-los com pessoas que não sejam programadores, como uma forma de discutir os requisitos e os recursos de sua aplicação.

O TDD e as metodologias ágeis para desenvolvimento de software geralmente andam juntos, e um dos tópicos discutidos com frequência é a aplicação mínima viável (minimum viable app); qual é o artefato mais simples que podemos construir e que seja utilizável? Vamos começar construindo isso para que possamos testar a profundidade da água o mais rápido possível.

Uma lista de tarefas mínima viável realmente só precisa permitir que o usuário insira alguns itens referentes às tarefas e que se lembre deles no próximo acesso.

Abra *functional_tests.py* e escreva uma história um pouco parecida com esta:

functional_tests.py

```
from selenium import webdriver

browser = webdriver.Firefox()

# Edith ouviu falar de uma nova aplicação online interessante para
# lista de tarefas. Ela decide verificar sua homepage
browser.get('http://localhost:8000')

# Ela percebe que o título da página e o cabeçalho mencionam listas de
# tarefas (to-do)
assert 'To-Do' in browser.title

# Ela é convidada a inserir um item de tarefa imediatamente

# Ela digita "Buy peacock feathers" (Comprar penas de pavão) em uma caixa
# de texto (o hobby de Edith é fazer iscas para pesca com fly)

# Quando ela tecla enter, a página é atualizada, e agora a página lista
# "1: Buy peacock feathers" como um item em uma lista de tarefas

# Ainda continua havendo uma caixa de texto convidando-a a acrescentar outro
# item. Ela insere "Use peacock feathers to make a fly" (Usar penas de pavão
# para fazer um fly – Edith é bem metódica)

# A página é atualizada novamente e agora mostra os dois itens em sua lista

# Edith se pergunta se o site lembrará de sua lista. Então ela nota
# que o site gerou um URL único para ela -- há um pequeno
# texto explicativo para isso.

# Ela acessa esse URL - sua lista de tarefas continua lá.
```


Satisfeita, ela volta a dormir

```
browser.quit()
```

Temos uma palavra para comentários...

Quando comecei a trabalhar no Resolver, eu costumava deixar bons comentários descritivos virtuosamente espalhados por todo o meu código. Meus colegas me disseram: “Harry, temos uma palavra para comentários. Nós os chamamos de mentiras.” Fiquei chocado! No entanto, eu havia aprendido na escola que os comentários eram uma boa prática!

Eles estavam exagerando para causar efeito. Com certeza, há um espaço para comentários que acrescentem contexto e propósito. A questão que eles queriam enfatizar, porém, era o fato de não fazer sentido escrever um comentário que simplesmente repita o que você está fazendo no código:

```
# incrementa wibble de 1  
wibble += 1
```

Não só não faz sentido como também há o perigo de você se esquecer de atualizar os comentários quando atualizar o código e eles acabarem se tornando enganosos. O ideal é se esforçar para deixar seu código bem legível, usar nomes de variáveis e de funções muito bons e estruturar bem o código a ponto de não precisar mais de nenhum comentário para explicar o *que* o código faz. Basta alguns aqui e ali para explicar o *porquê*.

Há outros lugares em que os comentários são muito úteis. Veremos que o Django os utiliza bastante nos arquivos que gera para nós como uma forma de fazer pequenas sugestões úteis sobre a sua API. Além disso, é claro, usamos comentários para explicar a História do Usuário em nossos testes funcionais – ao nos forçar a criar uma história coerente para o teste, garantimos que estaremos sempre testando do ponto de vista do usuário.

Há mais diversão a ser desfrutada nessa área: itens como *Behaviour-Driven Development*, ou Desenvolvimento Orientado a Comportamentos (veja o Apêndice E), e DSLs para testes, mas esses são assuntos para outros livros.

Você perceberá que, além de escrever o teste na forma de comentários, atualizei o assert para que procurasse a palavra “To-Do” no lugar de “Django”. Isso significa que estamos esperando que o teste vá falhar agora. Vamos tentar executá-lo.

Em primeiro lugar, inicie o servidor:

```
$ python manage.py runserver
```

Em seguida, em outro shell, execute os testes:

```
$ python functional_tests.py
```

```
Traceback (most recent call last):
```

```
File "functional_tests.py", line 10, in <module>
```

```
    assert 'To-Do' in browser.title
```

```
AssertionError
```

Isso é o que chamamos de *falha esperada* (expected fail), o que, na verdade, é uma boa notícia – não tão boa quanto a de um teste que passa, mas, pelo menos, a falha é pelo motivo correto; podemos ter certa dose de confiança de que escrevemos o teste corretamente.

Módulo unittest da biblioteca-padrão de Python

Há alguns pequenos aspectos irritantes com os quais provavelmente teremos de lidar. Em primeiro lugar, a mensagem “AssertionError” não ajuda muito – seria interessante se o teste nos informasse o que realmente foi encontrado no título do navegador. Além disso, o teste deixou uma janela do Firefox pendente no desktop, portanto seria bom se ela fosse fechada automaticamente para nós.

Uma opção seria usar o segundo parâmetro da palavra reservada assert – algo como:

```
assert 'To-Do' in browser.title, "Browser title was " + browser.title
```

Poderíamos também usar um try/finally para limpar a janela usada do Firefox. Esses tipos de problema, porém, são muito comuns em testes, e há algumas soluções prontas para nós no módulo unittest da biblioteca-padrão. Vamos usá-las! Em *functional_tests.py*:

functional_tests.py

```
from selenium import webdriver
```

```
import unittest
```

```
class NewVisitorTest(unittest.TestCase): ❶
```

```

def setUp(self): ❸
    self.browser = webdriver.Firefox()

def tearDown(self): ❹
    self.browser.quit()

def test_can_start_a_list_and_retrieve_it_later(self): ❶
    # Edith ouviu falar de uma nova aplicação online interessante
    # para lista de tarefas. Ela decide verificar sua homepage
    self.browser.get('http://localhost:8000')

    # Ela percebe que o título da página e o cabeçalho mencionam
    # listas de tarefas (to-do)
    self.assertIn('To-Do', self.browser.title) ❷
    self.fail('Finish the test!') ❺

    # Ele é convidada a inserir um item de tarefa imediatamente
    [...o restante dos comentários se mantém]

if __name__ == '__main__': ❻
    unittest.main(warnings='ignore') ❼

```

Provavelmente você deve ter percebido alguns pontos nesse código:

- ❶ Os testes estão organizados em classes, que herdam de `unittest.TestCase`.
- ❷ O corpo principal do teste está em um método chamado `test_can_start_a_list_and_retrieve_it_later`. Qualquer método cujo nome começa com `test_` é um método de teste e será executado pelo executor de testes (test runner). Podemos ter mais de um método `test_` por classe. Ter bons nomes descritivos para nossos métodos de teste também é uma boa ideia.
- ❸ `setUp` e `tearDown` são métodos especiais executados antes e depois de cada teste. Eles foram usados para iniciar e finalizar o nosso navegador – observe que são um pouco parecidos com um `try/except` no sentido em que `tearDown` executará mesmo que haja um erro durante o teste propriamente dito.¹ Agora não haverá mais

nenhuma janela do Firefox pendente!

- ④ Usamos `self.assertIn` no lugar de simplesmente `assert` para fazer nossas asserções de teste. O `unittest` disponibiliza muitas funções auxiliares como essa para fazer asserções de teste, como `assertEqual`, `assertTrue`, `assertFalse`, e assim por diante. Mais informações podem ser encontradas na documentação de `unittest` (<http://docs.python.org/3/library/unittest.html>)
- ⑤ `self.fail` simplesmente falha, independentemente do que houver, gerando a mensagem de erro especificada. Ele foi usado como um lembrete para finalizar o teste.
- ⑥ Por fim, temos a cláusula `if name == 'main'` (se você ainda não a viu, é assim que um script Python verifica se está sendo executado a partir da linha de comando, e não está simplesmente sendo importado por outro script). Chamamos `unittest.main()`, que inicia o executor de testes de `unittest`; ele encontrará automaticamente as classes e métodos de testes no arquivo e os executará.
- ⑦ `warnings='ignore'` suprime um `ResourceWarning` supérfluo, que estava sendo gerado quando escrevemos este livro. Talvez ele já tenha desaparecido quando você estiver lendo este texto; sintá-se à vontade se quiser experimentar removê-lo!



Se você já leu a documentação de testes de Django, talvez tenha visto algo chamado `LiveServerTestCase` e está se perguntando se devemos usá-lo agora. Nota máxima para você por ter lido o manual simpático! `LiveServerTestCase` é um pouco complicado demais por enquanto, mas prometo que eu o usarei em um capítulo mais adiante...

Vamos testar!

```
$ python functional_tests.py
```

```
F
```

```
=====
```

```
FAIL: test_can_start_a_list_and_retrieve_it_later (__main__.NewVisitorTest)
```

```
-----  
Traceback (most recent call last):
```

```
File "functional_tests.py", line 18, in
```

```
test_can_start_a_list_and_retrieve_it_later
    self.assertIn('To-Do', self.browser.title)
AssertionError: 'To-Do' not found in 'Welcome to Django'
```

```
-----
Ran 1 test in 1.747s
```

```
FAILED (failures=1)
```

Um pouco melhor, não é mesmo? Nossa janela do Firefox foi limpa e temos um relatório elegantemente formatado informando quantos testes foram executados e quantos falharam; o `assertIn` nos apresentou uma mensagem de erro prestativa contendo informações úteis para depuração. Muito bom!

Commit

Esta é uma boa hora para fazer um commit; é uma mudança apropriadamente autocontida. Expandimos nosso teste funcional de modo a incluir comentários que descrevam a tarefa a que nos propomos fazer, isto é, nossa lista de tarefas mínima viável. Também reescrevemos o teste de modo a usar o módulo Python `unittest` e suas várias funções auxiliares para testes.

Execute um `git status` – isso deve fazer com que você tenha certeza de que o único arquivo que mudou foi *functional_tests.py*. Então execute um `git diff`, que mostrará a diferença entre o último commit e o que está no momento no disco. O comando deverá informar-lhe que *functional_tests.py* mudou substancialmente:

```
$ git diff
diff --git a/functional_tests.py b/functional_tests.py
index d333591..b0f22dc 100644
--- a/functional_tests.py
+++ b/functional_tests.py
@@ -1,6 +1,45 @@
 from selenium import webdriver
+import unittest

-browser = webdriver.Firefox()
```

```
-browser.get('http://localhost:8000')
+class NewVisitorTest(unittest.TestCase):
```

```
-assert 'Django' in browser.title
+ def setUp(self):
+ self.browser = webdriver.Firefox()
+
+ def tearDown(self):
+ self.browser.quit()
[...]
```

Vamos agora executar este comando:

```
$ git commit -a
```

O `-a` quer dizer “adicione automaticamente qualquer mudança nos arquivos monitorados (isto é, qualquer arquivo para o qual tenhamos feito commit antes). Nenhum arquivo novo será adicionado (você deve executar explicitamente um `git add` para isso), mas, geralmente, como ocorre nesse caso, não haverá nenhum arquivo novo, portanto esse é um atalho conveniente.

Quando o editor aparecer, acrescente uma mensagem de commit descritiva, como “Primeiro FT especificado nos comentários, utilizando agora o `unittest`”.

Estamos agora em uma excelente posição para começar a escrever um pouco de código de verdade para a nossa aplicação de listas. Continue lendo!

Conceitos úteis de TDD

História de Usuário (User Story)

É uma descrição de como a aplicação funcionará do ponto de vista do usuário. É usada para estruturar um teste funcional.

Falha esperada (Expected failure)

É quando um teste falha do modo como esperávamos que o fizesse.

1 A não ser que haja uma exceção dentro de setUp; nesse caso, tearDown não executará.

CAPÍTULO 3

Testando uma página inicial simples com testes de unidade

Encerramos o último capítulo com um teste funcional em falha, informando-nos que era desejado que a página inicial de nosso site tivesse “To-Do” em seu título. É hora de começar a trabalhar em nossa aplicação.

Aviso: a situação está prestes a se tornar séria

Os dois primeiros capítulos foram intencionalmente leves e tranquilos. A partir de agora, começaremos a ver um código um pouco mais substancial. Eis uma previsão: em algum ponto, as coisas começarão a dar errado. Você verá resultados diferentes daqueles que direi que você deveria estar vendo. É algo bom, pois será uma genuína experiência de aprendizado para desenvolvimento da personalidade.

Uma possibilidade é eu ter dado alguma explicação ambígua e você ter feito algo diferente do que eu pretendia. Pare e pense no que estamos tentando fazer neste ponto do livro. Qual arquivo estamos editando, o que queremos que o usuário seja capaz de fazer, o que estamos testando e por quê? Talvez você tenha editado a função ou o arquivo errado, ou esteja executando os testes incorretos. Acho que você aprenderá mais sobre TDD com esses momentos em que vai “parar e pensar” do que com todas as vezes em que seguir as instruções e copiar e colar ocorrerem tranquilamente.

Ou pode haver um bug de verdade. No entanto, seja persistente, leia a mensagem de erro com atenção (veja a Seção “Lendo os tracebacks”, um pouco mais adiante no capítulo) e você vai chegar ao fundo da questão. É provável que seja apenas uma vírgula ou uma barra final faltando ou, quem sabe, um `s` ausente em um dos métodos de busca do Selenium. Como Zed

Shaw afirma de modo muito apropriado, porém, esse tipo de depuração também é uma parte absolutamente vital do aprendizado, portanto conviva com isso!

Você sempre poderá me enviar um email (ou tentar o Google Group em <https://groups.google.com/forum/#!forum/obey-the-testing-goat-book>) se realmente não souber o que fazer. Tenha uma feliz depuração!

Nossa primeira aplicação Django e nosso primeiro teste de unidade

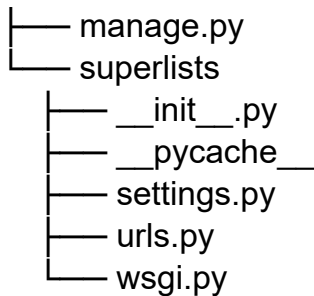
O Django incentiva você a estruturar o seu código em *aplicações* (apps): a teoria diz que um projeto pode ter muitas aplicações, que você pode usar aplicações de terceiros desenvolvidas por outras pessoas e até mesmo reutilizar uma de suas próprias aplicações em um projeto diferente... devo, porém, admitir que eu mesmo jamais consegui fazer isso! Ainda assim, as aplicações são uma boa maneira de manter o seu código organizado.

Vamos iniciar uma aplicação para as nossas listas de tarefas:

```
$ python manage.py startapp lists
```

Esse comando criará uma pasta em *superlists/lists*, ao lado de *superlists/superlists*, e, dentro dela, uma série de arquivos placeholder para itens, como modelos, views (visões) e, de interesse imediato para nós, testes:

```
superlists/
├── db.sqlite3
├── functional_tests.py
├── lists
│   ├── admin.py
│   ├── apps.py
│   ├── __init__.py
│   ├── migrations
│   │   └── __init__.py
│   ├── models.py
│   ├── tests.py
│   └── views.py
```



Testes de unidade e como eles diferem dos testes funcionais

Como ocorre com muitos dos rótulos que atribuímos às coisas, a linha entre testes de unidade e testes funcionais pode se tornar um pouco vaga às vezes. A distinção básica, porém, é que os testes funcionais testam a aplicação a partir de seu lado externo, do ponto de vista do usuário. Os testes de unidade testam a aplicação a partir de dentro, do ponto de vista do programador.

A abordagem com TDD que estou seguindo pretende que nossa aplicação seja coberta pelos dois tipos de testes. Desse modo, nosso fluxo de trabalho terá uma aparência um pouco semelhante a este:

1. Começamos escrevendo um *teste funcional*, descrevendo a nova funcionalidade do ponto de vista do usuário.
2. Depois que tivermos um teste funcional que falhe, começamos a pensar em como escrever um código que faça o teste passar (ou, pelo menos, avançar em relação à falha atual). Usaremos agora um ou mais *testes de unidade* para definir como queremos que o nosso código se comporte – a ideia é que cada linha do código de produção que escrevermos deva ser testada por (no mínimo) um de nossos testes de unidade.
3. Depois que tivermos um teste de unidade com falha, escrevemos a menor quantidade possível de *código de aplicação*, apenas o suficiente para fazer o teste de unidade passar. Podemos iterar entre os passos 2 e 3 algumas vezes, até acharmos que o teste

funcional vai avançar um pouco.

4. Agora, podemos executar novamente nossos testes funcionais e ver se eles passam, ou se avançam um pouco. Isso talvez exija a escrita de novos testes de unidade e mais códigos novos, e assim por diante.

Podemos notar que, durante todo o processo, os testes funcionais direcionam o desenvolvimento que fazemos de um nível alto, enquanto os testes de unidade direcionam o que fazemos em um nível mais baixo.

Isso parece um pouco redundante? Às vezes podemos ter essa impressão, mas os testes funcionais e os testes de unidade têm objetivos realmente muito distintos, e geralmente acabarão sendo bem diferentes.



Os testes funcionais devem ajudar você a construir uma aplicação com as funcionalidades corretas e garantir que você não causará falhas acidentalmente. Os testes de unidade devem ajudá-lo a escrever um código que seja limpo e livre de bugs.

Chega de teoria por enquanto – vamos ver como tudo isso se parece na prática.

Testes de unidade no Django

Vamos ver como podemos escrever um teste de unidade para a nossa view da página inicial. Abra o arquivo novo em *lists/tests.py* e você verá algo como:

`lists/tests.py`

```
from django.test import TestCase
```

```
# Create your tests here.
```

Django sugeriu prestativamente que usemos uma versão especial de TestCase, disponibilizada por ele. É uma versão expandida do unittest.TestCase padrão, com alguns recursos adicionais específicos

de Django, os quais discutiremos nos próximos capítulos.

Já vimos que o ciclo de TDD envolve começar com um teste que falha e então escrever um código para fazê-lo passar. Bem, antes de chegarmos a esse ponto, queremos nos certificar de que o teste de unidade que estamos escrevendo seja definitivamente executado por nosso executor de testes automatizado, qualquer que seja ele. No caso de *functional_tests.py*, estamos executando-o diretamente, mas esse arquivo criado por Django parece um pouco mágico. Assim, somente por garantia, vamos criar um teste bobo que falhe deliberadamente:

lists/tests.py

```
from django.test import TestCase

class SmokeTest(TestCase):

    def test_bad_maths(self):
        self.assertEqual(1 + 1, 3)
```

Vamos agora chamar esse misterioso executor de testes do Django. Como sempre, é um comando de *manage.py*:

```
$ python manage.py test
Creating test database for alias 'default'...
F
=====
=====
FAIL: test_bad_maths (lists.tests.SmokeTest)
-----
Traceback (most recent call last):
  File ".../superlists/lists/tests.py", line 6, in test_bad_maths
    self.assertEqual(1 + 1, 3)
AssertionError: 2 != 3

-----

Ran 1 test in 0.001s

FAILED (failures=1)
System check identified no issues (0 silenced).
```

Destroying test database for alias 'default'...

Excelente. O mecanismo parece estar funcionando. É uma boa hora para um commit:

```
$ git status # deve mostrar a você que lists/ não está sendo controlado
```

```
$ git add lists
```

```
$ git diff --staged # mostrará a diferença da qual você está prestes a fazer commit
```

```
$ git commit -m "Add app for lists, with deliberately failing unit test"
```

Como, sem dúvida, você deve ter adivinhado, a flag `-m` permite passar uma mensagem de commit na linha de comando, portanto não será necessário usar um editor. Cabe a você escolher o modo em que gostaria de usar a linha de comando do Git; mostrarei as opções principais que já vi sendo usadas. A regra principal é: *certifique-se de que você vai sempre revisar o commit que está prestes a fazer antes de executá-lo.*

MVC de Django, URLs e funções de view

Django está estruturado com base em um padrão clássico de *MVC* (Model-View-Controller, ou Modelo-Visão-Controlador). Bem, *de modo geral*. Definitivamente, ele tem modelos, mas suas views assemelham-se mais a um controlador, e são os templates que, na verdade, compõem a parte da visão, mas a ideia geral está presente. Se estiver interessado, poderá ver os detalhes da discussão nas FAQs de Django (<https://docs.djangoproject.com/en/1.11/faq/general/>).

Independentemente de tudo mais, como em qualquer servidor web, a principal tarefa de Django é decidir o que fazer quando um usuário pedir um URL específico em nosso site. O fluxo de trabalho de Django é semelhante a este:

1. Uma *requisição* HTTP chega para um *URL* em particular.
2. O Django utiliza algumas regras para decidir qual função de *view* deve lidar com a requisição (chamamos isso de *resolver* o URL).
3. A função de view processa a requisição e devolve uma *resposta*

HTTP.

Portanto, queremos testar dois pontos:

- Podemos resolver o URL da raiz do site ("/") para uma determinada função de view que criamos?
- Podemos fazer essa função de view devolver um pouco de HTML que fará o teste funcional passar?

Vamos começar pelo primeiro ponto. Abra *lists/tests.py* e modifique o nosso teste bobo para algo como:

lists/tests.py

```
from django.urls import resolve
from django.test import TestCase
from lists.views import home_page ❷

class HomePageTest(TestCase):

    def test_root_url_resolves_to_home_page_view(self):
        found = resolve('/') ❶
        self.assertEqual(found.func, home_page) ❶
```

O que está acontecendo aqui?

- ❶ `resolve` é a função que Django utiliza internamente para resolver URLs e descobrir para qual função de view eles devem ser mapeados. Estamos verificando se `resolve`, quando é chamado com "/", que é a raiz do site, encontra uma função chamada `home_page`.
- ❷ Que função é essa? É a função de view que escreveremos a seguir, a qual, na verdade, devolverá o HTML que queremos. Com base no import, podemos ver que estamos planejando armazená-la em *lists/views.py*.

Então o que você acha que acontecerá quando executarmos os testes?

```
$ python manage.py test
ImportError: cannot import name 'home_page'
```

É um erro bastante previsível e nem um pouco interessante:

tentamos importar algo que ainda nem escrevemos. No entanto, continua sendo uma boa notícia – para o TDD, uma exceção prevista conta como uma falha esperada. Como temos tanto um teste funcional quanto um teste de unidade em falha, temos a total bênção do Testing Goat para escrever um código.

Finalmente! Vamos realmente escrever um pouco de código de aplicação!

É empolgante, não é mesmo? Considere-se avisado: o TDD implica que longos períodos de expectativas são apaziguados apenas gradualmente, em pequenos incrementos. Em particular, por estarmos aprendendo e apenas no começo, só nos permitiremos modificar (ou acrescentar) uma linha de código por vez – e, a cada vez, faremos apenas o mínimo de alterações necessário para tratar a falha atual no teste.

Estou sendo deliberadamente radical nesse caso, mas qual é a nossa falha de teste no momento? Não conseguimos importar `home_page` de `lists.views`? Tudo bem, vamos corrigir isso – e somente isso. Em `lists/views.py`:

`lists/tests.py`

```
from django.shortcuts import render
```

```
# Create your views here.
```

```
home_page = None
```

“Você deve estar brincando!” – posso ouvir você dizendo.

Posso ouvi-lo porque é o que eu costumava dizer (enfaticamente) quando meus colegas me mostravam o TDD no começo. Bem, continue comigo e discutiremos em breve se estamos ou não levando tudo isso um pouco longe demais. Por enquanto, porém, deixe-se levar, mesmo que com certa dose de exasperação, e vejamos se nossos testes podem nos ajudar a escrever o código correto, com um pequeno passo de cada vez.

Vamos executar os testes novamente:

```
$ python manage.py test
```

```
Creating test database for alias 'default'...
```

```
E
```

```
=====
```

```
ERROR: test_root_url_resolves_to_home_page_view  
(lists.tests.HomePageTest)
```

```
-----  
Traceback (most recent call last):
```

```
File ".../superlists/lists/tests.py", line 8, in  
test_root_url_resolves_to_home_page_view
```

```
    found = resolve('/')
```

```
File ".../django/urls/base.py", line 27, in resolve
```

```
    return get_resolver(urlconf).resolve(path)
```

```
File ".../django/urls/resolvers.py", line 392, in resolve
```

```
    raise Resolver404({'tried': tried, 'path': new_path})
```

```
django.urls.exceptions.Resolver404: {'tried': [[<RegexURLResolver  
<RegexURLPattern list> (admin:admin) ^admin/>]], 'path': ''}
```

```
-----  
Ran 1 test in 0.002s
```

```
FAILED (errors=1)
```


```
System check identified no issues (0 silenced).
```

```
Destroying test database for alias 'default'...
```

Lendo os tracebacks

Vamos reservar um tempo discutindo como ler os tracebacks, pois é algo que temos que fazer bastante em TDD. Logo você aprenderá a passar os olhos por eles e captar as pistas relevantes:

```
=====
```

```
ERROR: test_root_url_resolves_to_home_page_view  
(lists.tests.HomePageTest) 
```

```
-----  
Traceback (most recent call last):
```

```
File ".../superlists/lists/tests.py", line 8, in  
test_root_url_resolves_to_home_page_view
```

```

found = resolve('/') ❸
File ".../django/urls/base.py", line 27, in resolve
    return get_resolver(urlconf).resolve(path)
File ".../django/urls/resolvers.py", line 392, in resolve
    raise Resolver404({'tried': tried, 'path': new_path})
django.urls.exceptions.Resolver404: {'tried': [[<RegexURLResolver ❶
<RegexURLPattern list> (admin:admin) ^admin/>]], 'path': ''} ❶
-----
[...]
```

- ❶ O primeiro lugar que você observará geralmente é o *próprio erro*. Às vezes, isso é tudo que você precisará ver, e permitirá identificar prontamente o problema. Em outras ocasiões, porém, como nesse caso, o problema em si não será tão evidente.
- ❷ O próximo ponto a ser verificado com atenção é: *qual testa está falhando?* Definitivamente é o teste que esperávamos, isto é, aquele que acabamos de escrever? Nesse caso, a resposta é sim.
- ❸ Em seguida, vamos procurar o local no *nosso código de teste* que disparou a falha. Trabalhamos de cima para baixo no traceback, procurando o nome do arquivo de teste a fim de verificar qual função de teste e em qual linha de código a falha teve origem. Nesse caso, é na linha em que chamamos a função `resolve` para o URL `"/`.

Geralmente há um quarto passo, em que olhamos na sequência em busca de qualquer *código de nossa própria aplicação* que esteja envolvido no problema. Em nosso caso, tudo é código Django, mas veremos muitos exemplos desse quarto passo mais adiante no livro.

Reunindo tudo, segundo a nossa interpretação do traceback, ele nos informa que, quando tentamos resolver `"/`, o Django gerou um erro 404 – em outras palavras, o Django não pôde encontrar um URL mapeado para `"/`. Vamos resolver isso.

urls.py

Nossos testes nos informam que precisamos de um mapeamento de URL. O Django utiliza um arquivo chamado *urls.py* para mapear

URLs a funções de view. Há um *urls.py* principal para todo o site na pasta *superlists/superlists*. Vamos analisá-lo:

Exemplo 5 – *superlists/urls.py*

```
"""superlists URL Configuration
```

```
The `urlpatterns` list routes URLs to views. For more information please see:  
    https://docs.djangoproject.com/en/1.11/topics/http/urls/
```

```
Examples:
```

```
Function views
```

1. Add an import: `from my_app import views`
2. Add a URL to `urlpatterns`: `url(r'^$', views.home, name='home')`

```
Class-based views
```

1. Add an import: `from other_app.views import Home`
2. Add a URL to `urlpatterns`: `url(r'^$', Home.as_view(), name='home')`

```
Including another URLconf
```

1. Import the `include()` function: `from django.conf.urls import url, include`
2. Add a URL to `urlpatterns`: `url(r'^blog/', include('blog.urls'))`

```
"""
```

```
from django.conf.urls import url  
from django.contrib import admin
```

```
urlpatterns = [  
    url(r'^admin/', admin.site.urls),  
]
```

Como sempre, há muitos comentários úteis e sugestões padrões do Django.

Uma entrada de `url` começa com uma expressão regular que define a quais URLs ela se aplica, e prossegue informando para onde essas requisições devem ser enviadas – pode ser para uma função de `view` que você importar ou, quem sabe, para outro arquivo *urls.py* em outro lugar.

A primeira entrada de exemplo contém a expressão regular `^$`, que representa uma string vazia – poderia ser o mesmo que a raiz de nosso site, que estávamos testando com `/`? Vamos descobrir – o que acontecerá se a incluirmos?



Se você nunca deparou antes com expressões regulares, poderá simplesmente acreditar em minha palavra sobre elas por enquanto – contudo faça uma anotação mental para conhecê-las.

Vamos nos livrar também do URL admin, pois não usaremos o site de administração do Django por enquanto:

Exemplo 6 – superlists/urls.py

```
from django.conf.urls import url
from lists import views

urlpatterns = [
    url(r'^$', views.home_page, name='home'),
]
```

Execute os testes de unidade novamente usando `python manage.py test`:

```
[...]
TypeError: view must be a callable or a list/tuple in the case of include().
```

Tivemos progressos! Não estamos mais vendo um 404.

O traceback é confuso, mas a mensagem no final nos informa o que está acontecendo: os testes de unidade realmente fizeram a ligação entre o URL “/” e `home_page = None` em `lists/views.py`, e está reclamando agora que a view `home_page` não pode ser chamada. Isso nos dá uma justificativa para alterá-la de `None` para uma função de verdade. Toda mudança de código é motivada pelos testes!

Vamos voltar para `lists/views.py`:

Exemplo 7 – lists/tests.py

```
from django.shortcuts import render

# Create your views here.
def home_page():
    pass
```

E agora?

```
$ python manage.py test
```

Creating test database for alias 'default'...

.

Ran 1 test in 0.003s

OK

System check identified no issues (0 silenced).

Destroying test database for alias 'default'...

Viva! Nosso primeiríssimo teste de unidade passou! É um momento tão marcante que acho que vale a pena fazer um commit:

```
$ git diff # deve mostrar mudanças em urls.py, tests.py e views.py
```

```
$ git commit -am "First unit test and url mapping, dummy view"
```

Essa foi a última variação que mostrarei em git commit – as flags `a` e `m` juntas; essa variação adiciona todas as mudanças nos arquivos controlados e utiliza a mensagem de commit da linha de comando.



git commit -am é a fórmula mais rápida, mas também é a que dá o mínimo de feedback a você sobre o commit sendo feito, portanto não se esqueça de executar um git status e um git diff antes para que as mudanças prestes a ser incluídas estejam claras.

Fazendo testes de unidade em uma view

Vamos prosseguir escrevendo um teste para a nossa view de modo que ela seja mais do que uma função que não faz nada, mas uma que devolva uma resposta de verdade, contendo HTML, ao navegador. Abra `lists/tests.py` e acrescente um novo *método de teste*. Explicarei cada uma de suas partes:

Exemplo 8 – lists/tests.py

```
from django.urls import resolve
from django.test import TestCase
from django.http import HttpRequest
```

```
from lists.views import home_page
```

```

class HomePageTest(TestCase):

    def test_root_url_resolves_to_home_page_view(self):
        found = resolve('/')
        self.assertEqual(found.func, home_page)

    def test_home_page_returns_correct_html(self):
        request = HttpRequest() ❶
        response = home_page(request) ❷
        html = response.content.decode('utf8') ❸
        self.assertTrue(html.startswith('<html>')) ❹
        self.assertIn('<title>To-Do lists</title>', html) ❺
        self.assertTrue(html.endswith('</html>')) ❻

```

O que está acontecendo nesse novo teste?

- ❶ Criamos um objeto `HttpRequest`, que é o que o Django verá quando o navegador de um usuário requisitar uma página.
- ❷ Ele é passado para a nossa view `home_page`, que nos dará uma resposta. Você não ficará surpreso ao saber que esse objeto é uma instância de uma classe chamada `HttpResponse`.
- ❸ Em seguida extraímos `.content` da resposta. Esses são os bytes brutos, isto é, os uns e zeros que seriam enviados para o navegador do usuário. Chamamos `.decode()` para convertê-los na string HTML enviada ao usuário.
- ❹ Queremos que ela comece com uma tag `<html>` que será fechada no final.
- ❺ Além disso, queremos uma tag `<title>` em algum lugar no meio, contendo as palavras “To-Do lists” – pois é isso que especificamos em nosso teste funcional.

Mais uma vez, o teste de unidade é direcionado pelo teste funcional, mas também é muito mais próximo do código propriamente dito – no momento, estamos pensando como programadores.

Vamos executar os testes de unidade agora e ver como nos saímos:

```

TypeError: home_page() takes 0 positional arguments but 1 was given

```

O ciclo de testes de unidade/código

Podemos começar a nos adequar ao *ciclo de testes de unidade/código* do TDD agora:

1. No terminal, execute os testes de unidade e observe como eles falham.
2. No editor, faça uma mudança mínima de código para tratar a falha de teste atual.

E repita!

Quanto mais nervosos estivermos para deixar o nosso código correto, menor e mais minimalista será cada mudança de código que fizermos – a ideia é estar absolutamente certo de que cada porção de código seja justificada por um teste.

Pode parecer trabalhoso, e, à primeira vista, será. No entanto, depois que pegar o jeito, você se verá programando rapidamente, mesmo dando passos microscópicos – é assim que escrevemos todo o nosso código de produção no trabalho.

Vamos ver a rapidez com que conseguimos fazer esse ciclo acontecer:

- **Alteração mínima de código:**

lists/views.py

```
def home_page(request):  
    pass
```

- **Testes:**

```
html = response.content.decode('utf8')
```

```
AttributeError: 'NoneType' object has no attribute 'content'
```

- **Código** – usamos `django.http.HttpResponse`, conforme previsto:

lists/views.py

```
from django.http import HttpResponse
```

```
# Create your views here.  
def home_page(request):  
    return HttpResponseRedirect()
```

- **Testes novamente:**

```
self.assertTrue(html.startswith('<html>'))  
AssertionError: False is not true
```

- **Código novamente:**

lists/views.py

```
def home_page(request):  
    return HttpResponse('<html>')
```

- **Testes:**

AssertionError: '<title>To-Do lists</title>' not found in '<html>'

- **Código:**

lists/views.py

```
def home_page(request):  
    return HttpResponse('<html><title>To-Do lists</title>')
```

- **Testes – quase lá?**


```
self.assertTrue(html.endswith('</html>'))  
AssertionError: False is not true
```

- **Vamos lá, faça um último esforço:**

lists/views.py

```
def home_page(request):  
    return HttpResponse('<html><title>To-Do lists</title></html>')
```

- **Tem certeza?**

```
$ python manage.py test
Creating test database for alias 'default'...
```

```
..
```

```
-----
Ran 2 tests in 0.001s
```

```
OK
System check identified no issues (0 silenced).
Destroying test database for alias 'default'...
```

Sim! Vamos agora executar nossos testes funcionais. Não se esqueça de iniciar o servidor de desenvolvimento novamente, caso ele não esteja mais executando. Parece a emoção final da corrida agora; com certeza, é agora... não é mesmo?

```
$ python functional_tests.py
```

```
F
```

```
=====
=====
```

```
FAIL: test_can_start_a_list_and_retrieve_it_later (__main__.NewVisitorTest)
```

```
-----
Traceback (most recent call last):
```

```
  File "functional_tests.py", line 19, in
test_can_start_a_list_and_retrieve_it_later
    self.fail('Finish the test!')
```

```
AssertionError: Finish the test!
```

```
-----
Ran 1 test in 1.609s
```

```
FAILED (failures=1)
```

Falhou? Como assim? Ah, é apenas o nosso pequeno lembrete? Sim? Sim! Temos uma página web!

Aham. Bem, *eu* achei que seria um final emocionante para o capítulo. Talvez você ainda esteja um pouco perplexo, quem sabe ansioso para ouvir uma justificativa para todos esses testes; não se preocupe, tudo isso será feito, mas espero que tenha sentido pelo menos um pouco da empolgação próximo ao final.

Vamos fazer somente um pequeno commit para nos acalmar e

refletir sobre o que fizemos:

```
$ git diff # deve mostrar o nosso novo teste em tests.py e a view em views.py  
$ git commit -am "Basic view now returns minimal HTML"
```

Foi um capítulo e tanto! Por que não tentamos digitar `git log`, possivelmente usando a flag `--oneline`, para lembrar o que fizemos?

```
$ git log --oneline  
a6e6cc9 Basic view now returns minimal HTML  
450c0f3 First unit test and url mapping, dummy view  
ea2b037 Add app for lists, with deliberately failing unit test  
[...]
```

Nada mal – cobrimos:

- como iniciar uma aplicação Django;
- o executor de teste de unidade do Django;
- a diferença entre testes funcionais e testes de unidade;
- resolução de URL no Django e o *urls.py*;
- funções de view e objetos para requisição e resposta no Django;
- devolução de HTML básico.

Comandos e conceitos úteis

Execução do servidor de desenvolvimento do Django

python manage.py runserver

Execução de testes funcionais

python functional_tests.py

Execução de testes de unidade

python manage.py test

O ciclo de testes de unidade/código

1. Execute os testes de unidade no terminal.
2. Faça uma alteração mínima de código no editor.
3. Repita!

CAPÍTULO 4

O que estamos fazendo com todos esses testes? (E a refatoração)

Agora que vimos o básico do TDD em ação, é hora de fazer uma pausa e discutir por que estamos fazendo isso.

Imagino que vários de vocês, caros leitores, estão contendo certa frustração efervescente – talvez alguns já tenham feito um pouco de testes de unidade antes, enquanto outros, quem sabe, estejam apenas com pressa. Vocês têm se contido com perguntas como:

- Todos esses testes não são um pouco excessivos demais?
- Certamente alguns deles são redundantes, não são? Há duplicação entre os testes funcionais e os testes de unidade.
- Quero dizer, o que você está fazendo ao importar `django.core.urlresolvers` em seus testes de unidade? Isso não seria testar o Django, isto é, testar um código de terceiros? Achei que isso seria inadmissível.
- Esses testes de unidade pareceram excessivamente triviais – testar uma linha de declaração e uma função de uma linha que devolve uma constante! Não seria um desperdício de tempo? Não deveríamos deixar nossos testes para tarefas mais complexas?
- E o que dizer de todas aquelas pequenas modificações durante o ciclo de testes de unidade/código? Não poderíamos, certamente, ter ido direto para o final? Quero dizer, `home_page = None`!? Como assim?

- Você não está me dizendo que *realmente* programa assim na vida real, está?

Ah, pequeno gafanhoto, houve uma época em que eu também já fiz várias perguntas como essas, mas só porque são perguntas extremamente boas. De fato, ainda me faço perguntas como essas o tempo todo. Tudo isso realmente é importante? Não haveria aí um pouco de cargo cult¹?

Programar é como puxar um balde de água de um poço

Em última análise, programar é difícil. Com frequência, somos inteligentes, portanto somos bem-sucedidos. O TDD existe para nos ajudar quando não somos assim tão inteligentes. Kent Beck (que basicamente inventou o TDD) utiliza a metáfora de puxar um balde de água de um poço com uma corda: quando o poço não é muito profundo e o balde não está muito cheio, é fácil. E até mesmo puxar um balde cheio no início é bem fácil. Depois de um tempo, porém, você ficará cansado. O TDD é como ter uma chave-catraca que permite interromper o seu progresso, fazer uma pausa e garantir que não haja retrocessos. Assim, você não precisará ser inteligente o tempo *todo*.



Figura 4.1 – Teste TUDO (fonte da ilustração original: Allie Brosh, *Hyperbole and a Half* (<http://bit.ly/1iXxdYp>)).

Tudo bem, quem sabe, *de modo geral*, você esteja preparado para aceitar que o TDD seja uma boa ideia, mas talvez continue achando que estou exagerando. Testar a menor das implementações e dar

muitos passos ridiculamente pequenos?

O TDD é uma *disciplina*, e isso significa que não é algo que surge naturalmente; como muitas das compensações não são imediatas, mas só aparecem no longo prazo, você precisará se forçar a segui-lo no momento. É isso que a imagem do Testing Goat deveria representar – você deve ser um pouco teimoso a esse respeito.

Sobre o mérito dos testes triviais para funções triviais

No curto prazo, pode parecer um pouco bobo escrever testes para funções e constantes simples.

É perfeitamente possível imaginar que estamos fazendo TDD “na maior parte”, porém seguindo regras mais flexíveis nos pontos em que não fazemos testes de unidade em *absolutamente* tudo. Neste livro, porém, meu objetivo é demonstrar um TDD completo e rigoroso. Como um kata em uma arte marcial, a ideia é aprender os movimentos em um contexto controlado, em que não haja adversidades, para que as técnicas façam parte de sua memória muscular. Parece trivial agora porque começamos com um exemplo bem simples. O problema surge quando sua aplicação se torna complexa – é nesse caso que você realmente precisará de seus testes. E o perigo é que a complexidade tende a se imiscuir gradualmente. Talvez você não perceba que ela esteja surgindo, mas logo se sentirá como uma rã que foi fervida.

Há mais duas observações a serem feitas a favor de testes pequenos e simples para funções simples.

A primeira é que, se os testes forem realmente triviais, não consumirão tanto tempo assim para serem escritos. Então, pare de reclamar e simplesmente já os deixe escritos.

A segunda observação é que é sempre bom ter um placeholder. Ter um teste *presente* para uma função simples significa que haverá uma barreira psicológica muito menor a ser transposta quando a função simples se tornar um pouquinho mais complexa – talvez ela adquira um `if`. Então, algumas semanas depois, ela pode ganhar um `laço for`. Sem que você se dê conta, ela passa a ser uma *factory* (fábrica) de parser de árvore recursiva e polimórfica baseada em metaclasses. Todavia, como havia testes para a função desde o princípio, acrescentar um novo teste a cada passo é muito natural, e a função

será bem testada. A alternativa envolve tentar decidir quando uma função se torna “complicada o suficiente”, o que é extremamente subjetivo, mas, pior ainda, por não haver um placeholder, parecerá que muito mais esforço será exigido, e a cada passo você se sentirá tentado a adiar um pouco mais – em breve, você terá uma sopa de rã!

Em vez de tentar descobrir algumas regras subjetivas improdutivas para decidir quando você deve escrever os testes e quando poderá se livrar deles sem maiores problemas, sugiro que seja disciplinado por enquanto – como ocorre com qualquer disciplina, é preciso investir tempo para conhecer as regras antes de poder violá-las.

Agora vamos voltar ao nosso assunto principal.

Usando o Selenium para testar interações com o usuário

Onde estávamos no final do último capítulo? Vamos executar o teste novamente e descobrir:

```
$ python functional_tests.py
F
=====
=====
FAIL: test_can_start_a_list_and_retrieve_it_later (__main__.NewVisitorTest)
-----
Traceback (most recent call last):
  File "functional_tests.py", line 19, in
test_can_start_a_list_and_retrieve_it_later
    self.fail('Finish the test!')
AssertionError: Finish the test!

-----

Ran 1 test in 1.609s

FAILED (failures=1)
```

Você fez o teste e obteve um erro que informava *Problem loading page* (Problema na carga da página) ou *Unable to connect* (Incapaz de se conectar)? Eu também. Isso ocorreu porque esquecemos de

iniciar o servidor de desenvolvimento antes usando `manage.py runserver`. Faça isso e obterá a mensagem de falha que estamos esperando.



Um dos ótimos aspectos sobre o TDD é que jamais precisamos nos preocupar com nos esquecermos do que vem a seguir – basta executar os testes novamente e eles informarão no que precisamos trabalhar.

“Finish the test” (Termine o teste), diz o teste, portanto vamos fazer exatamente isso! Abra `functional_tests.py`, e vamos estender o nosso FT:

`functional_tests.py`

```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
import time
import unittest

class NewVisitorTest(unittest.TestCase):

    def setUp(self):
        self.browser = webdriver.Firefox()

    def tearDown(self):
        self.browser.quit()

    def test_can_start_a_list_and_retrieve_it_later(self):
        # Edith ouviu falar de uma nova aplicação online interessante para
        # lista de tarefas. Ela decide verificar sua homepage
        self.browser.get('http://localhost:8000')

        # Ela percebe que o título da página e o cabeçalho mencionam listas
        # de tarefas (to-do)
        self.assertIn('To-Do', self.browser.title)
        header_text = self.browser.find_element_by_tag_name('h1').text ❶
        self.assertIn('To-Do', header_text)

        # Ela é convidada a inserir um item de tarefa imediatamente
        inputbox = self.browser.find_element_by_id('id_new_item') ❶
```

```

self.assertEqual(
    inputbox.get_attribute('placeholder'),
    'Enter a to-do item'
)

# Ela digita "Buy peacock feathers" (Comprar penas de pavão) em uma
# caixa de texto (o hobby de Edith é fazer iscas para pesca com fly)
inputbox.send_keys('Buy peacock feathers') ❷

# Quando ela tecla enter, a página é atualizada, e agora a página lista
# "1: Buy peacock feathers" como um item em uma lista de tarefas
inputbox.send_keys(Keys.ENTER) ❸
time.sleep(1) ❹

table = self.browser.find_element_by_id('id_list_table')
rows = table.find_elements_by_tag_name('tr') ❶
self.assertTrue(
    any(row.text == '1: Buy peacock feathers' for row in rows)
)

# Ainda continua havendo uma caixa de texto convidando-a a acrescentar
# outro item. Ela insere "Use peacock feathers to make a fly"
# (Usar penas de pavão para fazer um fly – Edith é bem metódica)
self.fail('Finish the test!')

# A página é atualizada novamente e agora mostra os dois itens em sua
lista
[...]
```

- ❶ Estamos usando vários dos métodos disponibilizados pelo Selenium para analisar páginas web: `find_element_by_tag_name`, `find_element_by_id` e `find_elements_by_tag_name` (observe o `s` extra, que significa que vários elementos serão devolvidos, e não apenas um).
- ❷ Também usamos `send_keys`, que é o modo de o Selenium digitar em elementos de entrada.
- ❸ A classe `Keys` (não se esqueça de importá-la) nos permite enviar teclas especiais, como `Enter`.²
- ❹ Quando teclamos `Enter`, a página é atualizada. `time.sleep` está

presente para garantir que o navegador terminou a carga antes de fazermos qualquer asserção sobre a nova página. Isso é chamado de “espera explícita” (essa é uma espera bem simples; faremos melhorias nela no Capítulo 6).



Preste atenção na diferença entre as funções `find_element_...` e `find_elements_...` do Selenium. Uma devolve um elemento e lança uma exceção se não puder encontrar o elemento, enquanto a outra devolve uma lista, que poderá estar vazia.

Além disso, observe aquela função `any`. É um recurso embutido (built-in) pouco conhecido de Python. Nem preciso explicá-lo, não é mesmo? Python é mesmo uma maravilha.

Todavia, se você é um de meus leitores que não conhece Python, o que há em `any` é uma *expressão geradora*, que é como uma *list comprehension* (compreensão de lista), porém mais impressionante. Você deveria ler sobre esse assunto. Se pesquisar no Google, encontrará o próprio Guido explicando-o muito bem (<http://bit.ly/1iXxD18>). Volte e me diga se não é uma verdadeira maravilha!

Vamos ver como prosseguimos:

```
$ python functional_tests.py
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate
element: h1
```

Decodificando essa informação, o teste diz que não é capaz de encontrar um elemento `<h1>` na página. Vamos ver o que podemos fazer para adicionar isso no HTML de nossa página inicial.

Grandes mudanças em um teste funcional geralmente merecem um commit próprio. Deixei de fazer isso em minha primeira versão e me arrependi depois, quando mudei de ideia e a alteração acabou se misturando com várias outras modificações. Quanto mais atômicos forem os seus commits, melhor será:

```
$ git diff # deve exibir mudanças em functional_tests.py
$ git commit -am "Functional test now checks we can input a to-do item"
```

A regra “Não teste constantes”, e os templates que vêm para nos salvar

Vamos dar uma olhada em nossos testes de unidade em *lists/tests.py*. No momento, estamos procurando strings HTML específicas, mas essa não é uma maneira particularmente eficiente de testar HTML. Em geral, uma das regras para os testes de unidade é *Não teste constantes*, e testar HTML como texto se assemelha muito com testar uma constante.

Em outras palavras, se você tiver um código como:

```
wibble = 3
```

Não faria muito sentido ter um teste assim:

```
from myprogram import wibble
assert wibble == 3
```

Os testes de unidade realmente têm a ver com testar a lógica, o controle de fluxo e a configuração. Fazer asserções sobre qual sequência de caracteres temos exatamente em nossas strings HTML não é um teste que faça isso.

Além do mais, misturar strings brutas em Python não é realmente uma boa maneira de lidar com HTML. Há uma solução muito melhor, que consiste em usar templates. Além do mais, se pudermos manter o HTML separado, em um arquivo cujo nome termine com *.html*, teremos um destaque melhor para a sintaxe! Há muitos frameworks de templates Python por aí, e o Django tem o próprio sistema, que funciona muito bem. Vamos utilizá-lo!

Refatorando de modo a usar um template

O que queremos agora é fazer com que nossa função de view devolva exatamente o mesmo HTML, apenas usando um processo diferente. É o que chamamos de refatorar – quando tentamos

melhorar o código *sem alterar a sua funcionalidade*.

A última informação é realmente importante. Se você tentar adicionar uma nova funcionalidade ao mesmo tempo em que estiver refatorando, é bem provável que acabe enfrentando problemas. Refatorar é, na verdade, uma disciplina completa por si só, e há até mesmo um livro de referência para o assunto: *Refactoring*, de Martin Fowler³ (<http://refactoring.com/>).

A primeira regra é que você não pode refatorar sem testes. Felizmente estamos fazendo TDD, portanto nos encontramos muito à frente nesse jogo. Vamos verificar se nossos testes passam; são eles que nos garantem que nossa refatoração preservará o comportamento:

```
$ python manage.py test
[...]  
OK
```

Ótimo! Começaremos tomando nossa string HTML e colocando-a no próprio arquivo. Crie um diretório chamado *lists/templates* para manter os *templates* e então abra um arquivo em *lists/templates/home.html*, para o qual transferiremos o nosso HTML:⁴

lists/templates/home.html

```
<html>  
  <title>To-Do lists</title>  
</html>
```

Hummm, agora a sintaxe está em destaque... Muito melhor! Vamos então alterar a nossa função de view:

lists/views.py

```
from django.shortcuts import render  
  
def home_page(request):  
    return render(request, 'home.html')
```

Em vez de construir o próprio HttpResponse, estamos usando agora a

função `render` do Django. Ela aceita a requisição como seu primeiro parâmetro (explicaremos os motivos mais adiante) e o nome do template a ser renderizado. Django procurará automaticamente as pastas cujos nomes sejam *templates* em qualquer um dos diretórios de sua aplicação. Então ele construirá um `HttpResponse` para você com base no conteúdo do template.



Os templates são um recurso bem eficaz do Django, e seu ponto forte principal consiste em substituir variáveis Python no texto HTML. Não estamos usando esse recurso ainda, mas o faremos em capítulos mais adiante. É por isso que usamos `render` e (depois) `render_to_string` em vez de, digamos, ler manualmente o arquivo de disco com a função embutida `open`.

Vamos ver se isso funciona:

```
$ python manage.py test
```

```
[...]
```

```
=====
```

```
ERROR: test_home_page_returns_correct_html (lists.tests.HomePageTest) ❷
```

```
-----  
Traceback (most recent call last):
```

```
File "../superlists/lists/tests.py", line 17, in
```

```
test_home_page_returns_correct_html
```

```
    response = home_page(request) ❸
```

```
File "../superlists/lists/views.py", line 5, in home_page
```

```
    return render(request, 'home.html' )❹
```

```
File "/usr/local/lib/python3.6/dist-packages/django/shortcuts.py", line 48, in render
```

```
    return HttpResponse(loader.render_to_string(*args, **kwargs),
```

```
File "/usr/local/lib/python3.6/dist-packages/django/template/loader.py", line 170, in render_to_string
```

```
    t = get_template(template_name, dirs)
```

```
File "/usr/local/lib/python3.6/dist-packages/django/template/loader.py", line 144, in get_template
```

```
    template, origin = find_template(template_name, dirs)
```

```
File "/usr/local/lib/python3.6/dist-packages/django/template/loader.py", line 136, in find_template
```

```
    raise TemplateDoesNotExist(name)
```


django.template.base.TemplateDoesNotExist: home.html ❶

Ran 2 tests in 0.004s

Eis outra chance de analisar um traceback:

- ❶ Começamos pelo erro: não foi possível encontrar o template.
- ❷ Em seguida, verificamos qual teste está falhando: certamente é o nosso teste da view HTML.
- ❸ Então localizamos a linha que causou a falha em nossos testes: é no ponto em que chamamos a função `home_page`.
- ❹ Por fim, procuramos a parte do código de nossa aplicação que causou a falha: é quando tentamos chamar `render`.

Por que Django não consegue encontrar o template? Ele está exatamente no local em que achamos que deveria estar, na pasta *lists/templates*.

O fato é que ainda não registramos *oficialmente* nossa aplicação de listas junto ao Django. Infelizmente só executar o comando `startapp` e ter o que obviamente é uma aplicação em sua pasta de projeto não é suficiente. Você deve dizer ao Django que *realmente* quer isso, e acrescentá-la ao *settings.py* também. Serviço completo. Abra o arquivo e procure uma variável chamada `INSTALLED_APPS`, na qual adicionaremos `lists`:

superlists/settings.py

```
# Application definition
```

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'lists',  
]
```

Podemos ver que já existem muitas aplicações ali, por padrão. Basta acrescentar a nossa, `lists`, no final da lista. Não se esqueça da vírgula no final – talvez ela não seja obrigatória, mas um dia você ficará realmente irritado ao esquecê-la enquanto o Python concatena duas strings em linhas diferentes...

Agora podemos tentar executar os testes novamente:

```
$ python manage.py test
[...]
self.assertTrue(html.endswith('</html>'))
AssertionError: False is not true
```

Droga, ainda não foi dessa vez.



De acordo com o fato de seu editor de texto insistir em acrescentar quebras de linha no final dos arquivos, talvez você nem sequer veja esse erro. Nesse caso, poderá seguramente ignorar a próxima parte e avançar direto para o ponto em que podemos ver a listagem exibir OK.

Contudo, houve progressos! Parece que o nosso template pôde ser encontrado, mas a última das três asserções falhou. Aparentemente há algo errado no final da saída. Tive que executar alguns `print(repr(html))` para depurar, mas o fato é que a mudança para templates introduziu uma quebra de linha (`\n`) adicional no final. Podemos fazer a asserção passar assim:

`lists/tests.py`

```
self.assertTrue(html.strip().endswith('</html>'))
```

É uma pequena trapaça, mas um espaço em branco no final de um arquivo HTML realmente não deveria ser importante para nós. Vamos tentar executar os testes novamente:

```
$ python manage.py test
[...]
OK
```

Nossa refatoração de código agora está completa, e os testes indicam que estamos satisfeitos com o fato de o comportamento ter sido preservado. Podemos agora alterar os testes de modo que eles

não testem mais constantes; em vez disso, eles devem simplesmente verificar se estamos renderizando o template correto.

Django Test Client

Uma maneira de fazer o teste é renderizar manualmente o template por conta própria e, então, comparar com o que a view devolve. O Django tem uma função chamada `render_to_string`, que nos permite fazer isso:

lists/tests.py

```
from django.template.loader import render_to_string
[...]
```

```
def test_home_page_returns_correct_html(self):
    request = HttpRequest()
    response = home_page(request)
    html = response.content.decode('utf8')
    expected_html = render_to_string('home.html')
    self.assertEqual(html, expected_html)
```

Entretanto, essa é uma maneira um pouco complicada de testar se estamos usando o template correto. E toda essa inconveniência com `.decode()` e `.strip()` causa distrações. Como alternativa, o Django disponibiliza uma ferramenta chamada Django Test Client (<https://docs.djangoproject.com/en/1.11/topics/testing/tools/#the-test-client>), que apresenta maneiras embutidas de verificar quais templates estão sendo usados. Eis a sua aparência:

lists/tests.py

```
def test_home_page_returns_correct_html(self):
    response = self.client.get('/') ❶

    html = response.content.decode('utf8') ❷
    self.assertTrue(html.startswith('<html>'))
    self.assertIn('<title>To-Do lists</title>', html)
    self.assertTrue(html.strip().endswith('</html>'))

    self.assertTemplateUsed(response, 'home.html') ❸
```

- ❶ Em vez de criar manualmente um objeto `HttpRequest` e chamar a função de view de forma direta, chamamos `self.client.get`, passando-lhe o URL que queremos testar.
- ❷ Deixaremos os testes antigos por enquanto, somente para garantir que tudo está funcionando do modo como achamos que está.
- ❸ `.assertTemplateUsed` é o método de teste que a classe `TestCase` de Django nos disponibiliza. Ela nos permite verificar qual template foi usado para renderizar uma resposta (veja bem, ela só funcionará para respostas que foram obtidas pelo cliente de teste).

E esse teste continuará passando:

```
Ran 2 tests in 0.016s
```

OK

Somente porque sempre suspeito de um teste que ainda não vi falhar, vamos causar deliberadamente uma falha:

lists/tests.py

```
self.assertTemplateUsed(response, 'wrong.html')
```

Dessa maneira, também saberemos como é a aparência de suas mensagens de erro:

```
AssertionError: False is not true : Template 'wrong.html' was not a template used to render the response. Actual template(s) used: home.html
```

Essa informação é muito útil! Vamos fazer a modificação de volta para a asserção correta. Enquanto realizamos isso, podemos apagar nossas asserções antigas. Também é possível apagar o teste antigo `test_root_url_resolves`, pois isso é implicitamente testado pelo Django Test Client. Combinamos dois testes longos em um só!

lists/tests.py (ch04|010)

```
from django.test import TestCase
```

```
class HomePageTest(TestCase):
```

```
def test_uses_home_template(self):
    response = self.client.get('/')
    self.assertTemplateUsed(response, 'home.html')
```

O ponto principal, porém, é que, em vez de testar constantes, estamos testando a nossa implementação. Ótimo!⁵

Por que simplesmente não usamos o Django Test Client desde o começo?

Você pode estar se perguntando: “Por que simplesmente não usamos o Django Test Client desde o início?”. Na vida real, é isso que eu faria. No entanto, queria lhe mostrar o modo “manual” de fazer essa tarefa por dois motivos. Em primeiro lugar, porque me permitiu apresentar os conceitos um a um e a manter a curva de aprendizado tão suave quanto possível. Em segundo, talvez você nem sempre use Django para construir suas aplicações, e as ferramentas de teste nem sempre podem estar disponíveis – mas é sempre possível chamar funções diretamente e analisar suas respostas!

O Django Test Client também apresenta desvantagens; mais adiante no livro, discutiremos a diferença entre os testes de unidade totalmente isolados e os testes “integrados” em direção aos quais o cliente de testes nos impulsiona. Por enquanto, porém, essa é uma opção bastante pragmática.

Sobre a refatoração

Esse foi um exemplo absolutamente trivial de refatoração. No entanto, como Kent Beck afirma em seu livro *Test-Driven Development: By Example*⁶, “Estou recomendando que você realmente trabalhe dessa maneira? Não. Recomendo que você seja capaz de trabalhar dessa maneira”⁷.

Com efeito, enquanto escrevia este livro, meu primeiro instinto foi mergulhar de cabeça e alterar o teste antes – fazê-lo usar a função `assertTemplateUsed` diretamente, apagar as três asserções supérfluas, deixando apenas uma verificação do conteúdo em relação à renderização esperada e, então, prosseguir e fazer a alteração no código. Observe, porém, como isso teria deixado margem para que

eu provocasse quebras no código: eu poderia ter definido o template de modo a conter *qualquer* string arbitrária, em vez de a string com as tags <html> e <title> corretas.



Ao refatorar, trabalhe no código ou nos testes, mas não em ambos ao mesmo tempo.

Há sempre uma tendência de pular alguns passos e realizar ajustes no comportamento enquanto a refatoração é feita, mas logo teremos alterações em meia dúzia de arquivos diferentes, perderemos totalmente o controle da situação e não haverá mais nada funcionando. Se não quiser acabar como o Refactoring Cat (<http://bit.ly/1iXyRt4> – Figura 4.2), atenha-se aos passos pequenos; mantenha a refatoração e as alterações em funcionalidades totalmente separadas.



Figura 4.2 – Refactoring Cat – não se esqueça de ver o GIF

totalmente animado (source: 4GIFs.com)



Vamos deparar com o “Refactoring Cat” novamente neste livro, como um exemplo do que acontece quando nos deixamos levar e queremos fazer alterações demais de uma só vez. Pense nele como o diabinho do desenho animado que seria a contrapartida do Testing Goat; o diabinho apareceria sobre seu outro ombro e daria conselhos ruins a você...

Fazer um commit após qualquer refatoração é uma boa ideia:

```
$ git status # veja tests.py, views.py, settings.py + nova pasta templates
$ git add . # também adicionará a pasta templates não controlada
$ git diff --staged # revise as alteração para as quais estamos prestes
                  # a fazer commit
$ git commit -m "Refactor home page view to use a template"
```

Um pouco mais sobre a nossa página inicial

Enquanto isso, nosso teste funcional continua falhando. Vamos agora realizar uma mudança de código real para fazê-lo passar. Como o nosso HTML está em um template, podemos nos sentir à vontade para fazer alterações aí, sem a necessidade de escrever testes de unidade extras. Queríamos um `<h1>`:

lists/templates/home.html

```
<html>
  <head>
    <title>To-Do lists</title>
  </head>
  <body>
    <h1>Your To-Do list</h1>
  </body>
</html>
```

Vamos ver se nosso teste funcional gosta um pouco mais disso:

```
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate
element: [id="id_new_item"]
```

Tudo bem...

lists/templates/home.html

```
[...]  
  <h1>Your To-Do list</h1>  
  <input id="id_new_item" />  
</body>  
[...]
```

E agora?

```
AssertionError: " != 'Enter a to-do item'
```

Vamos adicionar o nosso texto como placeholder...

lists/templates/home.html

```
<input id="id_new_item" placeholder="Enter a to-do item" />
```

O que resulta em:

```
selenium.common.exceptions.NoSuchElementException: Message: Unable to  
locate  
element: [id="id_list_table"]
```

Então podemos prosseguir e colocar a tabela na página. Nessa fase, ela estará simplesmente vazia...

lists/templates/home.html

```
<input id="id_new_item" placeholder="Enter a to-do item" />  
<table id="id_list_table">  
</table>  
</body>
```

O que diz agora o FT?

```
File "functional_tests.py", line 43, in  
test_can_start_a_list_and_retrieve_it_later  
    any(row.text == '1: Buy peacock feathers' for row in rows)  
AssertionError: False is not true
```

Um pouco enigmático. Podemos usar o número da linha para rastrear, e o fato é que é aquela função `any` sobre a qual eu me gabei antes – ou, mais exatamente, o `assertTrue`, que não tem uma mensagem de falha muito explícita. Podemos passar uma mensagem de erro personalizada como argumento para a maioria

dos métodos `assertX` em `unittest`:

`functional_tests.py`

```
self.assertTrue(
    any(row.text == '1: Buy peacock feathers' for row in rows),
    "New to-do item did not appear in table"
)
```

Se o FT for executado novamente, você deverá ver a nossa mensagem:

```
AssertionError: False is not true : New to-do item did not appear in table
```

Agora, porém, para fazer esse teste passar, precisaremos realmente processar a submissão do formulário feita pelo usuário. E esse é um assunto para o próximo capítulo.

Por enquanto, vamos fazer um commit:

```
$ git diff
$ git commit -am "Front page HTML now generated from a template"
```

Graças a um pouco de refatoração, temos a nossa view configurada para renderizar um template, paramos de testar constantes e estamos agora em boa posição para começar a processar entradas do usuário.

Revisão: o processo de TDD

Vimos até agora todos os aspectos principais do processo de TDD na prática:

- testes funcionais;
- testes de unidade;
- o ciclo de testes de unidade/código;
- refatoração.

É hora de fazer uma pequena recapitulação e, quem sabe, até mesmo ver alguns fluxogramas. Perdoe-me, mas anos desperdiçados como consultor de gerenciamento me arruinaram. Pelo lado positivo, teremos recursão.

O que é o processo de TDD de modo geral? Veja a Figura 4.3.

Escrevemos um teste. Executamos o teste e vemos que ele falha. Escrevemos um mínimo de código para fazer o teste avançar um pouco. Executamos o teste novamente e repetimos o processo até que ele passe. Então, opcionalmente, podemos refatorar o nosso código usando os nossos testes para garantir que não quebraremos nada.

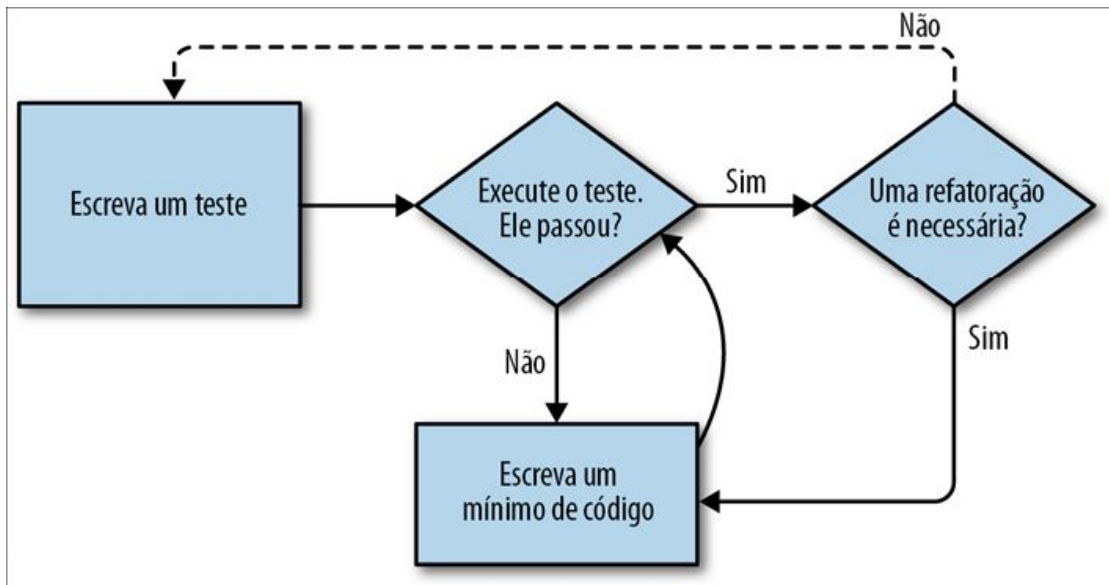


Figura 4.3 – Processo geral de TDD.

Como isso se aplica quando temos testes funcionais e testes de unidade? Bem, podemos pensar no teste funcional como tendo uma visão de alto nível do ciclo, em que “escrever o código” para fazer os testes funcionais passarem envolve, na verdade, o uso de outro ciclo de TDD menor que utiliza testes de unidade. Veja a Figura 4.4.

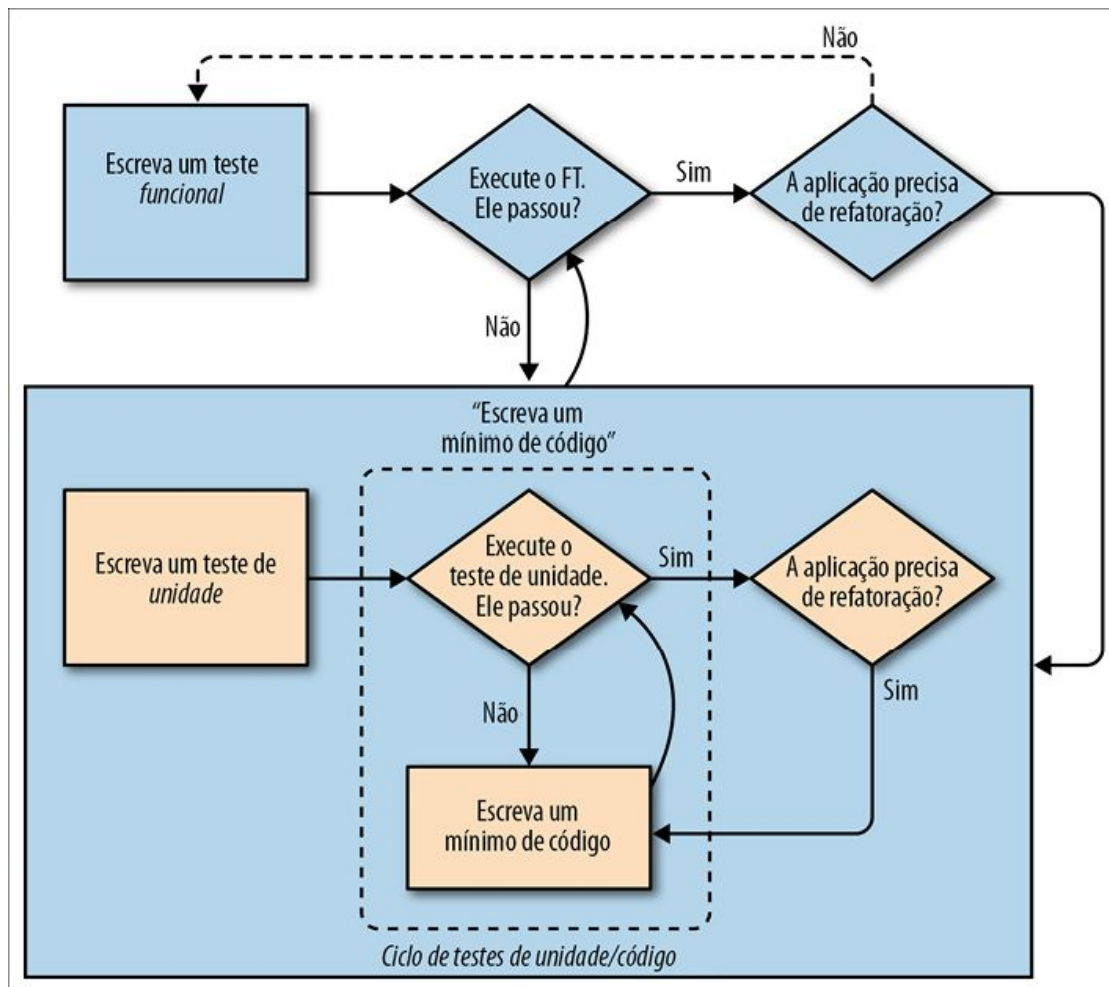


Figura 4.4. – O processo de TDD com testes funcionais e testes de unidade.

Escrevemos um teste funcional e vemos que ele falha. Então o processo de “escrever código” para fazê-lo passar é um miniciclo de TDD por si só: escrevemos um ou mais testes de unidade e passamos pelo ciclo de testes de unidade/código até que os testes de unidade passem. Então, retornamos ao nosso FT a fim de verificar se ele avança um pouco mais e se podemos escrever um pouco mais de nossa aplicação – usando mais testes de unidade, e assim por diante.

E o que dizer da refatoração no contexto dos testes funcionais? Bem, isso significa que usamos o teste funcional para verificar se preservamos o comportamento de nossa aplicação, mas podemos alterar ou adicionar e remover testes de unidade, e usar um ciclo de

testes de unidade para modificar a implementação.

Os testes funcionais, em última instância, são o juiz que decide se sua aplicação funciona ou não. Os testes de unidade são uma ferramenta para ajudá-lo no caminho.

Às vezes, essa forma de olhar a situação é chamada de “TDD de laço duplo”. Uma de minhas eminentes revisoras técnicas, Emily Bache, escreveu uma postagem de blog (<http://bit.ly/1iXzoLR>) sobre o assunto, a qual eu recomendo para ter uma perspectiva diferente.

Exploraremos todas as diferentes partes desse fluxo de trabalho com mais detalhes nos próximos capítulos.

Como “conferir” o seu código, ou siga direto (se tiver que fazer isso)

Todos os exemplos de código que usei no livro estão disponíveis em meu repositório (<https://github.com/hjwp/book-example/>) no GitHub. Portanto, se algum dia você quiser comparar o seu código com o meu, poderá dar uma olhada lá.

Cada capítulo tem o próprio branch, cujo nome é baseado em seu nome conciso. O branch para este capítulo está em https://github.com/hjwp/book-example/tree/chapter_philosophy_and_refactoring, por exemplo. É um snapshot do código, como ele deveria estar no *final* do capítulo.

É possível ver uma lista completa deles no Apêndice J, assim como as instruções sobre como fazer download deles ou usar o Git para comparar seu código com o meu.

-
- ¹ N.T.: Uma programação *cargo cult* é um estilo de programação caracterizada pela inclusão de códigos ou estruturas de programa que não servem a nenhum propósito verdadeiro, feito como se fosse um ritual. É sintomático de um programador que não compreende o bug que está tentando resolver ou não entende a aparente solução. (Baseado em: https://en.wikipedia.org/wiki/Cargo_cult_programming)
 - ² Você também poderia simplesmente usar a string “\n”, mas Keys também permite enviar teclas especiais como Ctrl, portanto pensei em mostrá-la.
 - ³ N.T.: Edição em português: *Refatoração* (Bookman, 2004).
 - ⁴ Algumas pessoas gostam de usar outra subpasta cujo nome depende da

aplicação (isto é, *lists/templates/lists*) e, então, referir-se ao template como *lists/home.html*, o que se chama “namespacing de template”. Achei que isso seria demasiadamente complicado para esse pequeno projeto, mas talvez valha a pena em projetos maiores. Há mais informações no tutorial do Django (<http://bit.ly/1iXxWZL>).

5 Você não consegue seguir em frente porque está se perguntando o que são as informações do tipo *ch04/0xx* ao lado de algumas listagens de código? Elas se referem a commits específicos (https://github.com/hjwp/book-example/commits/chapter_philosophy_and_refactoring) no repositório de exemplos do livro. Têm a ver com os próprios testes de meu livro (<https://github.com/hjwp/Book-TDD-Web-Dev-Python/tree/master/tests>). Você sabe, são os testes para os testes no livro sobre testes. Naturalmente eles têm os próprios testes.

6 N.T.: Edição em português: *TDD Desenvolvimento guiado por testes* (Bookman, 2010).

7 N.T.: Tradução livre de acordo com o original em inglês.

CAPÍTULO 5

Salvando a entrada do usuário: testando o banco de dados

Queremos aceitar a entrada do item de tarefa da usuária e enviá-la ao servidor para que possamos, de algum modo, salvá-la e exibi-la de volta para ela.

Quando comecei a escrever este capítulo, pulei imediatamente para o que eu achava ser o design correto: vários modelos para listas e itens de lista, um punhado de URLs diferentes para acrescentar novas listas e itens, três novas funções de view e aproximadamente meia dúzia de novos testes de unidade para tudo isso. Contudo, parei para pensar. Embora estivesse bem certo de que eu era inteligente o suficiente para lidar com todos esses problemas de uma só vez, o ponto principal do TDD é permitir que você faça uma tarefa de cada vez, quando for necessário fazê-la. Assim decidi deliberadamente assumir uma visão de curto prazo e, em qualquer dado momento, fazer somente o que seria necessário para os testes funcionais avançarem um pouco mais.

É uma demonstração de como o TDD pode oferecer suporte a um estilo iterativo de desenvolvimento – talvez não seja o caminho mais rápido, mas, no final, você chegará lá. Há um bom efeito colateral, que é o fato de ele me permitir apresentar novos conceitos como modelos, tratamento de requisições POST, tags de template de Django, e assim por diante, *individualmente*, em vez de ter que despejar tudo de uma só vez em você.

No entanto, nada disso quer dizer que você *não deva* pensar com antecedência e ser inteligente. No próximo capítulo, usaremos um

pouco mais de design e planejamento prévio, e mostraremos como isso se encaixa no TDD. Por enquanto, porém, vamos prosseguir sem muita preocupação e fazer somente o que os testes nos dizem para fazer.

Preparando o nosso formulário para enviar uma requisição POST

No final do último capítulo, os testes nos informavam que não podíamos salvar a entrada do usuário. Por enquanto, usaremos uma requisição POST padrão de HTML. É um pouco enfadonho, mas também é simples e fácil de enviar – podemos usar todo tipo de HTML5 e JavaScript sexy mais adiante no livro.

Para nosso navegador enviar uma requisição POST, devemos executar duas tarefas:

1. Dar ao elemento `<input>` um atributo `name=`.
2. Encapsulá-lo em uma tag `<form>` com `method="POST"`.

Vamos ajustar o nosso template em *lists/templates/home.html*:

lists/templates/home.html

```
<h1>Your To-Do list</h1>
<form method="POST">
  <input name="item_text" id="id_new_item" placeholder="Enter a to-do item" />
</form>
<table id="id_list_table">
```

Agora a execução de nossos FTs resulta em um erro um pouco enigmático e inesperado:

```
$ python functional_tests.py
[...]
Traceback (most recent call last):
  File "functional_tests.py", line 40, in
test_can_start_a_list_and_retrieve_it_later
    table = self.browser.find_element_by_id('id_list_table')
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
```



```
locate
element: [id="id_list_table"]
```

Quando um teste funcional falha de maneira inesperada, há várias atitudes que podemos tomar para fazer a depuração:

- adicionar instruções `print` para mostrar, por exemplo, qual é o texto da página atual;
- melhorar a *mensagem de erro* para mostrar mais informações sobre o estado atual;
- acessar manualmente o site por conta própria;
- usar `time.sleep` para fazer uma pausa no teste durante a execução.

Veremos todos esses itens ao longo deste livro, mas a opção `time.sleep` é uma que me vejo usando com muita frequência. Vamos experimentar usá-la agora.

Convenientemente, já tivemos um `sleep` logo antes de o erro ocorrer; vamos apenas estendê-lo um pouco:

functional_tests.py

```
# Quando ela tecla enter, a página é atualizada, e agora a página lista
# "1: Buy peacock feathers" como um item em uma lista de tarefas
inputbox.send_keys(Keys.ENTER)
time.sleep(10)
```

```
table = self.browser.find_element_by_id('id_list_table')
```

Dependendo da rapidez com que o Selenium executa em seu microcomputador, talvez você já tenha tido um vislumbre das informações a seguir, mas, quando executamos os testes funcionais novamente, temos tempo para ver o que está acontecendo: você deverá ver uma página com a aparência mostrada na Figura 5.1, com muitas informações de depuração do Django.

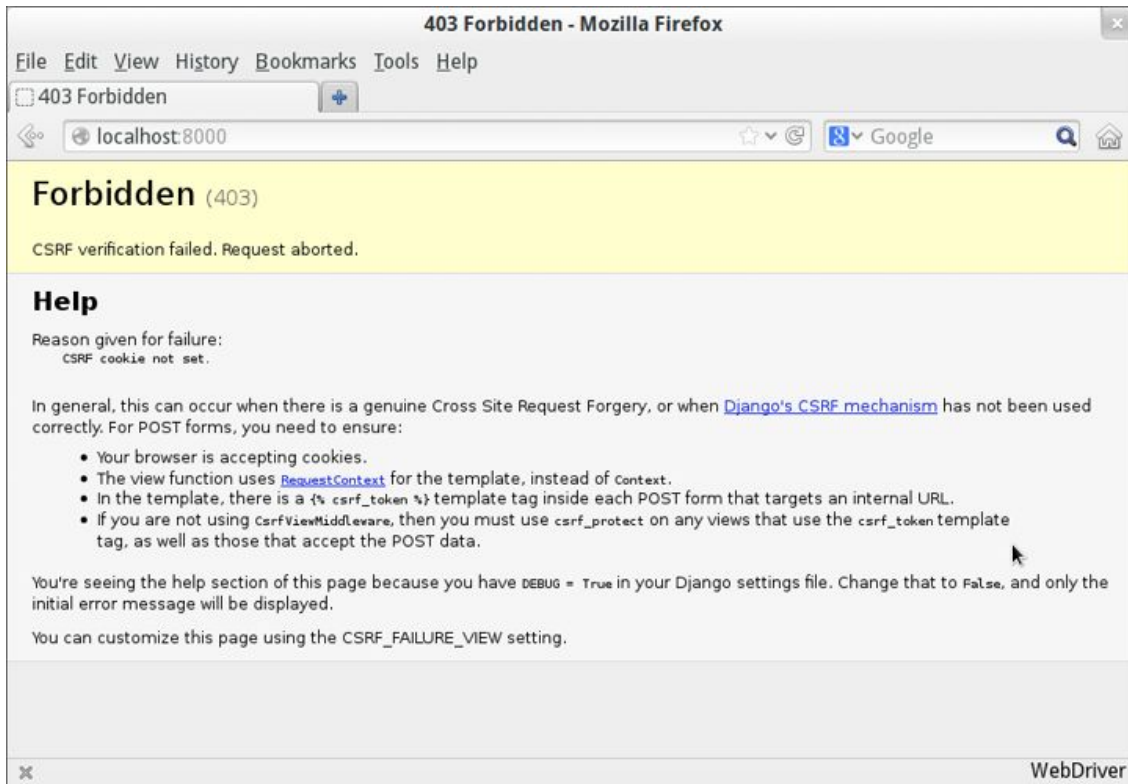


Figura 5.1 – Página de DEBUG do Django mostrando um erro de CSRF.

Segurança: surpreendentemente divertido!

Se você nunca ouviu falar de uma exploração de falhas (exploit) *Cross-Site Request Forgery*, por que não pesquisar sobre ela agora? Como todas as explorações de falhas de segurança, é interessante ler sobre ela, pois essa representa um uso engenhoso de um sistema de formas inesperadas...

Quando voltei para a universidade a fim de me graduar em Ciência da Computação, me inscrevi na disciplina de Segurança por um senso de obrigação: *Oh céus, provavelmente será bem chato e maçante, mas suponho que seja melhor eu fazer essa matéria.* O fato é que foi uma das disciplinas mais fascinantes de todo o curso – absolutamente cheia da alegria de fazer hacking e da mentalidade específica necessária para pensar em como os sistemas podem ser usados de formas não planejadas.

Gostaria de recomendar o livro utilizado em meu curso: *Security Engineering*, de Ross Anderson. É bem leve quanto à criptografia pura, mas está repleto de discussões interessantes sobre tópicos inesperados como arrombamento de cadeados, falsificação de títulos bancários, economia de cartuchos de

impressoras jato de tinta e spoofing (falsificação) em jatos da Força Aérea Sul-Africana com replay attacks (ataques de reprodução de mensagem). É um volume enorme, com quase oito centímetros de espessura, mas garanto que é um livro absolutamente empolgante.

A proteção do Django contra o CSRF envolve colocar um pequeno token gerado automaticamente em cada formulário criado para identificar requisições POST como provenientes do site original. Até agora nosso template havia sido exclusivamente HTML, e, nesse passo, fizemos o primeiro uso da mágica de templates do Django. Para adicionar o token de CSRF, usamos uma *tag de template*, que tem a sintaxe de chaves com caracteres de porcentagem, `{% ... %}` – famosa por ser a combinação das duas teclas mais irritantes do mundo:

lists/templates/home.html

```
<form method="POST">
  <input name="item_text" id="id_new_item" placeholder="Enter a to-do item" />
  {% csrf_token %}
</form>
```

O Django substituirá isso durante a renderização com um `<input type="hidden">` contendo o token de CSRF. Uma nova execução do teste funcional agora nos dará uma falha esperada:

```
AssertionError: False is not true : New to-do item did not appear in table
```

Como o nosso `time.sleep` demorado continua presente, haverá uma pausa no teste na última tela, mostrando que o texto do novo item desaparecerá depois que o formulário for submetido, e a página será atualizada de modo a exibir um formulário vazio novamente. Isso ocorre porque ainda não adaptamos o nosso servidor para lidar com a requisição POST – ele simplesmente a ignorará e exibirá a página inicial regular.

No entanto, podemos pôr de volta o nosso `time.sleep` breve:

functional_tests.py

```
# "1: Buy peacock feathers" como um item em uma lista de tarefas
```

```
inputbox.send_keys(Keys.ENTER)
time.sleep(1)
```

```
table = self.browser.find_element_by_id('id_list_table')
```

Processando uma requisição POST no servidor

Como não especificamos um atributo `action=` no formulário, a submissão está sendo feita para o mesmo URL a partir do qual ele foi renderizado por padrão (isto é, `/`), que é tratado pela nossa função `home_page`. Vamos adaptar a view para que ela seja capaz de lidar com uma requisição POST.

Isso implica um novo teste de unidade para a view `home_page`. Abra `lists/tests.py` e acrescente um novo método em `HomePageTest`:

lists/tests.py (ch05l005)

```
def test_uses_home_template(self):
    response = self.client.get('/')
    self.assertTemplateUsed(response, 'home.html')
```

```
def test_can_save_a_POST_request(self):
    response = self.client.post('/', data={'item_text': 'A new list item'})
    self.assertIn('A new list item', response.content.decode())
```

Para fazer um POST, chamamos `self.client.post`, e, como podemos ver, ele aceita um argumento `data` que contém os dados de formulário que queremos enviar. Então, verificamos se o texto de nossa requisição POST vai parar no HTML renderizado. Isso nos dá a nossa falha esperada:

```
$ python manage.py test
[...]
AssertionError: 'A new list item' not found in '<html>\n <head>\n
<title>To-Do lists</title>\n </head>\n <body>\n <h1>Your To-Do
list</h1>\n <form method="POST">\n <input name="item_text"
[...]
</body>\n</html>\n'
```

Podemos fazer o teste passar adicionando um `if` e definindo um

caminho diferente no código para as requisições POST. Em um estilo típico de TDD, começamos com um valor de retorno propositalmente bobo:

lists/views.py

```
from django.http import HttpResponse
from django.shortcuts import render

def home_page(request):
    if request.method == 'POST':
        return HttpResponse(request.POST['item_text'])
    return render(request, 'home.html')
```

Esse código faz nossos testes de unidade passarem, mas não é realmente o que queremos. O que queremos fazer de fato é adicionar a submissão de POST à tabela no template da página inicial.

Passando variáveis Python para serem renderizadas no template

Já havíamos visto uma pequena amostra da sintaxe de template do Django, e agora é hora de começar a conhecer a verdadeira eficácia dela, que está em passar variáveis de nosso código de view em Python para templates HTML.

Vamos começar observando como a sintaxe do template nos permite incluir um objeto Python em nosso template. A notação é {{ ... }}, que exibe o objeto na forma de uma string:

lists/templates/home.html

```
<body>
  <h1>Your To-Do list</h1>
  <form method="POST">
    <input name="item_text" id="id_new_item" placeholder="Enter a to-do item"
  />
  {% csrf_token %}
</form>
```

```
<table id="id_list_table">
  <tr><td>{{ new_item_text }}</td></tr>
</table>
</body>
```

Vamos adaptar o nosso teste de unidade para que ele verifique se ainda continuamos usando o template:

lists/tests.py

```
def test_can_save_a_POST_request(self):
    response = self.client.post('/', data={'item_text': 'A new list item'})
    self.assertIn('A new list item', response.content.decode())
    self.assertTemplateUsed(response, 'home.html')
```

Esse teste falhará conforme esperado:

```
AssertionError: No templates used to render the response
```

Muito bem, nosso valor de retorno propositalmente bobo não está mais enganando nossos testes, portanto temos permissão para reescrever a nossa view e lhe dizer que passe o parâmetro de POST para o template. A função `render` aceita como terceiro argumento um dicionário que mapeia nomes de variáveis do template aos seus valores:

lists/views.py (ch05l009)

```
def home_page(request):
    return render(request, 'home.html', {
        'new_item_text': request.POST['item_text'],
    })
```

Vamos executar os testes de unidade novamente:

```
ERROR: test_uses_home_template (lists.tests.HomePageTest)
[...]
File ".../superlists/lists/views.py", line 5, in home_page
    'new_item_text': request.POST['item_text'],
[...]
django.utils.datastructures.MultiValueDictKeyError: "'item_text'"
```

Tivemos uma *falha inesperada*.

Caso você se lembre das regras para ler tracebacks, perceberá que,

na verdade, essa é uma falha em um teste *diferente*. Conseguimos fazer com que o teste no qual estávamos trabalhando passasse, porém os testes de unidade sofreram uma consequência inesperada, isto é, houve uma regressão: quebramos o caminho de código em que não há nenhuma requisição POST.

Esse é o ponto principal pelo qual temos testes. Sim, poderíamos ter previsto que isso aconteceria, mas suponha que estivéssemos tendo um dia ruim ou não estivéssemos prestando atenção: nossos testes simplesmente nos impediram de causar falhas acidentalmente em nossa aplicação e, por estarmos usando TDD, identificamos prontamente o problema. Não tivemos que esperar uma equipe de QA nem alternar para um navegador web e clicar manualmente em nosso site, e podemos prosseguir fazendo imediatamente a correção. Eis o modo de realizar isso:

Exemplo 10 – lists/views.py

```
def home_page(request):  
    return render(request, 'home.html', {  
        'new_item_text': request.POST.get('item_text', ''),  
    })
```

Dê uma olhada em `dict.get` (<http://docs.python.org/3/library/stdtypes.html#dict.get>) caso você não tenha certeza do que está acontecendo nesse caso.

Os testes de unidade agora devem passar. Vamos ver o que dizem os testes funcionais:

```
AssertionError: False is not true : New to-do item did not appear in table
```



Se seus testes funcionais mostrarem um erro diferente nesse ponto, ou em qualquer ponto deste capítulo, reclamando sobre uma `StaleElementReferenceException`, talvez seja necessário aumentar a espera explícita de `time.sleep` – experimente usar 2 ou 3 segundos em vez de 1; em seguida, continue a leitura no próximo capítulo para ver uma solução mais robusta.

Humm, não é uma mensagem de erro incrivelmente prestativa.

Vamos usar outra de nossas técnicas de depuração de FT: melhorar a mensagem de erro. Essa, provavelmente, é a técnica mais construtiva, pois essas mensagens de erro melhoradas permanecem no código para ajudar a depurar quaisquer erros futuros:

functional_tests.py (ch05l011)

```
self.assertTrue(
    any(row.text == '1: Buy peacock feathers' for row in rows),
    f"New to-do item did not appear in table. Contents were:\n{table.text}" ❶
)
```

- ❶ Caso você ainda não tenha visto essa sintaxe antes, é a nova sintaxe “f-string” de Python (provavelmente é o recurso novo mais empolgante do Python 3.6). Basta prefixar uma string com f e então você poderá utilizar a sintaxe de chaves para inserir variáveis locais. Há mais informações nas release notes (notas de lançamento de versão) do Python 3.6 (<https://docs.python.org/3/whatsnew/3.6.html#pep-498-formatted-string-literals>).

Isso nos dá uma mensagem de erro mais útil:

```
AssertionError: False is not true : New to-do item did not appear in table.
Contents were:
Buy peacock feathers
```

Sabe o que seria melhor ainda? Deixar essa asserção um pouco menos inteligente. Como você talvez se lembre, eu estava muito satisfeito comigo mesmo por usar a função any, mas um de meus leitores da Versão Preliminar (obrigado, Jason!) sugeriu uma implementação muito mais simples. Podemos substituir todas as quatro linhas de assertTrue por um único assertIn:

functional_tests.py (ch05l012)

```
self.assertIn('1: Buy peacock feathers', [row.text for row in rows])
```

Muito melhor. Fique sempre bastante preocupado se achar que está sendo inteligente, pois o que provavelmente você estará sendo é

excessivamente complicado. E conseguimos obter a mensagem de erro gratuitamente:

```
self.assertIn('1: Buy peacock feathers', [row.text for row in rows])
AssertionError: '1: Buy peacock feathers' not found in ['Buy peacock feathers']
```

Considere-me devidamente punido.



Se, por outro lado, seu FT parecer dizer que a tabela está vazia (“not found in []”, ou seja, não encontrado em []), verifique sua tag `<input>` – ela tem o atributo `name="item_text"` correto? Sem ele, a entrada do usuário não será associada com a chave correta em `request.POST`.

A questão é que o FT quer que enumeremos os itens da lista com um “1:” no início do primeiro item da lista. A maneira mais rápida de conseguir que esse teste passe é fazer uma modificação rápida “trapaceira” no template:

`lists/templates/home.html`

```
<tr><td>1: {{ new_item_text }}</td></tr>
```

Vermelho/Verde/Refatorar e triangulação

O ciclo de testes de unidade/código às vezes é ensinado como *Vermelho*, *Verde*, *Refatorar* (Red, Green, Refactor):

- Comece escrevendo um teste de unidade que falhe (*Vermelho*).
- Escreva o código mais simples possível para fazer o teste passar (*Verde*), *mesmo que isso signifique trapacear*.
- *Refatore* para chegar a um código melhor, que faça mais sentido.

Então, o que fazemos durante a fase Refatorar? O que justifica passar de uma implementação em que “trapaceamos” para outra com a qual estejamos satisfeitos?

Uma metodologia é *eliminar duplicações*: se seu teste utiliza uma constante mágica (como o “1:” na frente de nosso item de lista) e o código de sua aplicação também a usa, isso é considerada uma duplicação, portanto justifica uma refatoração. Remover a constante mágica do código da aplicação geralmente significa que temos que parar de trapacear.

Acho que isso deixa tudo um pouco vago demais, portanto, em geral, gosto de usar uma segunda técnica, chamada de *triangulação*: se seus testes permitirem que você se saia bem escrevendo um código “trapaceiro” com o qual não esteja satisfeito, por exemplo, devolvendo uma constante mágica, *escreva outro teste* que o force a escrever um código melhor. É isso que estamos fazendo quando estendemos o FT a fim de verificar se obtemos um “2:” quando inserimos um *segundo* item na lista.

Chegamos ao `self.fail('Finish the test!')`. Se estendermos o nosso FT para verificar a adição de um segundo item na tabela (somos amigos de copiar e colar), começaremos a perceber que nossa primeira solução provisória realmente é, hum, provisória:

functional_tests.py

```
# Ainda continua havendo uma caixa de texto convidando-a a acrescentar
# outro item. Ela insere "Use peacock feathers to make a fly"
# (Usar penas de pavão para fazer um fly – Edith é bem metódica)
inputbox = self.browser.find_element_by_id('id_new_item')
inputbox.send_keys('Use peacock feathers to make a fly')
inputbox.send_keys(Keys.ENTER)
time.sleep(1)

# A página é atualizada novamente e agora mostra os dois itens em sua lista
table = self.browser.find_element_by_id('id_list_table')
rows = table.find_elements_by_tag_name('tr')
self.assertIn('1: Buy peacock feathers', [row.text for row in rows])
self.assertIn(
    '2: Use peacock feathers to make a fly',
    [row.text for row in rows]
)

# Edith se pergunta se o site lembrará de sua lista. Então ela nota
# que o site gerou um URL único para ela -- há um pequeno
# texto explicativo para isso.
self.fail("Finish the test!")

# Ela acessa esse URL – sua lista de tarefas continua lá.
```

Certamente os testes funcionais devolverão um erro:

```
AssertionError: '1: Buy peacock feathers' not found in ['1: Use peacock feathers to make a fly']
```

Três acertos e refatorar

Antes de prosseguir – temos um *code smell*¹ ruim nesse FT. Temos três blocos de código quase idênticos verificando novos itens na lista que está na tabela. Há um princípio chamado *DRY* (*Don't Repeat Yourself*, ou Não Se Repita), que gostamos de aplicar seguindo o mantra *três acertos e refatorar*. Você pode copiar e colar um código uma vez, e talvez seja prematuro tentar remover a duplicação causada por isso, mas, depois que você tiver três ocorrências, é hora de remover a duplicação.

Começaremos fazendo um commit do que temos até agora. Apesar de sabermos que nosso site tem uma falha importante – é capaz de lidar somente com um item na lista –, ele está muito além do ponto em que estava. Talvez precisemos reescrevê-lo totalmente, ou quem sabe não, mas a regra é: antes de fazer qualquer refatoração, sempre faça um commit:

```
$ git diff
# deve exibir mudanças em functional_tests.py, home.html,
# tests.py e views.py
$ git commit -a
```

Vamos voltar à refatoração de nosso teste funcional: poderíamos usar uma função inline, mas isso causaria um pequeno distúrbio no fluxo do teste. Vamos utilizar um método auxiliar – lembre-se de que somente métodos que comecem com `test_` serão executados como testes, portanto outros métodos podem ser usados para seus propósitos particulares:

functional_tests.py

```
def tearDown(self):
    self.browser.quit()

def check_for_row_in_list_table(self, row_text):
```

```
table = self.browser.find_element_by_id('id_list_table')
rows = table.find_elements_by_tag_name('tr')
self.assertIn(row_text, [row.text for row in rows])
```

```
def test_can_start_a_list_and_retrieve_it_later(self):
    [...]
```

Gosto de colocar os métodos auxiliares próximos ao início da classe, entre `tearDown` e o primeiro teste. Vamos usar isso no FT:

functional_tests.py

```
# Quando ela tecla enter, a página é atualizada, e agora a página lista
# "1: Buy peacock feathers" como um item em uma lista de tarefas
inputbox.send_keys(Keys.ENTER)
time.sleep(1)
self.check_for_row_in_list_table('1: Buy peacock feathers')
```

```
# Ainda continua havendo uma caixa de texto convidando-a a acrescentar
# outro item. Ela insere "Use peacock feathers to make a fly"
# (Usar penas de pavão para fazer um fly – Edith é bem metódica)
inputbox = self.browser.find_element_by_id('id_new_item')
inputbox.send_keys('Use peacock feathers to make a fly')
inputbox.send_keys(Keys.ENTER)
time.sleep(1)
```

```
# A página é atualizada novamente e agora mostra os dois itens em sua lista
self.check_for_row_in_list_table('1: Buy peacock feathers')
self.check_for_row_in_list_table('2: Use peacock feathers to make a fly')
```

```
# Edith se pergunta se o site lembrará de sua lista. Então ela nota
[...]
```

Executamos o FT novamente para verificar se ele ainda se comporta do mesmo modo...

```
AssertionError: '1: Buy peacock feathers' not found in ['1: Use peacock
feathers to make a fly']
```

Muito bem. Agora podemos fazer `commit` da refatoração do FT como uma mudança pequena e atômica por si só:

```
$ git diff # verifica as mudanças em functional_tests.py
```

```
$ git commit -a
```

E de volta ao trabalho. Se, em algum momento, vamos tratar mais de um item na lista, precisaremos de certo tipo de persistência, e os bancos de dados são uma solução robusta nessa área.

ORM do Django e o nosso primeiro modelo

Um *ORM* (*Object-Relational Mapper*, ou Mapeador Objeto-Relacional) é uma camada de abstração para dados armazenados em um banco de dados com tabelas, linhas e colunas. O ORM nos permite trabalhar com bancos de dados usando metáforas conhecidas de orientação a objetos que funcionam bem com o código. As classes são mapeadas para tabelas do banco de dados, os atributos são mapeados para colunas e uma instância individual da classe representa uma linha de dados no banco de dados.

O Django tem um ORM excelente, e escrever um teste de unidade que o utilize é, na verdade, uma ótima maneira de conhecê-lo, pois exercita o código ao especificar como queremos que ele funcione.

Vamos criar uma classe nova em *lists/tests.py*:

lists/tests.py

```
from lists.models import Item
[...]
```

```
class ItemModelTest(TestCase):

    def test_saving_and_retrieving_items(self):
        first_item = Item()
        first_item.text = 'The first (ever) list item'
        first_item.save()

        second_item = Item()
        second_item.text = 'Item the second'
        second_item.save()

        saved_items = Item.objects.all()
        self.assertEqual(saved_items.count(), 2)
```

```
first_saved_item = saved_items[0]
second_saved_item = saved_items[1]
self.assertEqual(first_saved_item.text, 'The first (ever) list item')
self.assertEqual(second_saved_item.text, 'Item the second')
```

Podemos ver que criar um novo registro no banco de dados é uma questão relativamente simples e consiste em criar um objeto, atribuir-lhe alguns atributos e chamar uma função `.save()`. O Django também nos oferece uma API para consultar o banco de dados por meio de um atributo de classe `.objects`, e usamos a consulta mais simples possível, isto é, `.all()`, que devolve todos os registros dessa tabela. O resultado é devolvido na forma de um objeto do tipo lista chamado `QuerySet` a partir do qual podemos extrair objetos individuais, além de chamar outras funções como `.count()`. Então conferimos os objetos conforme foram salvos no banco de dados a fim de verificar se as informações corretas foram salvas.

O ORM do Django tem vários outros recursos úteis e intuitivos; agora pode ser um bom momento para passar os olhos pelo tutorial do Django (<https://docs.djangoproject.com/en/1.11/intro/tutorial01/>), que tem uma excelente introdução para o ORM.



Escrevi esse teste de unidade em um estilo bem verboso, como forma de apresentar o ORM do Django. Não recomendo que você escreva os testes para seus modelos desse modo “na vida real”. Na verdade, vamos reescrever esse teste mais tarde no Capítulo 15 de modo que ele se torne muito mais conciso.

Terminologia 2: testes de unidade *versus* testes integrados, e o banco de dados

Os puristas dirão que um “verdadeiro” teste de unidade jamais deveria entrar em contato com o banco de dados, e que o teste que acabei de escrever deveria ser chamado de forma mais apropriada de teste integrado porque ele não testa somente o nosso código, mas depende também de um sistema externo – isto é, de um banco de dados.

Não há problemas em ignorar essa distinção por enquanto – temos dois tipos de testes: os testes funcionais de alto nível, que testam a aplicação do ponto de vista do usuário, e esses testes de nível mais baixo, que testam do ponto de vista do programador.

Retomaremos esse assunto e discutiremos os testes de unidade e os testes integrados no Capítulo 23, mais próximo do final do livro.

Vamos experimentar executar o teste de unidade. Eis outro ciclo de testes de unidade/código:

```
ImportError: cannot import name 'Item'
```

Muito bem, vamos lhe dar algo para importar de *lists/models.py*. Estamos nos sentindo confiantes, portanto pularemos o passo `Item = None` e passaremos diretamente para a criação de uma classe:

lists/models.py

```
from django.db import models
```

```
class Item(object):  
    pass
```

Isso leva o nosso teste até este ponto:

```
first_item.save()  
AttributeError: 'Item' object has no attribute 'save'
```

Para dar um método `save` à nossa classe `Item` e transformá-la em um verdadeiro modelo do Django, fazemos com que ela herde da classe `Model`:

lists/models.py

```
from django.db import models
```

```
class Item(models.Model):  
    pass
```

Nossa primeira migração de banco de dados

A próxima ocorrência que vemos é um erro de banco de dados:

```
django.db.utils.OperationalError: no such table: lists_item
```

No Django, a tarefa do ORM é modelar o banco de dados, mas há um segundo sistema responsável por de fato construí-lo, e esse sistema se chama *migrações* (migrations). Sua tarefa é dar a você a capacidade de adicionar e remover tabelas e colunas com base nas mudanças que forem feitas em seus arquivos *models.py*.

Uma forma de pensar nesse sistema é vê-lo como um sistema de controle de versões para o seu banco de dados. Como veremos mais tarde, é um recurso particularmente útil se precisarmos atualizar um banco de dados implantado em um servidor ativo.

Por enquanto, tudo que precisamos saber é como construir nossa primeira migração de banco de dados, e faremos isso com o comando `makemigrations`:²

```
$ python manage.py makemigrations
```

```
Migrations for 'lists':
```

```
lists/migrations/0001_initial.py
```

```
- Create model Item
```

```
$ ls lists/migrations
```

```
0001_initial.py __init__.py __pycache__
```

Se estiver curioso, poderá dar uma olhada no arquivo de migrações, e verá que ele é uma representação de nossos acréscimos em *models.py*.

Nesse ínterim, perceberemos que nossos testes avançarão um pouco mais.

Os testes vão surpreendentemente longe

Os testes de fato vão surpreendentemente longe:

```
$ python manage.py test lists
```

```
[...]
```

```
self.assertEqual(first_saved_item.text, 'The first (ever) list item')
```

```
AttributeError: 'Item' object has no attribute 'text'
```

Estamos a oito linhas completas depois da última falha – passamos por todo o processo de salvar os dois `Items`, e verificamos que eles foram salvos no banco de dados, mas o Django simplesmente parece não se lembrar do atributo `.text`.

Incidentalmente, se o Python for uma novidade para você, talvez você tenha se surpreendido inclusive de podermos ter feito uma atribuição a `.text`. Em uma linguagem como Java, provavelmente você obteria um erro de compilação. O Python é mais flexível.

As classes que herdam de `models.Model` são mapeadas para tabelas no banco de dados. Por padrão, elas obtêm um atributo `id` gerado automaticamente, que será uma coluna de chave primária no banco de dados, mas você precisa definir explicitamente qualquer outra coluna desejada; eis o modo de definir um campo de texto:

lists/models.py

```
class Item(models.Model):
    text = models.TextField()
```

O Django tem muitos outros tipos de campos, como `IntegerField`, `CharField`, `DateField`, e assim por diante. Optei por `TextField` em vez de `CharField` porque esse último exige uma restrição de tamanho, o que pode parecer arbitrário neste ponto. Você pode ler mais sobre tipos de campos no tutorial do Django (<http://bit.ly/1sIDAGH>) e em sua documentação

(<https://docs.djangoproject.com/en/1.11/ref/models/fields/>).

Um novo campo implica uma nova migração

A execução dos testes resulta em outro erro de banco de dados:

```
django.db.utils.OperationalError: no such column: lists_item.text
```

Isso se deve ao fato de termos adicionado outro campo novo em nosso banco de dados, o que significa que precisamos criar outra migração. Nossos testes foram gentis em nos informar!

Vamos tentar:

```
$ python manage.py makemigrations
```

```
You are trying to add a non-nullable field 'text' to item without a default; we can't do that (the database needs something to populate existing rows).
```

```
Please select a fix:
```

```
1) Provide a one-off default now (will be set on all existing rows with a null value for this column)
```

2) Quit, and let me add a default in models.py
Select an option:2

Ah. Não podemos adicionar a coluna sem um valor default. Vamos escolher a opção 2 e definir um default em *models.py*. Imagino que você achará a sintaxe razoavelmente autoexplicativa:

lists/models.py

```
class Item(models.Model):  
    text = models.TextField(default="")
```

Agora a migração deverá ser concluída:

```
$ python manage.py makemigrations  
Migrations for 'lists':  
lists/migrations/0002_item_text.py  
- Add field text to item
```

Então duas novas linhas em *models.py*, duas migrações de banco de dados e, como resultado, o atributo `.text` em nossos objetos de modelo agora é reconhecido como um atributo especial, portanto é salvo no banco de dados, e os testes passam...

```
$ python manage.py test lists  
[...]
```

```
Ran 3 tests in 0.010s  
OK
```

Vamos, portanto, fazer um commit de nosso primeiríssimo modelo!

```
$ git status # veja tests.py, models.py e 2 migrações não controladas  
$ git diff # revise as mudanças em tests.py e em models.py  
$ git add lists  
$ git commit -m "Model for list Items and associated migration"
```

Salvando o POST no banco de dados

Vamos adaptar o teste para a requisição POST em nossa página inicial e dizer que queremos que a view salve um novo item no banco de dados em vez de simplesmente o passar para a sua resposta. Podemos fazer isso acrescentando três novas linhas no teste existente chamado `test_can_save_a_POST_request`:

lists/tests.py

```
def test_can_save_a_POST_request(self):
    response = self.client.post('/', data={'item_text': 'A new list item'})

    self.assertEqual(Item.objects.count(), 1) ❶
    new_item = Item.objects.first() ❷
    self.assertEqual(new_item.text, 'A new list item') ❸

    self.assertIn('A new list item', response.content.decode())
    self.assertTemplateUsed(response, 'home.html')
```

- ❶ Verificamos se um novo Item foi salvo no banco de dados. `objects.count()` é uma versão abreviada de `objects.all().count()`.
- ❷ `objects.first()` é o mesmo que `objects.all()[0]`.
- ❸ Verificamos se o texto do item está correto.

Esse teste está ficando um pouco extenso demais. Parece que muitos aspectos diferentes estão sendo testados. É outro *code smell* – um teste de unidade longo deve ser dividido em dois ou pode ser uma indicação de que o que você está testando é demasiadamente complicado. Vamos adicionar esse problema em uma pequena lista de tarefas nossa, talvez em um papel de rascunho:

• **Code smell: teste de POST é longo demais?**

Anotar isso em um papel de rascunho como esse nos garante que não esqueceremos, portanto nos sentimos à vontade para voltar àquilo em que estávamos trabalhando. Executamos os testes novamente e vemos uma falha esperada:

```
self.assertEqual(Item.objects.count(), 1)
AssertionError: 0 != 1
```

Vamos ajustar a nossa view:

lists/views.py

```
from django.shortcuts import render
from lists.models import Item
```

```
def home_page(request):
    item = Item()
    item.text = request.POST.get('item_text', "")
    item.save()

    return render(request, 'home.html', {
        'new_item_text': request.POST.get('item_text', "")
    })
```

Implementei uma solução bem ingênua e provavelmente você poderá identificar um problema bem óbvio, que é o fato de que salvaremos itens vazios a cada requisição feita para a página inicial. Vamos acrescentar isso em nossa lista de itens a serem corrigidos mais tarde. Isso, junto com o fato infelizmente evidente de que, no momento, não temos nenhuma maneira de ter listas distintas para pessoas diferentes. Continuaremos ignorando isso por enquanto.

Lembre-se de que não estou dizendo que você sempre deva ignorar problemas evidentes como esse na “vida real”. Sempre que identificarmos problemas com antecedência, há a necessidade de fazer uma avaliação para saber se devemos parar o que estamos fazendo e recomeçar ou se devemos deixá-los de lado até mais tarde. Às vezes, terminar o que estamos fazendo ainda valerá a pena, enquanto em outras ocasiões o problema poderá ser tão significativo que justificaria parar e repensar.

Vamos ver como os testes de unidade se saem... Eles passam! Muito bem. Podemos fazer um pouco de refatoração:

lists/views.py

```
return render(request, 'home.html', {
    'new_item_text': item.text
})
```

Vamos dar uma espiada rápida em nosso papel de rascunho. Acrescentei alguns dos outros itens que estão em nossa mente:

- **Não salvar itens em branco a cada requisição**
- **Code smell: teste de POST é longo demais?**

- **Exibir vários itens da tabela**
- **Aceitar mais de uma lista!**

Vamos começar pelo primeiro item. Poderíamos fazer ajustes em uma asserção em um teste existente, mas é melhor deixar que os testes de unidade testem um item de cada vez, portanto vamos adicionar um novo teste:

lists/tests.py

```
class HomePageTest(TestCase):
    [...]

    def test_only_saves_items_when_necessary(self):
        self.client.get('/')
        self.assertEqual(Item.objects.count(), 0)
```

Isso nos dá uma falha `1 != 0`. Vamos corrigi-la. Preste atenção: embora seja uma alteração bem pequena na lógica da view, há um bocado de pequenos ajustes no código da implementação:

lists/views.py

```
def home_page(request):
    if request.method == 'POST':
        new_item_text = request.POST['item_text'] ❶
        Item.objects.create(text=new_item_text) ❷
    else:
        new_item_text = "" ❶

    return render(request, 'home.html', {
        'new_item_text': new_item_text, ❶
    })
```

- ❶ Usamos uma variável chamada `new_item_text`, que armazenará o conteúdo do POST ou a string vazia.
- ❷ `.objects.create` é um bom atalho para criar um novo `Item`, sem a necessidade de chamar `.save()`.

Com isso, o teste passa:

```
Ran 4 tests in 0.010s
```

OK

Redirecionar após um POST

Mas, puxa vida, toda essa dança com `new_item_text = "` está me deixando muito insatisfeito. Felizmente temos agora uma oportunidade para corrigir isso. Uma função de view possui duas tarefas: processar entradas de usuário e devolver uma resposta apropriada. Cuidamos da primeira parte, que é salvar a entrada do usuário no banco de dados, portanto vamos trabalhar agora na segunda parte.

Dizem que sempre devemos fazer um redirecionamento depois de um POST (<https://en.wikipedia.org/wiki/Post/Redirect/Get>), portanto vamos fazê-lo. Mais uma vez, alteramos o nosso teste de unidade para salvar uma requisição POST a fim de dizer que, em vez de renderizar uma resposta com o item contido aí, ele deve fazer um redirecionamento de volta à página inicial:

lists/tests.py

```
def test_can_save_a_POST_request(self):
    response = self.client.post('/', data={'item_text': 'A new list item'})

    self.assertEqual(Item.objects.count(), 1)
    new_item = Item.objects.first()
    self.assertEqual(new_item.text, 'A new list item')

    self.assertEqual(response.status_code, 302)
    self.assertEqual(response['location'], '/')
```

Não esperamos mais uma resposta com um `.content` renderizado por um template, portanto perdemos as asserções que verificam isso. A resposta representará um *redirecionamento* HTTP, que deve ter um código de status 302 e apontará o navegador para uma nova localidade.

Isso nos dá o erro `200 != 302`. Agora podemos organizar substancialmente a nossa view:

lists/views.py (ch05l028)

```
from django.shortcuts import redirect, render
from lists.models import Item

def home_page(request):
    if request.method == 'POST':
        Item.objects.create(text=request.POST['item_text'])
        return redirect('/')

    return render(request, 'home.html')
```

Agora os testes devem passar:

```
Ran 4 tests in 0.010s
```

```
OK
```

Melhor prática para testes de unidade: cada teste deve testar somente um aspecto

Nossa view agora faz um redirecionamento após um POST, o que é uma boa prática, e reduzimos um tanto quanto o teste de unidade, mas podemos fazer algo melhor.

A boa prática para testes de unidade diz que cada teste deve testar apenas um aspecto. O motivo é que isso facilita o rastreamento de bugs. Ter várias asserções em um teste significa que, se o teste falhar em uma das asserções iniciais, não saberemos qual será o status das asserções finais. Como veremos no próximo capítulo, se provocarmos acidentalmente uma falha nessa view, queremos saber se essa falha está na hora de salvar os objetos ou no tipo da resposta.

Talvez nem sempre escreveremos testes de unidade perfeitos com asserções únicas na primeira tentativa, mas agora parece uma boa hora para separar nossas preocupações:

lists/tests.py

```
def test_can_save_a_POST_request(self):
```

```
self.client.post('/', data={'item_text': 'A new list item'})
```

```
self.assertEqual(Item.objects.count(), 1)
new_item = Item.objects.first()
self.assertEqual(new_item.text, 'A new list item')
```

```
def test_redirects_after_POST(self):
    response = self.client.post('/', data={'item_text': 'A new list item'})
    self.assertEqual(response.status_code, 302)
    self.assertEqual(response["location"], '/')
```

Agora devemos ver cinco testes passarem, em vez de quatro:

```
Ran 5 tests in 0.010s
```

OK

Renderizando itens no template

Muito melhor! Vamos voltar à nossa lista de tarefas:

- ~~Não salvar itens em branco a cada requisição~~
- ~~Code smell: teste de POST é longo demais?~~
- Exibir vários itens da tabela
- Aceitar mais de uma lista!

Riscar itens da lista é quase tão satisfatório quanto ver os testes passarem!

O terceiro item é o último dos “fáceis”. Vamos criar um novo teste de unidade que verifica se o template também é capaz de exibir vários itens de lista:

lists/tests.py

```
class HomePageTest(TestCase):
    [...]

    def test_displays_all_list_items(self):
        Item.objects.create(text='itemey 1')
```



```
Item.objects.create(text='itemey 2')

response = self.client.get('/')

self.assertIn('itemey 1', response.content.decode())
self.assertIn('itemey 2', response.content.decode())
```



Você está se perguntando sobre o espaçamento de linhas no teste? Estou agrupando duas linhas no início para configurar o teste, uma linha no meio que, na verdade, chama o código em teste, e as asserções no final. Isso não é obrigatório, mas ajuda a ver a estrutura do teste. Configurar, Exercitar, Fazer Asserções é a estrutura típica de um teste de unidade.

Esse teste falha conforme esperado:

```
AssertionError: 'itemey 1' not found in '<html>\n <head>\n [...]
```

A sintaxe de template do Django tem uma tag para iterar em listas, `{% for .. in .. %}`; podemos usá-la assim:

lists/templates/home.html

```
<table id="id_list_table">
  {% for item in items %}
    <tr><td>1: {{ item.text }}</td></tr>
  {% endfor %}
</table>
```

Esse é um dos principais pontos fortes do sistema de templating. Agora o template será renderizado com várias linhas `<tr>`, uma para cada item na variável `items`. Muito organizado! Apresentarei mais algumas pitadas da mágica de template do Django à medida que prosseguirmos, mas, em algum ponto, você deve ler sobre o restante delas na documentação do Django (<https://docs.djangoproject.com/en/1.11/topics/templates/>).

Somente alterar o template não faz com que nossos testes recebam o sinal verde; precisamos realmente passar os itens para ele a partir da view de nossa página inicial:

lists/views.py

```

def home_page(request):
    if request.method == 'POST':
        Item.objects.create(text=request.POST['item_text'])
        return redirect('/')

    items = Item.objects.all()
    return render(request, 'home.html', {'items': items})

```

Isso faz os testes de unidade passarem... Momento da verdade: o teste funcional passará?

```

$ python functional_tests.py
[...]
AssertionError: 'To-Do' not found in 'OperationalError at /'

```

Opa, aparentemente não. Vamos usar outra técnica de depuração de testes funcionais, e é uma das técnicas mais simples: acessar manualmente o site! Abra *http://localhost:8000* em seu navegador web e você verá uma página de depuração do Django informando o seguinte: “no such table: lists_item” (esta tabela não existe: lists_item), conforme vemos na Figura 5.2.

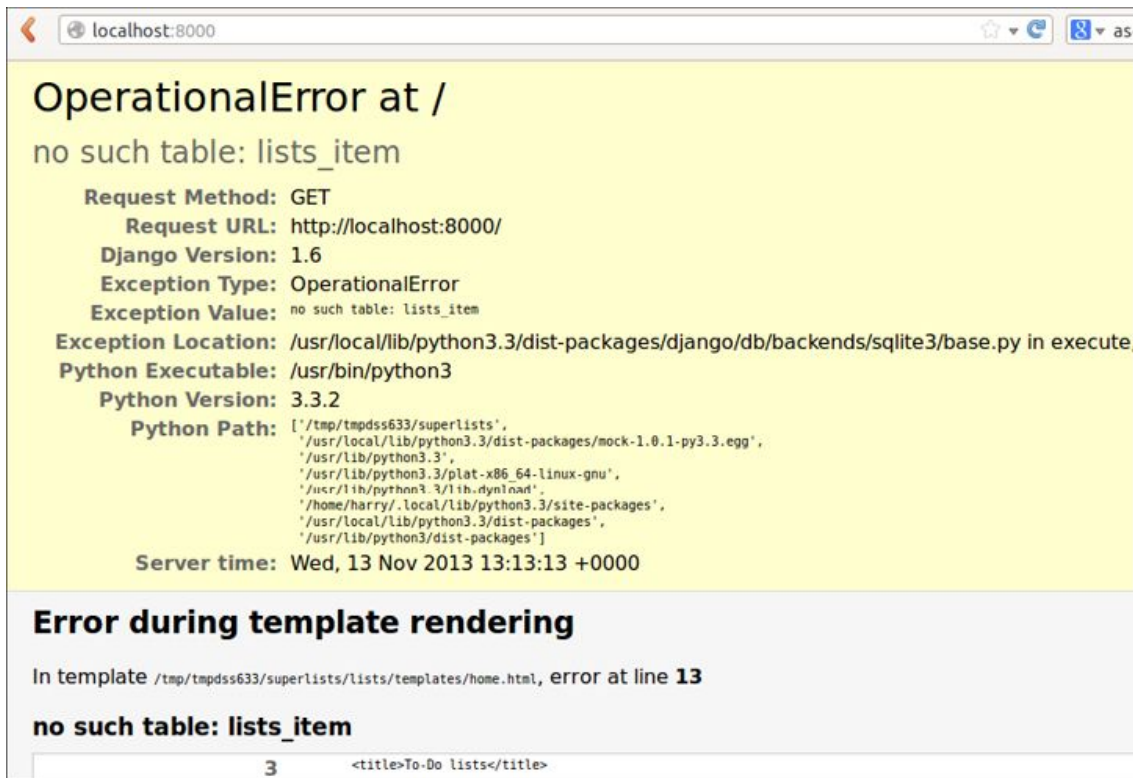


Figura 5.2 – Outra mensagem de depuração útil.

Criando nosso banco de dados de produção com migrate

Essa é outra mensagem de erro útil do Django, que basicamente está reclamando que não configuramos o banco de dados devidamente. Como então tudo funcionou bem nos testes de unidade? – posso ouvir você perguntar. Porque o Django cria um *banco de dados de teste* especial para os testes de unidade; é uma das mágicas que o TestCase do Django faz.

Para configurar o nosso “verdadeiro” banco de dados, precisamos criá-lo. Bancos de dados SQLite são apenas um arquivo em disco, e, em *settings.py*, veremos que, por padrão, o Django colocará o banco de dados em um arquivo chamado *db.sqlite3* no diretório-base do projeto:

superlists/settings.py

```
[...]
# Database
# https://docs.djangoproject.com/en/1.11/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Informamos tudo de que o Django precisa para criar o banco de dados, inicialmente por meio de *models.py* e então quando criamos o arquivo de migrações. Para realmente aplicar isso na criação de um verdadeiro banco de dados, usaremos outro comando de *manage.py*, que é o canivete suíço do Django – o comando migrate:

```
$ python manage.py migrate
```

```
Operations to perform:
```

```
Apply all migrations: admin, auth, contenttypes, lists, sessions
```

```
Running migrations:
```

```
Applying contenttypes.0001_initial... OK
```

```
Applying auth.0001_initial... OK
Applying admin.0001_initial... OK
Applying admin.0002_logentry_remove_auto_add... OK
Applying contenttypes.0002_remove_content_type_name... OK
Applying auth.0002_alter_permission_name_max_length... OK
Applying auth.0003_alter_user_email_max_length... OK
Applying auth.0004_alter_user_username_opts... OK
Applying auth.0005_alter_user_last_login_null... OK
Applying auth.0006_require_contenttypes_0002... OK
Applying auth.0007_alter_validators_add_error_messages... OK
Applying auth.0008_alter_user_username_max_length... OK
Applying lists.0001_initial... OK
Applying lists.0002_item_text... OK
Applying sessions.0001_initial... OK
```

Agora podemos atualizar a página em *localhost*, ver que nosso erro sumiu e tentar executar os testes funcionais novamente:³

```
AssertionError: '2: Use peacock feathers to make a fly' not found in ['1: Buy peacock feathers', '1: Use peacock feathers to make a fly']
```

Estamos tão perto! Só precisamos que a numeração de nossa lista esteja correta. Outra tag de template incrível do Django, `forloop.counter`, ajudará nesse caso:

lists/templates/home.html

```
{% for item in items %}
  <tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>
{% endfor %}
```

Se tentar novamente, você deverá ver o FT chegar ao final:

```
self.fail('Finish the test!')
AssertionError: Finish the test!
```

No entanto, à medida que o teste executar, você poderá perceber que algo está errado, como vemos na Figura 5.3.

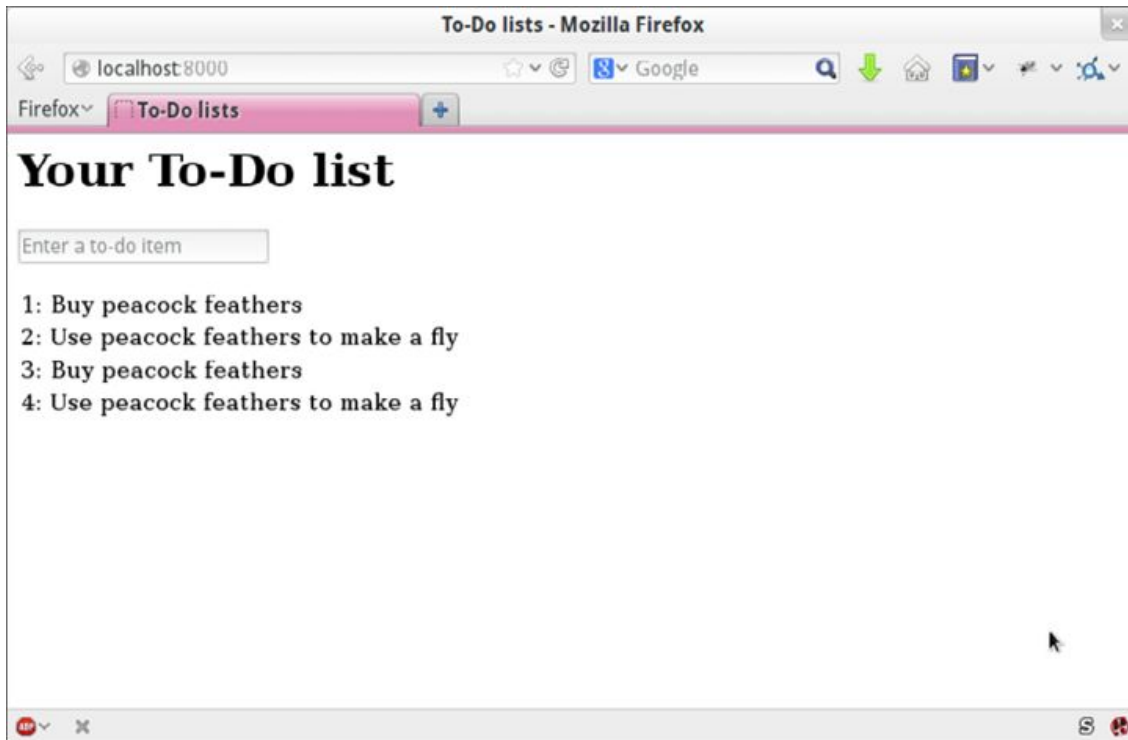


Figura 5.3 – Há itens de lista remanescentes da última execução do teste.

Oh, céus. Parece que execuções anteriores do teste estão deixando dados sobrando em nosso banco de dados. De fato, se você executar os testes novamente, verá que a situação piora:

- 1: Buy peacock feathers
- 2: Use peacock feathers to make a fly
- 3: Buy peacock feathers
- 4: Use peacock feathers to make a fly
- 5: Buy peacock feathers
- 6: Use peacock feathers to make a fly

Grrr. Estamos tão perto! Precisaremos de alguma maneira automatizada de fazer uma limpeza. Por enquanto, se quiser, você poderá fazer isso manualmente apagando o banco de dados e recriando-o do zero usando migrate:

```
$ rm db.sqlite3  
$ python manage.py migrate --noinput
```

Em seguida, garanta que o FT continue passando.

Exceto por esse pequeno bug em nosso teste funcional, temos um

código que funciona mais ou menos. Vamos fazer um commit.

Comece executando um `git status` e um `git diff`, e você deverá ver mudanças em `home.html`, `tests.py` e `views.py`. Vamos adicioná-los:

```
$ git add lists
```

```
$ git commit -m "Redirect after POST, and show all items in template"
```



Talvez você ache útil acrescentar marcadores para o final de cada capítulo, por exemplo, `git tag end-of-chapter-05`.

Revisão

Qual é a nossa situação?

- Conseguimos criar um formulário para adicionar novos itens à lista usando POST.
- Criamos um modelo simples no banco de dados para salvar itens de lista.
- Aprendemos a criar migrações de banco de dados, tanto para o banco de dados de teste (no qual elas são aplicadas automaticamente) quanto para o banco de dados real (no qual temos de aplicá-las manualmente).
- Usamos nossas duas primeiras tags de template no Django: `{% csrf_token %}` e o laço `{% for ... endfor %}`.
- Além disso, utilizamos no mínimo três técnicas diferentes de depuração de FT: instruções `print in-line`, `time.sleeps` e melhoria das mensagens de erro.

Contudo, tivemos alguns itens inseridos em nossa própria lista de tarefas, a saber: fazer o FT efetuar sua própria limpeza e, talvez como item mais crítico, acrescentar suporte para mais de uma lista.

- ~~Não salvar itens em branco a cada requisição~~
- ~~Code smell: teste de POST é longo demais?~~
- ~~Exibir vários itens da tabela~~

- **Fazer uma limpeza após executar o FT**
- **Aceitar mais de uma lista!**

Quero dizer, *poderíamos* lançar o site como está, mas as pessoas poderão achar estranho o fato de toda a população humana ter que compartilhar uma única lista de tarefas. Suponho que isso poderia fazer as pessoas pararem e pensarem sobre como estamos todos conectados uns aos outros, como todos compartilhamos um destino comum aqui na Espaçoave Terra e como devemos todos trabalhar em conjunto para solucionar os problemas globais que enfrentamos.

Em termos práticos, porém, o site não seria muito útil.

Pois é.

Conceitos úteis de TDD

Regressão

É quando um código novo provoca falha em algum aspecto da aplicação que costumava funcionar.

Falha inesperada

É quando um teste falha de uma maneira que não estávamos esperando. Isso significa que cometemos um erro em nossos testes ou que os testes nos ajudaram a identificar uma regressão, e precisamos fazer alguma correção em nosso código.

Vermelho/Verde/Refatorar

É outra forma de descrever o processo de TDD. Consiste em escrever um teste e o ver falhar (Vermelho), escrever um pouco de código para fazer o teste passar (Verde) e, então, Refatorar a fim de melhorar a implementação.

Triangulação

Consiste em adicionar um caso de teste com um novo exemplo específico para algum código existente a fim de justificar a generalização da implementação (o que poderia ser uma “trapaça” até esse ponto).

Três acertos e refatorar

É uma regra geral para a remoção de duplicações no código. Quando duas porções de código parecerem muito semelhantes, com frequência vale a pena esperar até vermos um terceiro caso de uso, de modo que tenhamos mais certeza sobre qual parte do código realmente é a parte comum e reutilizável a ser refatorada.

A lista de tarefas em um papel de rascunho

É um local para anotar fatos que nos ocorrem à medida que programamos, a fim de terminarmos o que estamos fazendo e retomarmos esses pontos mais tarde.

-
- 1 Caso você ainda não tenha deparado com o conceito, um “code smell” (mau cheiro no código) refere-se a algo em uma porção de código que faça com que você queira reescrevê-la. Jeff Atwood apresenta uma compilação de informações em seu blog Coding Horror (<http://www.codinghorror.com/blog/2006/05/code-smells.html>). Quanto mais experiência você adquirir como programador, mais apurado ficará o seu olfato para code smells...
 - 2 Você está se perguntando quando vamos executar “migrate”, assim como “makemigrations”? Continue lendo; isso será feito mais adiante no capítulo.
 - 3 Se você obtiver um erro diferente nesse ponto, experimente reiniciar o seu servidor de desenvolvimento – talvez ele tenha ficado confuso pelo fato de as alterações no banco de dados terem puxado o seu tapete.

CAPÍTULO 6

Melhorando os testes funcionais: garantindo o isolamento e removendo sleeps vodus

Antes de mergulhar de cabeça e corrigir nosso verdadeiro problema, vamos cuidar de alguns itens relacionados à organização da casa. No final do último capítulo, destacamos que diferentes execuções dos testes estavam interferindo umas com as outras, portanto vamos corrigir isso. Também não estou satisfeito com todos aqueles `time.sleeps` espalhados pelo código; parecem pouco científicos, portanto vamos substituí-los por algo mais confiável.

- Fazer uma limpeza após executar o FT
- Remover `time.sleeps`

Essas duas alterações estarão na direção das “melhores práticas” de testes, deixando nossos testes mais determinísticos e confiáveis.

Garantindo o isolamento dos testes em testes funcionais

Terminamos o último capítulo com um problema clássico de testes: como garantir o *isolamento* entre os testes. Cada execução de nossos testes funcionais estava deixando itens de lista sobrando no banco de dados, e isso vai interferir no resultado dos testes em sua

próxima execução.

Quando executamos testes de *unidade*, o executor de testes do Django cria automaticamente um banco de dados de teste totalmente novo (separado do verdadeiro), que pode ser reiniciado com segurança antes que cada teste individual seja executado, e então ser descartado no final. Contudo, nossos testes funcionais atualmente executam em um banco de dados “real”, o *db.sqlite3*.

Uma forma de enfrentar esse problema seria “desenvolver” a nossa própria solução e acrescentar código em *functional_tests.py* para fazer a limpeza. Os métodos `setUp` e `tearDown` são perfeitos para esse tipo de tarefa.

A partir do Django 1.4, porém, há uma classe nova chamada `LiveServerTestCase`, que pode fazer esse trabalho para você. Ela criará automaticamente um banco de dados de testes (assim como em uma execução de teste de unidade), e iniciará um servidor de desenvolvimento com o qual os testes funcionais serão executados. Embora, como ferramenta, ela tenha algumas limitações que teremos de contornar mais tarde, essa classe será extremamente útil nessa fase, portanto vamos analisá-la.

`LiveServerTestCase` espera ser executada pelo executor de testes do Django usando *manage.py*. No Django 1.6, o executor de testes encontrará qualquer arquivo cujo nome comece com *test*. Para manter tudo organizado, vamos criar uma pasta para nossos testes funcionais para que pareça um pouco que temos uma aplicação. Tudo de que o Django precisa é que esse seja um diretório de pacotes Python válido (isto é, um que contenha um `__init__.py`):

```
$ mkdir functional_tests  
$ touch functional_tests/__init__.py
```

Então *movemos* nossos testes funcionais, passando-os de um arquivo standalone chamado *functional_tests.py* para ser o *tests.py* da aplicação `functional_tests`. Usamos `git mv` para que o Git perceba que movemos o arquivo:

```
$ git mv functional_tests.py functional_tests/tests.py
```


\$ **git status** # mostra arquivo renomeado para `functional_tests/tests.py` e `__init__.py`

A essa altura, sua árvore de diretório deverá ter o seguinte aspecto:

```
.
├── db.sqlite3
├── functional_tests
│   ├── __init__.py
│   └── tests.py
├── lists
│   ├── admin.py
│   ├── apps.py
│   ├── __init__.py
│   ├── migrations
│   ├── 0001_initial.py
│   ├── 0002_item_text.py
│   ├── __init__.py
│   └── __pycache__
├── models.py
├── __pycache__
├── templates
│   └── home.html
├── tests.py
├── views.py
├── manage.py
└── superlists
    ├── __init__.py
    ├── __pycache__
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

`functional_tests.py` não existe mais, passando a ser `functional_tests/tests.py`. Agora, sempre que quisermos executar nossos testes funcionais, em vez de executar `python functional_tests.py`, usaremos `python manage.py test functional_tests`.



Você poderia misturar seus testes funcionais nos testes da aplicação `lists`. Minha tendência é mantê-los separados porque os testes funcionais geralmente têm preocupações que permeiam diferentes aplicações. Os FTs

são criados com o intuito de ver a situação do ponto de vista de seus usuários, e estes não se importam com o modo como você separou as tarefas entre as diferentes aplicações!

Vamos agora editar `functional_tests/tests.py` e alterar a nossa classe `NewVisitorTest` para fazê-la usar `LiveServerTestCase`:

functional_tests/tests.py (ch06l001)

```
from django.test import LiveServerTestCase
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
import time
```

```
class NewVisitorTest(LiveServerTestCase):
```

```
    def setUp(self):
        [...]
```

Em seguida, em vez de deixar fixo o acesso à porta 8000 do localhost, `LiveServerTestCase` disponibiliza um atributo chamado `live_server_url`:

functional_tests/tests.py (ch06l002)

```
    def test_can_start_a_list_and_retrieve_it_later(self):
        # Edith ouviu falar de uma nova aplicação online interessante para
        # lista de tarefas. Ela decide verificar sua homepage
        self.browser.get(self.live_server_url)
```

Também podemos remover `if name == 'main'` do final se quisermos, pois usaremos o executor de testes do Django para iniciar o FT.

Agora podemos executar nossos testes funcionais usando o executor de testes do Django, dizendo-lhe que execute somente os testes para a nossa nova aplicação `functional_tests`:

```
$ python manage.py test functional_tests
```

```
Creating test database for alias 'default'...
```

```
F
```

```
=====
=====
```

```
FAIL: test_can_start_a_list_and_retrieve_it_later
(functional_tests.tests.NewVisitorTest)
```

```
-----  
Traceback (most recent call last):  
  File "../superlists/functional_tests/tests.py", line 65, in  
test_can_start_a_list_and_retrieve_it_later  
    self.fail("Finish the test!")  
AssertionError: Finish the test!
```

```
-----  
Ran 1 test in 6.578s
```

```
FAILED (failures=1)  
System check identified no issues (0 silenced).  
Destroying test database for alias 'default'...
```

O FT chega até `self.fail`, exatamente como fazia antes da refatoração. Você também perceberá que, se os testes forem executados uma segunda vez, não haverá nenhum item de lista antigo remanescente do teste anterior – o próprio teste realizar a limpeza. Sucesso! Devemos fazer commit disso como uma alteração atômica:

```
$ git status # functional_tests.py renomeado + modificado, novo __init__.py  
$ git add functional_tests  
$ git diff --staged -M  
$ git commit # msg por exemplo "make functional_tests an app,  
# use LiveServerTestCase"
```

A flag `-M` em `git diff` é útil. Ela quer dizer “detectar *moves*”, portanto perceberá que `functional_tests.py` e `functional_tests/tests.py` são o mesmo arquivo, e mostrará um diff mais sensato para você (experimente executar o comando sem a flag!).

Executando somente os testes de unidade

Se executarmos `manage.py test` agora, o Django executará tanto os testes funcionais quanto os testes de unidade:

```
$ python manage.py test  
Creating test database for alias 'default'...  
.....F  
=====
```

FAIL: test_can_start_a_list_and_retrieve_it_later

[...]

AssertionError: Finish the test!

Ran 7 tests in 6.732s

FAILED (failures=1)

Para executar somente os testes de unidade, podemos especificar que queremos executar apenas os testes para a aplicação lists:

\$ python manage.py test lists

Creating test database for alias 'default'...

.....

Ran 6 tests in 0.009s

OK

System check identified no issues (0 silenced).

Destroying test database for alias 'default'...

Comandos úteis atualizados

Para executar os testes funcionais

```
python manage.py test functional_tests
```

Para executar os testes de unidade

```
python manage.py test lists
```

O que você faria se eu dissesse “execute os testes” e você não estivesse certo sobre os testes aos quais me refiro? Observe novamente o fluxograma no final do Capítulo 4 e tente descobrir em que ponto estamos. Como regra geral, executamos normalmente os testes funcionais somente depois que todos os testes de unidade passarem; portanto, se estiver em dúvida, experimente executar ambos!

Informação extra: atualizando o Selenium e o Geckodriver

Hoje, enquanto estava trabalhando neste capítulo novamente, percebi que os FTs estavam travando quando tentei executá-los.

O fato é que o Firefox havia se atualizado automaticamente durante a noite, e minhas versões de Selenium e de Geckodriver também precisavam ser atualizadas. Um acesso rápido à página de versões lançadas do geckodriver (<https://github.com/mozilla/geckodriver/releases>) confirmou que havia uma nova versão disponível. Assim, alguns downloads e upgrades se faziam necessários:

- Um rápido `pip install --upgrade selenium` antes.
- Então um download rápido do novo geckodriver.
- Salvei uma cópia de backup da versão antiga em um lugar por aí e coloquei a versão nova em seu lugar em algum ponto do PATH.
- Uma verificação rápida com `geckodriver --version` confirma que a nova versão estava pronta para ser usada.

Os FTs então voltaram a executar do modo como eu estava esperando.

Não houve nenhum motivo particular para que isso tivesse acontecido neste ponto do livro; de fato, é bem pouco provável que vá acontecer agora com você, mas poderá ocorrer em algum momento, e este parece um bom lugar para falar desse assunto, pois estamos fazendo uma reorganização da casa.

É um dos fatos com os quais você terá que conviver ao trabalhar com o Selenium. Embora seja possível fixar as versões de seu navegador e do Selenium (em um servidor de Integração Contínua, por exemplo), as versões de navegador não permanecem imutáveis no mundo real, e você precisa se manter em sintonia com as versões que seus usuários têm.



Se algo estranho estiver acontecendo com seus FTs, vale a pena tentar atualizar o Selenium.

Estamos agora de volta à programação normal.

Sobre esperas implícitas e explícitas, e time.sleeps vodus

Vamos falar sobre o `time.sleep` em nosso FT:

`functional_tests/tests.py`

```
# Quando ela tecla enter, a página é atualizada, e agora a página lista  
# "1: Buy peacock feathers" como um item em uma lista de tarefas  
inputbox.send_keys(Keys.ENTER)  
time.sleep(1)
```

```
self.check_for_row_in_list_table('1: Buy peacock feathers')
```

É isso que chamamos de “espera explícita”. É o oposto das “esperas implícitas”: em certos casos, o Selenium tenta esperar “automaticamente” para você quando acha que a página está carregando. Ele até disponibiliza um método chamado `implicitly_wait` que permite que você controle quanto tempo ele esperará caso você peça um elemento que aparentemente ainda não está na página.

De fato, na primeira edição, pude contar totalmente com as esperas implícitas. O problema é que elas são sempre um pouco instáveis, e, com o lançamento do Selenium 3, se tornaram menos confiáveis ainda. Ao mesmo tempo, a opinião geral da equipe do Selenium é que as esperas implícitas foram simplesmente uma péssima ideia, e devem ser evitadas.

Assim, esta edição tem esperas explícitas desde o início. O problema, porém, é que esses `time.sleeps` apresentam os próprios problemas. Atualmente estamos esperando um segundo, mas quem pode dizer que essa é a duração correta? Para a maior parte dos testes que executamos em nossa própria máquina, um segundo será tempo demais, e vai realmente retardar a execução de nossos FTs. 0, 1 segundo seria apropriado. Contudo, o problema é que, se configurarmos um valor tão baixo como esse, com frequência teremos uma falha espúria porque, por algum motivo qualquer, o notebook poderia estar um pouco mais lento exatamente naquele instante. E, mesmo com um segundo, não poderemos jamais ter certeza absoluta de que não vamos obter falhas aleatórias que não representarão um verdadeiro problema, e falso-positivos em testes são realmente irritantes (há muito mais informações sobre esse assunto em um artigo de Martin Fowler (<https://martinfowler.com/articles/nonDeterminism.html>)).



Erros inesperados de `NoSuchElementException` e `StaleElementException` são sintomas comuns quando nos esquecemos de usar uma espera explícita. Experimente remover o `time.sleep` e veja se você obtém um erro desse tipo.

Vamos então substituir nossos sleeps com uma ferramenta que esperará pelo tempo que for necessário, até um bom timeout demorado para capturar qualquer glitch. Renomearemos `check_for_row_in_list_table` para `wait_for_row_in_list_table`, e adicionaremos aí uma lógica de polling/retentativa:

`functional_tests/tests.py` (ch06I004)

```
from selenium.common.exceptions import WebDriverException
```

```
MAX_WAIT = 10 ❶
```

```
[...]
```

```
def wait_for_row_in_list_table(self, row_text):  
    start_time = time.time()  
    while True: ❷  
        try:  
            table = self.browser.find_element_by_id('id_list_table') ❸  
            rows = table.find_elements_by_tag_name('tr')  
            self.assertIn(row_text, [row.text for row in rows])  
            return ❹  
        except (AssertionError, WebDriverException) as e: ❺  
            if time.time() - start_time > MAX_WAIT: ❻  
                raise e ❻  
            time.sleep(0.5) ❺
```

- ❶ Usaremos uma constante chamada MAX_WAIT para definir o tempo máximo que estamos preparados para esperar. Dez segundos deve ser mais que suficiente para qualquer glitch ou lentidão aleatória.
- ❷ Aqui está o laço, que será executado indefinidamente, a menos que alcancemos uma das duas possíveis rotas de saída.
- ❸ Aqui estão nossas três linhas de asserções da versão antiga do método.
- ❹ Se passarmos por elas e nossa asserção passar, retornamos da função e escapamos do laço.
- ❺ Entretanto, se capturarmos uma exceção, esperaremos um período de tempo curto e continuaremos no laço para uma nova tentativa. Há dois tipos de exceções que queremos capturar: WebDriverException para quando a página não tiver sido carregada e o Selenium não puder encontrar o elemento da tabela na página, e AssertionError para quando a tabela estiver presente, mas talvez seja uma tabela de antes de a página ter sido recarregada, portanto ela ainda não conterá a nossa linha.
- ❻ Aqui está a nossa segunda rota de escape. Se chegarmos até

esse ponto, significa que nosso código ficou lançando exceções sempre que fizemos uma tentativa, até o nosso timeout esgotar. Portanto, dessa vez, lançamos a exceção novamente e deixamos que ela alcance o nosso teste e, mais provavelmente, acabe em nosso traceback, informando-nos por que o teste falhou.

Você está achando esse código um pouco feio, tornando um pouco mais difícil ver exatamente o que estamos fazendo? Eu concordo. Mais tarde, vamos refatorar um `wait_for` auxiliar e genérico a fim de separar a lógica de tempo e de relançamento da exceção do restante das asserções. Contudo, vamos esperar até precisarmos disso em vários lugares.



Se você já usou o Selenium antes, talvez saiba que ele tem algumas funções auxiliares para fazer as esperas (http://www.seleniumhq.org/docs/04_webdriver_advanced.jsp). Não sou grande fã delas. Ao longo deste livro, construiremos algumas ferramentas auxiliares de espera, as quais acho que resultarão em um código elegante e legível, mas é claro que você deve dar uma olhada nas esperas implementadas pelo Selenium em seu tempo livre e ver o que acha delas.

Agora podemos renomear nossas chamadas de métodos e remover os `time.sleeps` vodus:

functional_tests/tests.py (ch06I005)

```
[...]
# Quando ela tecla enter, a página é atualizada, e agora a página lista
# "1: Buy peacock feathers" como um item em uma lista de tarefas
inputbox.send_keys(Keys.ENTER)
self.wait_for_row_in_list_table('1: Buy peacock feathers')

# Ainda continua havendo uma caixa de texto convidando-a a acrescentar
# outro item. Ela insere "Use peacock feathers to make a fly"
# (Usar penas de pavão para fazer um fly – Edith é bem metódica)
inputbox = self.browser.find_element_by_id('id_new_item')
inputbox.send_keys('Use peacock feathers to make a fly')
inputbox.send_keys(Keys.ENTER)

# A página é atualizada novamente e agora mostra os dois itens em sua lista
```

```
self.wait_for_row_in_list_table('2: Use peacock feathers to make a fly')
self.wait_for_row_in_list_table('1: Buy peacock feathers')
[...]
```

Vamos executar os testes novamente:

```
$ python manage.py test
```

```
Creating test database for alias 'default'...
```

```
.....F
```

```
=====
=====
```

```
FAIL: test_can_start_a_list_and_retrieve_it_later
(functional_tests.tests.NewVisitorTest)
```

```
-----
Traceback (most recent call last):
```

```
File ".../superlists/functional_tests/tests.py", line 73, in
test_can_start_a_list_and_retrieve_it_later
    self.fail('Finish the test!')
```

```
AssertionError: Finish the test!
```

```
-----
Ran 7 tests in 4.552s
```

```
FAILED (failures=1)
```

```
System check identified no issues (0 silenced).
```

```
Destroying test database for alias 'default'...
```

Chegamos ao mesmo ponto; note que eliminamos alguns segundos do tempo da execução também. Pode não parecer muito no momento, mas esse valor vai sendo acumulado.

Apenas para verificar se o que fizemos está correto, vamos deliberadamente causar uma falha no teste de algumas maneiras e ver alguns erros. Inicialmente veremos se o erro correto será obtido caso procuremos uma linha de texto que jamais aparecerá:

functional_tests/tests.py (ch06I006)

```
rows = table.find_elements_by_tag_name('tr')
self.assertIn('foo', [row.text for row in rows])
return
```

Vemos que continuamos obtendo uma bela mensagem de falha de

teste autoexplicativa:

```
self.assertIn('foo', [row.text for row in rows])
AssertionError: 'foo' not found in ['1: Buy peacock feathers']
```

Vamos restaurar o código e provocar uma falha em outro ponto:

functional_tests/tests.py (ch06I007)

```
try:
    table = self.browser.find_element_by_id('id_nothing')
    rows = table.find_elements_by_tag_name('tr')
    self.assertIn(row_text, [row.text for row in rows])
    return
[...]
```

Sem dúvida, vamos também obter erros quando a página não contiver o elemento que estamos procurando:

```
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate
element: [id="id_nothing"]
```

Tudo parece em ordem. Vamos colocar o nosso código de volta, como deveria ser, e fazer uma última execução do teste:

```
$ python manage.py test
[...]
```

```
AssertionError: Finish the test!
```

Ótimo. Após esse rápido interlúdio, vamos trabalhar com afinco para fazer nossa aplicação realmente funcionar com várias listas.

As “melhores práticas” de testes aplicadas neste capítulo

Garantir o isolamento dos testes e administrar o estado global

Testes distintos não devem afetar uns aos outros. Isso significa que devemos reiniciar qualquer estado permanente no final de cada teste. O executor de testes do Django nos ajuda a fazer isso criando um banco de dados de testes, que é totalmente limpo entre cada teste. (Veja também o Capítulo 23.)

Evitar sleeps “vodus”

Sempre que precisarmos esperar que algo seja carregado, é tentador lançar mão de um `time.sleep` sujo e rápido. O problema, porém, é que o período de tempo que esperamos é sempre uma espécie de tiro no escuro, pois será curto demais e vulnerável a falhas espúrias, ou será longo demais e retardará a execução de nossos testes. Dê preferência a um laço de tentativas que faça polling de nossa aplicação e siga em frente o mais rápido possível.

Não depender das esperas implícitas do Selenium

O Selenium teoricamente faz algumas esperas “implícitas”, porém a implementação varia entre navegadores, e, quando escrevemos este livro, não era nada confiável no driver do Firefox no Selenium 3. Como diz o Zen de Python, “Explícito é melhor que implícito”, então prefira as esperas explícitas.

CAPÍTULO 7

Trabalhando de forma incremental

Vamos agora cuidar de nosso verdadeiro problema, que é o fato de o nosso design permitir somente uma lista global. Neste capítulo, mostrarei uma técnica crucial de TDD: como adaptar um código existente usando um processo incremental, passo a passo, que leve você de um estado funcional a outro estado funcional. Testing Goat, e não o Refactoring Cat.

Design pequeno quando necessário

Vamos pensar em como queremos oferecer suporte para o funcionamento de várias listas. Atualmente, o FT (que é o mais próximo que temos de um documento de design) diz o seguinte:

`functional_tests/tests.py`

```
# Edith se pergunta se o site lembrará de sua lista. Então ela nota
# que o site gerou um URL único para ela -- há um pequeno
# texto explicativo para isso.
self.fail("Finish the test!")
```

```
# Ela acessa esse URL – sua lista de tarefas continua lá.
```

```
# Satisfeita, ela volta a dormir
```

Queremos realmente ampliar isso, dizendo que diferentes usuários não vejam as listas uns dos outros, e cada um obtenha seu próprio URL como uma forma de retornar às suas listas que foram salvas. Como seria a aparência de um novo design?

Sem um grande design logo no início

O TDD está intimamente relacionado com o movimento ágil (agile) no desenvolvimento de software, que inclui uma reação contra o *Big Design Up Front* (Grande design logo no início) – a tradicional prática da engenharia de software segundo a qual, após um exercício demorado de coleta de requisitos, há uma etapa igualmente demorada de design na qual o software é planejado no papel. De acordo com a filosofia ágil, aprendemos mais com a resolução de problemas na prática do que com a teoria, especialmente quando confrontamos nossa aplicação com usuários reais o mais rápido possível. Em vez de ter uma fase prévia demorada de design, tentamos disponibilizar uma *aplicação mínima viável* (minimum viable application) mais cedo e deixamos o design evoluir gradualmente com base no feedback resultante do uso no mundo real.

Contudo, isso não significa que pensar no design seja uma tarefa definitivamente banida! No último grande capítulo, vimos que seguir em frente de forma tumultuada, sem raciocinar, pode, *eventualmente*, nos levar à resposta correta, mas, com frequência, pensar um pouco sobre o design pode nos ajudar a chegar lá mais rápido. Desse modo, vamos pensar em nossa aplicação mínima viável para listas e no tipo de design de que precisaremos para disponibilizá-la:

- Queremos que cada usuário seja capaz de armazenar a própria lista – pelo menos uma, por enquanto.
- Uma lista é composta de vários itens cujo atributo principal é um pequeno texto descritivo.
- Precisamos salvar as listas de uma visita para a próxima. Por enquanto, podemos dar a cada usuário um único URL para suas listas. Mais tarde, talvez queiramos ter alguma maneira de reconhecer automaticamente os usuários e mostrar-lhes as suas listas.

Para disponibilizar os itens “por enquanto”, parece que vamos armazenar as listas e seus itens em um banco de dados. Cada lista terá um URL exclusivo, e cada item da lista será um pequeno texto descritivo, associado a uma lista em particular.

YAGNI!

Depois que começarmos a pensar no design, pode ser difícil parar. Todo tipo de raciocínio virá à nossa mente – talvez queiramos dar um nome ou um título a cada lista, podemos reconhecer os usuários usando nome de usuário e senhas, adicionar um campo mais extenso de notas, bem como de descrições breves para a nossa lista, armazenar algum tipo de ordenação, e assim por diante. Contudo, obedecemos a outro princípio da doutrina do desenvolvimento ágil: “YAGNI”, que quer dizer “You ain’t gonna need it!” (Você não vai precisar disso!). Como desenvolvedores de software, nós nos divertimos criando algo, e às vezes é difícil resistir ao impulso de fazê-lo só porque uma ideia nos ocorreu e *talvez* precisemos dela. O problema é que, com muita frequência, independentemente do quão interessante seja a ideia, ela acaba *não* sendo usada. Teremos bastante código não utilizado, aumentando a complexidade da aplicação. O YAGNI é o mantra que usamos para resistir aos nossos impulsos criativos demasiadamente entusiásticos.

(No estilo) REST

Temos uma ideia da estrutura de dados que queremos – a parte referente ao Modelo do MVC (Model-View-Controller, ou Modelo-Visão-Controlador). O que dizer das partes que compõem a Visão e o Controlador? Como o usuário deve interagir com Lists e suas Items usando um navegador web?

O REST (Representational State Transfer, ou Transferência de Estado Representativo) é uma abordagem de web design, geralmente usada para orientar o design de API para web. Ao fazer

o design de um site voltado ao usuário, não é possível ater-se *estritamente* às regras de REST, mas elas podem oferecer um pouco de inspiração conveniente (vá direto ao Apêndice F se quiser ver uma API REST de verdade).

O REST sugere que tenhamos uma estrutura de URL que corresponda à nossa estrutura de dados – nesse caso, listas e itens de lista. Cada lista pode ter o próprio URL:

```
/lists/<identificador da lista>/
```

Isso atenderá ao requisito que especificamos em nosso FT. Para visualizar uma lista, usamos uma requisição GET (um acesso comum à página, feita por um navegador).

Para criar uma lista totalmente nova, teremos um URL especial que aceite requisições POST:

```
/lists/new
```

Para adicionar um novo item em uma lista existente, teremos um URL separado, para o qual poderemos enviar requisições POST:

```
/lists/<identificador da lista>/add_item
```

(Novamente, não estamos tentando seguir exatamente as regras de REST, que usaria uma requisição PUT nesse caso – estamos usando REST apenas como inspiração. Além do mais, não podemos usar PUT em um formulário HTML padrão.)

Em suma, nossa folha de rascunho para este capítulo tem um aspecto semelhante a este:

- **Ajustar o modelo para que os itens sejam associados a listas distintas**
- **Adicionar URLs exclusivos para cada lista**
- **Adicionar um URL para criar uma nova lista via POST**
- **Adicionar URLs para acrescentar um novo item em uma lista existente via POST**

Implementando o novo design de forma incremental usando TDD

Como usamos o TDD para implementar o novo design? Vamos observar novamente o fluxograma do processo de TDD na Figura 7.1.

No nível superior, usaremos uma combinação de adição de nova funcionalidade (acrescentando um novo FT e escrevendo um código novo para a aplicação) e refatoração de nossa aplicação – isto é, reescrever parte da implementação existente para que a mesma funcionalidade seja disponibilizada ao usuário, porém usando aspectos de nosso novo design. Poderemos usar o teste funcional existente para verificar se não causamos falhas no que já funciona, e o novo teste funcional para orientar os novos recursos.

No nível dos testes de unidade, acrescentaremos novos testes ou modificaremos os testes existentes a fim de testar as mudanças que queremos e, de modo semelhante, poderemos usar os testes de unidade que não sejam alterados para ajudar a garantir que não provocamos falhas em nada no processo.

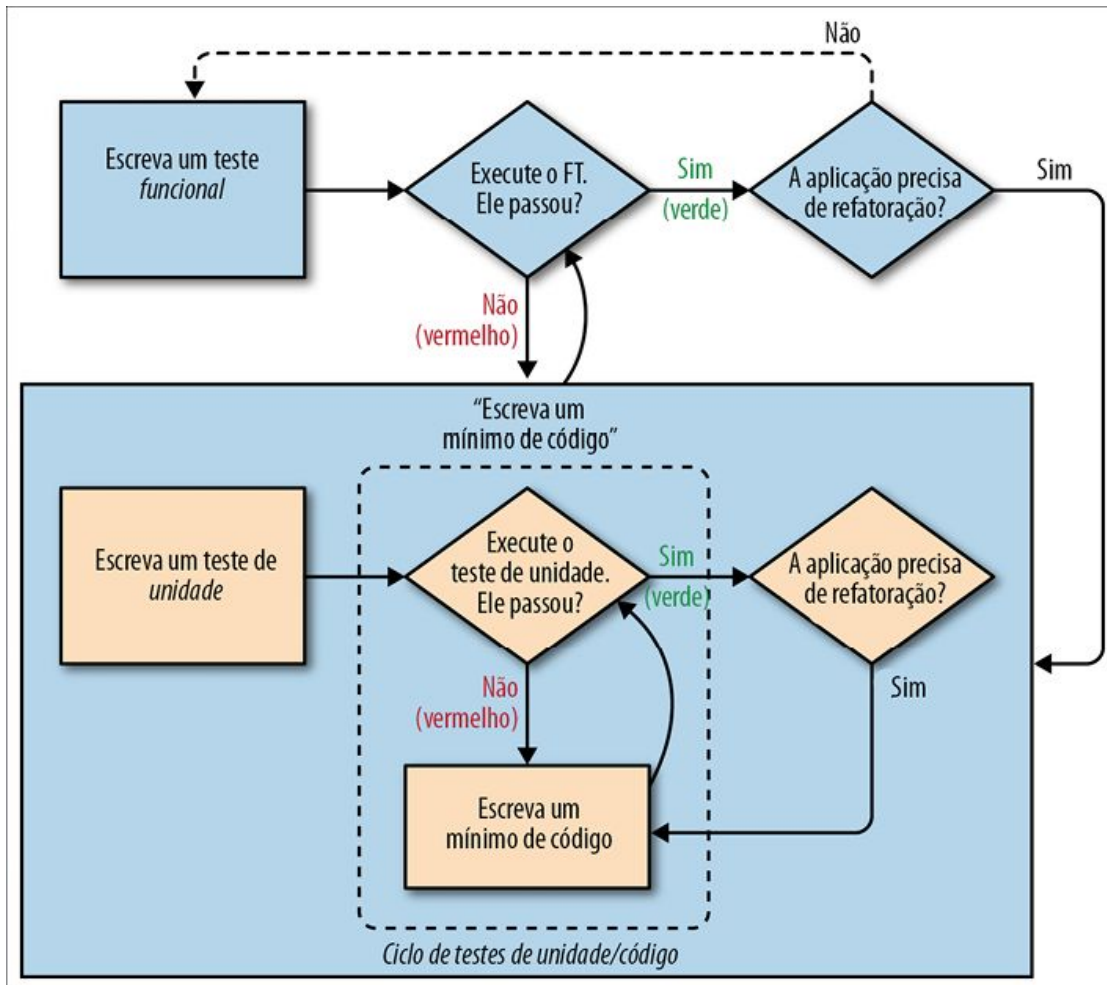


Figura 7.1. – O processo de TDD com testes funcionais e testes de unidade.

Garantindo que teremos um teste de regressão

Vamos traduzir nosso rascunho em um novo método de teste funcional, o qual introduzirá um segundo usuário e verificará se sua lista de tarefas está separada da lista de Edith.

Começaremos de modo muito semelhante ao primeiro teste. Edith adiciona um primeiro item a fim de criar uma lista de tarefas, mas introduziremos nossa primeira nova asserção – a lista de Edith deve ter o próprio URL exclusivo:

`functional_tests/tests.py (ch07I005)`

```
def test_can_start_a_list_for_one_user(self):
```

```
# Edith ouviu falar de uma nova aplicação online interessante para lista
# de tarefas. Ela decide
[...]
# A página é atualizada novamente e agora mostra os dois itens em sua lista
self.wait_for_row_in_list_table('2: Use peacock feathers to make a fly')
self.wait_for_row_in_list_table('1: Buy peacock feathers')

# Satisfeita, ela volta a dormir
```

```
def test_multiple_users_can_start_lists_at_different_urls(self):
    # Edith inicia uma nova lista de tarefas
    self.browser.get(self.live_server_url)
    inputbox = self.browser.find_element_by_id('id_new_item')
    inputbox.send_keys('Buy peacock feathers')
    inputbox.send_keys(Keys.ENTER)
    self.wait_for_row_in_list_table('1: Buy peacock feathers')

    # Ela percebe que sua lista tem um URL único
    edith_list_url = self.browser.current_url
    self.assertRegex(edith_list_url, '/lists/.+') ❶
```

- ❶ `assertRegex` é uma função auxiliar de `unittest` que verifica se uma string corresponde a uma expressão regular. Nós a usamos para verificar se nosso novo design em estilo REST foi implementado. Mais informações podem ser encontradas na documentação do `unittest` (<http://docs.python.org/3/library/unittest.html>)

A seguir suponha que outros usuários cheguem. Queremos verificar se eles não verão nenhum dos itens de Edith quando acessarem a página inicial, e se obtêm os próprios URLs exclusivos para suas listas:

functional_tests/tests.py (ch07I006)

```
[...]
self.assertRegex(edith_list_url, '/lists/.+') ❶

# Agora um novo usuário, Francis, chega ao site.

## Usamos uma nova sessão de navegador para garantir que nenhuma
```

informação

```
## de Edith está vindo de cookies etc
self.browser.quit()
self.browser = webdriver.Firefox()

# Francis acessa a página inicial. Não há nenhum sinal da lista de Edith
self.browser.get(self.live_server_url)
page_text = self.browser.find_element_by_tag_name('body').text
self.assertNotIn('Buy peacock feathers', page_text)
self.assertNotIn('make a fly', page_text)

# Francis inicia uma nova lista inserindo um item novo. Ele
# é menos interessante que Edith...
inputbox = self.browser.find_element_by_id('id_new_item')
inputbox.send_keys('Buy milk')
inputbox.send_keys(Keys.ENTER)
self.wait_for_row_in_list_table('1: Buy milk')

# Francis obtém seu próprio URL exclusivo
francis_list_url = self.browser.current_url
self.assertRegex(francis_list_url, '/lists/.+')
self.assertNotEqual(francis_list_url, edith_list_url)

# Novamente, não há nenhum sinal da lista de Edith
page_text = self.browser.find_element_by_tag_name('body').text
self.assertNotIn('Buy peacock feathers', page_text)
self.assertIn('Buy milk', page_text)

# Satisfeitos, ambos voltam a dormir
```

- ① Estou usando a convenção de dois sinais de suspenso (##) para representar “metacomentários” – comentários sobre *como* o teste está funcionando e por quê – desse modo, podemos distingui-los dos comentários usuais em FTs, que explicam a História do Usuário. São uma mensagem para nós mesmos no futuro, pois poderemos estar nos perguntando por que cargas d’água estamos saindo do navegador e iniciando um novo...

Afora isso, o novo teste é razoavelmente autoexplicativo. Vamos ver como nos saímos quando executamos os nossos FTs:

```
$ python manage.py test functional_tests
```

```
[...]
```

```
.F
```

```
=====
```

```
FAIL: test_multiple_users_can_start_lists_at_different_urls  
(functional_tests.tests.NewVisitorTest)
```

```
-----  
Traceback (most recent call last):
```

```
  File "../superlists/functional_tests/tests.py", line 83, in  
test_multiple_users_can_start_lists_at_different_urls
```

```
    self.assertRegex(edith_list_url, '/lists/.')
```

```
AssertionError: Regex didn't match: '/lists/.' not found in  
'http://localhost:8081/'
```

```
-----  
Ran 2 tests in 5.786s
```

```
FAILED (failures=1)
```

Foi bom, nosso primeiro teste continua passando e o segundo falha no ponto em que seria esperado. Vamos fazer um commit e então prosseguir construindo alguns modelos e views novos:

```
$ git commit -a
```

Iterando em direção ao novo design

Todo empolgado com nosso novo design, tive um extraordinário impulso de mergulhar de cabeça nesse ponto e começar a alterar *models.py*, o que teria causado falhas em metade dos testes de unidade, e então alterar quase todas as linhas de código, tudo de uma só vez. É um impulso natural, e o TDD, como disciplina, está em luta constante contra isso. Obedeça ao Testing Goat, e não ao Refactoring Cat! Não precisamos implementar nosso novo design reluzente em um único Big Bang. Vamos fazer pequenas alterações que nos levem de um estado funcional a outro, com o nosso design gentilmente nos orientando em cada etapa.

Há quatro itens em nossa lista de tarefas. O FT, com seu Regexp

didn't match, está nos informando que o segundo item – dar às listas o próprio URL e identificador – é aquele em que devemos trabalhar em seguida. Vamos fazer uma tentativa de corrigi-lo, e apenas isso.

O URL vem do redirecionamento após o POST. Em *lists/tests.py*, localize `test_redirects_after_POST` e mude o local esperado para o redirecionamento:

lists/tests.py

```
self.assertEqual(response.status_code, 302)
self.assertEqual(response['location'], '/lists/the-only-list-in-the-world/')
```

Isso parece um pouco estranho? Obviamente */lists/the-only-list-in-the-world* não é um URL que aparecerá no design final de nossa aplicação. No entanto, nós nos comprometemos a fazer uma alteração de cada vez. Enquanto nossa aplicação aceitar apenas uma lista, esse é o único URL que faz sentido. Continuamos avançando, pois teremos URLs diferentes para a nossa lista e a página inicial, que é um passo no caminho para um design mais RESTful (com capacidade para REST). Mais tarde, quando tivermos várias listas, será mais fácil fazer a mudança.



Outra forma de pensar é encarar essa abordagem como uma técnica de resolução de problemas: nosso novo design de URL atualmente não está implementado, portanto não funciona para zero item. Em última análise, queremos resolver para n itens, porém resolver para um item é um bom passo no caminho.

A execução dos testes de unidade resulta em uma falha esperada:

```
$ python manage.py test lists
[...]
AssertionError: '/' != '/lists/the-only-list-in-the-world/'
```

Podemos adaptar nossa view `home_page` em *lists/views.py*:

lists/views.py

```
def home_page(request):
    if request.method == 'POST':
```

```
Item.objects.create(text=request.POST['item_text'])
return redirect('/lists/the-only-list-in-the-world/')
```

```
items = Item.objects.all()
return render(request, 'home.html', {'items': items})
```

É claro que isso, sem dúvida, causará falhas nos testes funcionais, pois ainda não há um URL como esse em nosso site. Com certeza, se você executar os testes, verá que eles falharão logo após a tentativa de submeter o primeiro item, informando que a tabela com a lista não pode ser encontrada; isso ocorre porque o URL */the-only-list-in-the-world/* ainda não existe!

```
File ".../superlists/functional_tests/tests.py", line 57, in
test_can_start_a_list_for_one_user
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate
element: [id="id_list_table"]
```

```
[...]
```

```
File ".../superlists/functional_tests/tests.py", line 79, in
test_multiple_users_can_start_lists_at_different_urls
self.wait_for_row_in_list_table('1: Buy peacock feathers')
```

```
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate
element: [id="id_list_table"]
```

Não só o nosso primeiro teste está falhando, mas também o antigo. Isso nos diz que introduzimos uma *regressão*. Vamos tentar voltar para um estado funcional o mais rápido possível construindo um URL para a nossa única lista.

Dando um primeiro passo autocontido: um novo URL

Abra *lists/tests.py* e adicione uma nova classe de teste chamada *ListViewTest*. Então copie o método chamado *test_displays_all_list_items*

de `HomePageTest` para a nossa nova classe, renomeie e faça uma pequena adaptação nesse método:

lists/tests.py (ch07I009)

```
class ListViewTest(TestCase):

    def test_displays_all_items(self):
        Item.objects.create(text='itemey 1')
        Item.objects.create(text='itemey 2')

        response = self.client.get('/lists/the-only-list-in-the-world/')

        self.assertContains(response, 'itemey 1') ❶
        self.assertContains(response, 'itemey 2') ❶
```

- ❶ Eis um novo método auxiliar: em vez de usar a dança levemente irritante com `assertIn/response.content.decode()`, o Django oferece o método `assertContains`, que sabe como lidar com respostas e os bytes de seu conteúdo.

Vamos experimentar executar esse teste agora:

```
self.assertContains(response, 'itemey 1')
[...]
AssertionError: 404 != 200 : Couldn't retrieve content: Response code was 404
```

Esse é um bom efeito colateral do uso de `assertContains`: ele nos informa de imediato que o teste está falhando porque o nosso novo URL ainda não existe, e devolve 404.

Um novo URL

Nosso URL de lista singleton ainda não existe. Corrigiremos isso em `superlists/urls.py`.



Preste atenção nas barras finais em URLs, tanto aqui nos testes quanto em `urls.py`. Elas são causas comuns de bugs.

`superlists/urls.py`

```
urlpatterns = [  
    url(r'^$', views.home_page, name='home'),  
    url(r'^lists/the-only-list-in-the-world/$', views.view_list, name='view_list'),  
]
```

Se os testes forem executados novamente, obteremos:

```
AttributeError: module 'lists.views' has no attribute 'view_list'
```

Uma nova função de view

Elegantemente autoexplicativo. Vamos criar uma função de view dummy em *lists/views.py*:

lists/views.py

```
def view_list(request):  
    pass
```

Eis o resultado agora:

```
ValueError: The view lists.views.view_list didn't return an HttpResponse  
object. It returned None instead.
```

```
[...]  
FAILED (errors=1)
```

Restou apenas uma falha, e ela nos orienta para a direção correta. Vamos copiar as duas últimas linhas da view *home_page* e ver se elas são suficientes para resolver o problema:

lists/views.py

```
def view_list(request):  
    items = Item.objects.all()  
    return render(request, 'home.html', {'items': items})
```

Execute os testes de unidade novamente; eles devem passar:

```
Ran 7 tests in 0.016s  
OK
```

Vamos tentar os FTs de novo e ver o que eles nos dizem:

```
FAIL: test_can_start_a_list_for_one_user  
[...]  
File ".../superlists/functional_tests/tests.py", line 67, in
```

```
test_can_start_a_list_for_one_user
```

```
[...]
```

```
AssertionError: '2: Use peacock feathers to make a fly' not found in ['1: Buy peacock feathers']
```

```
FAIL: test_multiple_users_can_start_lists_at_different_urls
```

```
[...]
```

```
AssertionError: 'Buy peacock feathers' unexpectedly found in 'Your To-Do list\n1: Buy peacock feathers'
```

```
[...]
```

Os dois avançaram um pouco mais em relação à execução anterior, porém continuam falhando. Seria bom retornar a um estado funcional e fazer aquele primeiro teste passar novamente. O que ele está tentando nos dizer?

A falha está ocorrendo quando tentamos adicionar o segundo item. Temos que colocar nosso chapéu de depurador aqui. Sabemos que a página inicial está funcionando porque o teste chegou até a linha 67 no FT, portanto, no mínimo, adicionamos um primeiro item. Além disso, nossos testes de unidade estão todos passando, portanto temos certeza absoluta de que os URLs e as views estão fazendo o que deveriam – a página inicial exige o template correto e é capaz de tratar requisições POST, e a view *only-list-in-the-world* sabe como exibir todos os itens... mas não sabe como tratar requisições POST. Ah, isso nos dá uma pista.

Uma segunda pista é a regra geral segundo a qual quando todos os testes de unidade estiverem passando, mas os testes funcionais não, geralmente eles estão apontando para um problema que não foi coberto pelos testes de unidade e, em nosso caso, com frequência é um problema de template.

A resposta é que o nosso formulário de entrada *home.html* atualmente não especifica um URL explícito para o POST:

lists/templates/home.html

```
<form method="POST">
```

Por padrão, o navegador envia os dados de POST de volta ao

mesmo URL no qual ele está no momento. Quando estamos na página inicial, isso funciona bem, porém, quando estamos na *página only-list-in-the-world*, não funciona.

Agora podemos mergulhar de cabeça e adicionar o tratamento da requisição POST em nossa nova view, mas isso envolveria escrever um punhado de novos testes e o código, e, a essa altura, gostaríamos de retornar a um estado funcional o mais rápido possível. Na verdade, o que podemos fazer de mais rápido para corrigir a situação é simplesmente usar a view existente da página inicial, que já funciona, para todas as requisições POST:

lists/templates/home.html

```
<form method="POST" action="/">
```

Experimente fazer isso, e veremos nossos FTs voltarem para um estado mais satisfatório:

```
FAIL: test_multiple_users_can_start_lists_at_different_urls
```

```
[...]
```

```
AssertionError: 'Buy peacock feathers' unexpectedly found in 'Your To-Do
```

```
list\n1: Buy peacock feathers'
```

```
Ran 2 tests in 8.541s
```

```
FAILED (failures=1)
```

Nosso teste original passa novamente, portanto sabemos que voltamos a um estado funcional. A nova funcionalidade pode não estar correta ainda, mas, no mínimo, a parte antiga funciona conforme costumava ocorrer.

Verde? Refatorar

É hora de um pouco de arrumação.

Na dança do *Vermelho/Verde/Refatorar*, chegamos ao verde, portanto devemos observar o que é que precisa ser refatorado. Temos agora duas views: uma para a página inicial e outra para uma lista individual. Atualmente ambas usam o mesmo template e estão lhe passando todos os itens de lista que estão no banco de

dados no momento. Se observarmos os métodos de nossos testes de unidade, poderemos ver alguns pontos que provavelmente vamos querer mudar:

```
$ grep -E "class|def" lists/tests.py
class HomePageTest(TestCase):
    def test_uses_home_template(self):
    def test_displays_all_list_items(self):
    def test_can_save_a_POST_request(self):
    def test_redirects_after_POST(self):
    def test_only_saves_items_when_necessary(self):
class ListViewTest(TestCase):
    def test_displays_all_items(self):
class ItemModelTest(TestCase):
    def test_saving_and_retrieving_items(self):
```

Sem dúvida, podemos apagar o método `test_displays_all_list_items` de `HomePageTest`; ele não é mais necessário. Se você executar `manage.py test lists` agora, o comando deverá informar que foram executados 6 testes, e não 7:

```
Ran 6 tests in 0.016s
OK
```

A seguir, como realmente não precisamos mais do template da página inicial para exibir todos os itens de lista, ele deverá simplesmente mostrar uma única caixa de entrada convidando você a iniciar uma nova lista.

Outro pequeno passo: um template separado para visualizar listas

Como a view da página inicial e da lista agora são páginas bem distintas, elas deverão usar templates HTML diferentes; *home.html* pode ter a caixa de entrada única, enquanto um novo template, *list.html*, pode cuidar da apresentação da tabela com os itens existentes.

Vamos adicionar um novo teste a fim de verificar se um template diferente está sendo usado:

lists/tests.py

```
class ListViewTest(TestCase):

    def test_uses_list_template(self):
        response = self.client.get('/lists/the-only-list-in-the-world/')
        self.assertTemplateUsed(response, 'list.html')

    def test_displays_all_items(self):
        [...]
```

`assertTemplateUsed` é uma das funções mais úteis que o Django Test Client nos oferece. Vamos ver o que ela diz:

```
AssertionError: False is not true : Template 'list.html' was not a template
used to render the response. Actual template(s) used: home.html
```

Ótimo! Vamos alterar a view:

lists/views.py

```
def view_list(request):
    items = Item.objects.all()
    return render(request, 'list.html', {'items': items})
```

Obviamente, porém, esse template ainda não existe. Se os testes de unidade forem executados, veremos o seguinte:

```
django.template.exceptions.TemplateDoesNotExist: list.html
```

Vamos criar um arquivo novo em *lists/templates/list.html*:

```
$ touch lists/templates/list.html
```

É um template em branco, que nos dá o erro a seguir – é bom saber que os testes existem para garantir que vamos preenchê-lo:

```
AssertionError: False is not true : Couldn't find 'itemey 1' in response
```

O template para uma lista individual reutilizará boa parte do que temos atualmente em *home.html*, portanto podemos começar simplesmente o copiando:

```
$ cp lists/templates/home.html lists/templates/list.html
```

Isso faz com que os testes voltem a passar (verde). Vamos agora

fazer um pouco mais de arrumação (refatoração). Dissemos que a página inicial não precisa listar itens; ela só precisa do campo de entrada para uma nova lista, portanto podemos remover algumas linhas de *lists/templates/home.html*, e talvez ajustar levemente o h1 para dizer “Start a new To-Do list” (Iniciar uma nova lista de tarefas):

lists/templates/home.html

```
<body>
  <h1>Start a new To-Do list</h1>
  <form method="POST">
    <input name="item_text" id="id_new_item" placeholder="Enter a to-do item" />
    {% csrf_token %}
  </form>
</body>
```

Executamos novamente os testes de unidade para conferir se isso não causou falhas em outros lugares – bom...

Na verdade, não há necessidade de passar todos os itens para o template *home.html* em nossa view *home_page*, portanto podemos simplificar:

lists/views.py

```
def home_page(request):
    if request.method == 'POST':
        Item.objects.create(text=request.POST['item_text'])
        return redirect('/lists/the-only-list-in-the-world/')
    return render(request, 'home.html')
```

Vamos executar os testes de unidade mais uma vez; eles continuam passando. É hora de executar os testes funcionais:

```
AssertionError: '1: Buy milk' not found in ['1: Buy peacock feathers',
'2: Buy milk']
```

Nada mal! Nosso teste de regressão (o primeiro FT) está passando, e nosso novo teste agora conseguiu avançar um pouco mais – o teste nos informa que Francis não está obtendo a página para a própria lista (porque ele continua vendo alguns itens da lista de Edith).

Pode parecer que não avançamos muito, pois, do ponto de vista funcional, o site continua se comportando quase exatamente como fazia quando iniciamos o capítulo, mas isso, na verdade, é um progresso. Começamos no caminho para o nosso novo design e implementamos uma série de passos *sem piorar nada em relação ao que havia antes*. Vamos fazer commit de nosso progresso até agora:

```
$ git status # deve mostrar 4 arquivos alterados e 1 arquivo novo, list.html
$ git add lists/templates/list.html
$ git diff # deve mostrar que simplificamos home.html,
    # movemos um teste para uma nova classe em lists/tests.py,
    # adicionamos uma nova view em views.py, e simplificamos home_page,
    # além de termos feito uma adição em urls.py
$ git commit -a # adicione uma mensagem resumindo o que foi dito antes,
talvez
    # algo como "new URL, view and template to display lists"
    # (URL, view e template novos para exibir listas)
```

Um terceiro passo pequeno: um URL para adicionar itens de lista

Qual é a situação em relação à nossa própria lista de tarefas?

- **Ajustar o modelo para que os itens sejam associados a listas distintas**
- **Adicionar URLs exclusivos para cada lista...**
- **Adicionar um URL para criar uma nova lista via POST**
- **Adicionar URLs para acrescentar um novo item em uma lista existente via POST**

De *certo modo*, fizemos progresso no segundo item, apesar de ainda haver apenas uma lista no mundo. O primeiro item é um pouco assustador. Podemos fazer algo a respeito dos itens 3 e 4?

Vamos criar um novo URL para adicionar novos itens de lista. No mínimo isso simplificará a view da página inicial.

Uma classe de testes para a criação de uma nova lista

Abra `lists/tests.py` e *move* os métodos `test_can_save_a_POST_request` e `test_redirects_after_POST` para uma nova classe; em seguida, altere o URL para o qual eles fazem POST:

`lists/tests.py` (ch07I021-1)

```
class NewListTest(TestCase):

    def test_can_save_a_POST_request(self):
        self.client.post('/lists/new', data={'item_text': 'A new list item'})
        self.assertEqual(Item.objects.count(), 1)
        new_item = Item.objects.first()
        self.assertEqual(new_item.text, 'A new list item')

    def test_redirects_after_POST(self):
        response = self.client.post('/lists/new', data={'item_text': 'A new list item'})
        self.assertEqual(response.status_code, 302)
        self.assertEqual(response['location'], '/lists/the-only-list-in-the-world/')
```



Esse é outro local em que você eventualmente deve prestar atenção nas barras finais. É `/new`, sem barra no final. Segundo a convenção que estou usando, URLs sem uma barra no final são URLs de “ação” que modificam o banco de dados.

Enquanto estamos discutindo esse assunto, vamos conhecer um novo método do Django Test Client, `assertRedirects`:

`lists/tests.py` (ch07I021-2)

```
def test_redirects_after_POST(self):
    response = self.client.post('/lists/new', data={'item_text': 'A new list item'})
    self.assertRedirects(response, '/lists/the-only-list-in-the-world/')
```

Não há muito o que dizer sobre esse método, mas ele simplesmente substitui duas asserções por uma só...

Experimente executar o código anterior:

```
self.assertEqual(Item.objects.count(), 1)
AssertionError: 0 != 1
[...]
self.assertRedirects(response, '/lists/the-only-list-in-the-world/')
[...]
AssertionError: 404 != 302 : Response didn't redirect as expected: Response
code was 404 (expected 302)
```

A primeira falha nos informa que não estamos salvando um novo item no banco de dados, e a segunda diz que, em vez de devolver um redirecionamento 302, nossa view está devolvendo um 404. Isso ocorre porque não construímos um URL para `/lists/new`, de modo que `client.post` está simplesmente obtendo uma resposta de “não encontrado”.



Você se lembra de como separamos isso em dois testes antes? Se tivéssemos apenas um teste que verificasse tanto o salvamento dos dados quanto o redirecionamento, a falha ocorreria em `0 != 1`, o que teria sido muito mais difícil de depurar. Pergunte-me como eu sei disso.

Um URL e uma view para a criação de uma nova lista

Vamos construir agora o nosso novo URL:

`superlists/urls.py`

```
urlpatterns = [
    url(r'^$', views.home_page, name='home'),
    url(r'^lists/new$', views.new_list, name='new_list'),
    url(r'^lists/the-only-list-in-the-world/$', views.view_list, name='view_list'),
]
```

A seguir, obtivemos um no attribute 'new_list', portanto vamos corrigir isso em `lists/views.py`:

`lists/views.py (ch07l023-1)`

```
def new_list(request):
    pass
```

Em seguida, obtivemos “The view lists.views.new_list didn’t return an HttpResponse object” (A view lists.views.new_list não devolveu um objeto HttpResponse). (Isso está se tornando bem familiar!) Poderíamos devolver um HttpResponse bruto, mas, como sabemos que precisaremos de um redirecionamento, vamos emprestar uma linha de home_page:

lists/views.py (ch07l023-2)

```
def new_list(request):  
    return redirect('/lists/the-only-list-in-the-world/')
```

Eis o resultado:

```
self.assertEqual(Item.objects.count(), 1)  
AssertionError: 0 != 1
```

Parece razoavelmente simples. Vamos emprestar outra linha de home_page:

lists/views.py (ch07l023-3)

```
def new_list(request):  
    Item.objects.create(text=request.POST['item_text'])  
    return redirect('/lists/the-only-list-in-the-world/')
```

Agora todos os testes passam:

```
Ran 7 tests in 0.030s
```

OK

Além disso, os FTs me mostram que estou de volta ao estado funcional:

```
[...]  
AssertionError: '1: Buy milk' not found in ['1: Buy peacock feathers', '2: Buy milk']  
Ran 2 tests in 8.972s  
FAILED (failures=1)
```

Removendo código e testes não redundantes

Tudo parece bem. Como nossas novas views agora estão fazendo a maior parte do trabalho que home_page costumava fazer, devemos

ser capazes de simplificá-la bastante. Podemos remover toda a seção `if request.method == 'POST'`, por exemplo?

lists/views.py

```
def home_page(request):  
    return render(request, 'home.html')
```

Sim!

OK

Aproveitando que estamos fazendo isso, podemos remover o teste não redundante `test_only_saves_items_when_necessary` também!

Não é uma sensação boa? As funções de view estão bem mais simples. Vamos executar novamente os testes por garantia...

```
Ran 6 tests in 0.016s
```

OK

E quanto aos FTs?

Uma regressão! Apontando nossos formulários para o novo URL

Opa:

```
ERROR: test_can_start_a_list_for_one_user
```

```
[...]
```

```
File ".../superlists/functional_tests/tests.py", line 57, in  
test_can_start_a_list_for_one_user
```

```
    self.wait_for_row_in_list_table('1: Buy peacock feathers')
```

```
File ".../superlists/functional_tests/tests.py", line 23, in  
wait_for_row_in_list_table
```

```
    table = self.browser.find_element_by_id('id_list_table')
```

```
selenium.common.exceptions.NoSuchElementException: Message: Unable to  
locate
```

```
element: [id="id_list_table"]
```

```
ERROR: test_multiple_users_can_start_lists_at_different_urls
```

```
[...]
```

```
File ".../superlists/functional_tests/tests.py", line 79, in  
test_multiple_users_can_start_lists_at_different_urls
```

```
self.wait_for_row_in_list_table('1: Buy peacock feathers')
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate
element: [id="id_list_table"]
[...]
```

```
Ran 2 tests in 11.592s
FAILED (errors=2)
```

O erro ocorre porque nossos formulários continuam apontando para o URL antigo. Tanto em *home.html* quanto em *lists.html*, vamos mudá-los para:

lists/templates/home.html, lists/templates/list.html

```
<form method="POST" action="/lists/new">
```

Isso deve nos levar de volta a um estado funcional novamente:

```
AssertionError: '1: Buy milk' not found in ['1: Buy peacock feathers', '2: Buy
milk']
[...]
FAILED (failures=1)
```

Esse é outro commit bem autocontido, pois fizemos um punhado de alterações em nossos URLs, nossa *views.py* está com um aspecto muito melhor e mais organizado e temos certeza de que a aplicação continua funcionando como antes. Estamos ficando bons nessa conversa de estado-funcional-para-estado-funcional!

```
$ git status # 5 arquivos alterados
$ git diff # URLs para formulários duas vezes, código movido em
# views + testes, novo URL
$ git commit -a
```

Assim, podemos riscar um item de nossa lista de tarefas:

- **Ajustar o modelo para que os itens sejam associados a listas distintas**
- **Adicionar URLs exclusivos para cada lista**
- **~~Adicionar um URL para criar uma nova lista via POST~~**
- **Adicionar URLs para acrescentar um novo item em uma**

lista existente via POST

Encarando a situação de frente: ajustando nossos modelos

Chega de pôr a casa em ordem com nossos URLs. É hora de encarar a situação de frente e alterar nossos modelos. Vamos ajustar o teste de unidade do modelo. Somente para variar, apresentarei as mudanças na forma de um diff:

lists/tests.py

```
@@ -1,5 +1,5 @@
from django.test import TestCase
-from lists.models import Item
+from lists.models import Item, List

class HomePageTest(TestCase):
@@ -44,22 +44,32 @@ class ListViewTest(TestCase):

-class ItemModelTest(TestCase):
+class ListAndItemModelsTest(TestCase):

    def test_saving_and_retrieving_items(self):
+ list_ = List()
+ list_.save()
+
    first_item = Item()
    first_item.text = 'The first (ever) list item'
+ first_item.list = list_
    first_item.save()

    second_item = Item()
    second_item.text = 'Item the second'
+ second_item.list = list_
    second_item.save()
```



```

+ saved_list = List.objects.first()
+ self.assertEqual(saved_list, list_)
+
    saved_items = Item.objects.all()
    self.assertEqual(saved_items.count(), 2)

    first_saved_item = saved_items[0]
    second_saved_item = saved_items[1]
    self.assertEqual(first_saved_item.text, 'The first (ever) list item')
+ self.assertEqual(first_saved_item.list, list_)
    self.assertEqual(second_saved_item.text, 'Item the second')
+ self.assertEqual(second_saved_item.list, list_)

```

Criamos um novo objeto List e, em seguida, atribuímos cada item a ele como sua propriedade `.list`. Verificamos se a lista é devidamente salva e, também, se os dois itens salvaram seu relacionamento com a lista. Além disso, você perceberá que podemos comparar objetos de lista um com o outro diretamente (`saved_list` e `list_`) – internamente, eles se compararão verificando se sua chave primária (o atributo `.id`) é igual.



Estou usando o nome de variável `list_` para evitar “encobrir” a função embutida `list` de Python. Não é elegante, mas todas as demais opções que tentei eram igualmente deselegantes ou piores (`my_list`, `the_list`, `list1`, `listey...`).

É hora de outro ciclo de testes de unidade/código.

Nas duas primeiras iterações, em vez de mostrar explicitamente o código que deve ser inserido entre cada execução de testes, mostrarei as mensagens de erro esperadas com a execução dos testes. Deixarei que você descubra por conta própria qual deve ser cada alteração mínima de código.



Você precisa de uma dica? Retorne e dê uma olhada nos passos que executamos para introduzir o modelo Item no antepenúltimo capítulo.

Este deverá ser o seu primeiro erro:

```
ImportError: cannot import name 'List'
```

Corrija-o, e então você deverá ver:

```
AttributeError: 'List' object has no attribute 'save'
```

Em seguida, você deverá ver:

```
django.db.utils.OperationalError: no such table: lists_list
```

Então executamos um makemigrations:

```
$ python manage.py makemigrations
```

```
Migrations for 'lists':
```

```
lists/migrations/0003_list.py
```

```
- Create model List
```

E então você deverá ver:

```
self.assertEqual(first_saved_item.list, list_)
```

```
AttributeError: 'Item' object has no attribute 'list'
```

Um relacionamento de chave estrangeira

Como damos um atributo de lista ao nosso Item? Vamos apenas tentar fazê-lo ingenuamente, como o atributo text (e eis a sua chance de ver se sua solução até agora, a propósito, se parece com a minha):

lists/models.py

```
from django.db import models
```

```
class List(models.Model):
```

```
    pass
```

```
class Item(models.Model):
```

```
    text = models.TextField(default="")
```

```
    list = models.TextField(default="")
```

Como sempre, os testes nos informam que precisamos de uma migração:

```
$ python manage.py test lists
```

```
[...]
```

```
django.db.utils.OperationalError: no such column: lists_item.list
```

```
$ python manage.py makemigrations
```

```
Migrations for 'lists':
```

```
lists/migrations/0004_item_list.py
```

```
- Add field list to item
```

Vamos ver o que isso nos dá:

```
AssertionError: 'List object' != <List: List object>
```

Ainda não chegamos lá. Observe atentamente cada lado de !=. O Django salvou apenas a representação em string do objeto List. Para salvar o relacionamento com o objeto propriamente dito, informamos ao Django o relacionamento entre as duas classes usando uma ForeignKey:

lists/models.py

```
from django.db import models
```

```
class List(models.Model):
```

```
    pass
```

```
class Item(models.Model):
```

```
    text = models.TextField(default="")
```

```
    list = models.ForeignKey(List, default=None)
```

Isso exigirá uma migração também. Como a última migração acabou se mostrando um caminho errado, vamos apagá-la e substituí-la por uma nova:

```
$ rm lists/migrations/0004_item_list.py
```

```
$ python manage.py makemigrations
```

```
Migrations for 'lists':
```

```
lists/migrations/0004_item_list.py
```

```
- Add field list to item
```



Apagar migrações é perigoso. Precisaremos fazer isso ocasionalmente, pois nem sempre teremos o código correto para nossos modelos na primeira tentativa. Todavia, se você apagar uma migração que já tenha sido aplicada em um banco de dados em algum lugar, o Django ficará confuso sobre o estado em que o banco de dados se encontra e sobre como aplicar futuras migrações. Faça isso somente quando tiver certeza de que a migração não foi usada. Uma boa regra geral é que você jamais deve apagar ou modificar uma migração cujo commit já tenha sido efetuado em seu sistema de controle de versões.

Adaptando o restante do mundo aos novos modelos

Voltando aos nossos testes, o que acontecerá agora?

```
$ python manage.py test lists
[...]
ERROR: test_displays_all_items (lists.tests.ListViewTest)
django.db.utils.IntegrityError: NOT NULL constraint failed: lists_item.list_id
[...]
ERROR: test_redirects_after_POST (lists.tests.NewListTest)
django.db.utils.IntegrityError: NOT NULL constraint failed: lists_item.list_id
[...]
ERROR: test_can_save_a_POST_request (lists.tests.NewListTest)
django.db.utils.IntegrityError: NOT NULL constraint failed: lists_item.list_id

Ran 6 tests in 0.021s

FAILED (errors=3)
```

Oh, céus!

Há algumas notícias boas. Embora seja difícil ver, os testes de nosso modelo estão passando. No entanto, três dos testes de nossas views lamentavelmente estão falhando.

Isso se deve ao novo relacionamento que introduzimos entre Items e Lists, o qual exige que cada item tenha uma lista-pai, para o qual nossos testes e código antigos não estão preparados.

É exatamente por isso que temos testes! Vamos fazê-los funcionar

novamente. O mais fácil é `ListViewTest`; basta criar uma lista-pai para os nossos dois itens de teste:

lists/tests.py (ch07I031)

```
class ListViewTest(TestCase):

    def test_displays_all_items(self):
        list_ = List.objects.create()
        Item.objects.create(text='itemey 1', list=list_)
        Item.objects.create(text='itemey 2', list=list_)
```

Isso faz com que restem dois testes em falha, ambos tentando fazer um POST para a nossa view `new_list`. Ao decifrar os tracebacks utilizando nossa técnica usual, trabalhando de trás para frente, do erro até a linha do código de teste, escondida em algum ponto, está a linha de nosso próprio código que provocou a falha:

```
File "/.../superlists/lists/views.py", line 9, in new_list
    Item.objects.create(text=request.POST['item_text'])
```

É quando tentamos criar um item sem uma lista-pai. Portanto, faremos uma mudança semelhante na view:

lists/views.py

```
from lists.models import Item, List
[...]
def new_list(request):
    list_ = List.objects.create()
    Item.objects.create(text=request.POST['item_text'], list=list_)
    return redirect('/lists/the-only-list-in-the-world/')
```

Com isso, nossos testes voltam a passar novamente:

```
Ran 6 tests in 0.030s
```

OK

Você está se contorcendo internamente a essa altura? *Argh! Isso parece tão errado; criamos uma nova lista para a submissão de cada novo item, e ainda estamos exibindo todos os itens como se pertencessem à mesma lista!* Eu sei; sinto o mesmo. A abordagem

passo a passo, na qual vamos de um código funcional a outro, é contraintuitiva. Sempre tenho vontade de simplesmente mergulhar de cabeça e tentar corrigir tudo em uma só empreitada, em vez de ir de um estado semiterminado insólito para outro. Lembre-se, porém, do Testing Goat! Quando você estiver no alto de uma montanha, vai querer pensar cuidadosamente no lugar em que colocará cada pé, e dará um passo de cada vez, verificando, em cada etapa, se o local em que pisou não fará você cair de um penhasco.

Então, somente para nos garantir de que tudo funcionou, vamos executar novamente o FT:

```
AssertionError: '1: Buy milk' not found in ['1: Buy peacock feathers', '2: Buy milk']  
[...]
```

Sem dúvida, o teste nos leva até o ponto em que estávamos antes. Não causamos problema em nada e fizemos uma alteração no banco de dados. É algo com que devemos nos sentir satisfeitos! Vamos fazer um commit:

```
$ git status # 3 arquivos alterados, mais 2 migrações  
$ git add lists  
$ git diff --staged  
$ git commit
```

Podemos riscar mais um item da lista de tarefas:

- ~~Ajustar o modelo para que os itens sejam associados a listas distintas~~
- Adicionar URLs exclusivos para cada lista
- ~~Adicionar um URL para criar uma nova lista via POST~~
- Adicionar URLs para acrescentar um novo item em uma lista existente via POST

Cada lista deve ter o próprio URL

O que devemos usar como identificador único para nossas listas? Provavelmente, o mais simples, por enquanto, é usar o campo id do

banco de dados, gerado automaticamente. Vamos alterar `ListViewTest` para que os dois testes apontem para novos URLs.

Também modificaremos o teste antigo `test_displays_all_items` e o chamaremos de `test_displays_only_items_for_that_list`, fazendo-o verificar se somente os itens de uma lista específica serão exibidos:

lists/tests.py (ch07I033)

```
class ListViewTest(TestCase):

    def test_uses_list_template(self):
        list_ = List.objects.create()
        response = self.client.get(f'/lists/{list_.id}/')
        self.assertTemplateUsed(response, 'list.html')

    def test_displays_only_items_for_that_list(self):
        correct_list = List.objects.create()
        Item.objects.create(text='itemey 1', list=correct_list)
        Item.objects.create(text='itemey 2', list=correct_list)
        other_list = List.objects.create()
        Item.objects.create(text='other list item 1', list=other_list)
        Item.objects.create(text='other list item 2', list=other_list)

        response = self.client.get(f'/lists/{correct_list.id}/')

        self.assertContains(response, 'itemey 1')
        self.assertContains(response, 'itemey 2')
        self.assertNotContains(response, 'other list item 1')
        self.assertNotContains(response, 'other list item 2')
```



Há mais duas daquelas belas f-strings nessa listagem! Caso elas ainda continuem um pouco misteriosas, dê uma olhada na documentação (https://docs.python.org/3/reference/lexical_analysis.html#f-strings). (Embora, se sua formação em Ciência da Computação for tão ruim quanto a minha, provavelmente você deixará de lado a gramática formal.)

A execução dos testes de unidade resultam em um 404 esperado e em outro erro relacionado:

```
FAIL: test_displays_only_items_for_that_list (lists.tests.ListViewTest)
AssertionError: 404 != 200 : Couldn't retrieve content: Response code was 404
(expected 200)
[...]
FAIL: test_uses_list_template (lists.tests.ListViewTest)
AssertionError: No templates used to render the response
```

Capturando parâmetros de URLs

É hora de aprender a passar parâmetros de URLs para as views:

superlists/urls.py

```
urlpatterns = [
    url(r'^$', views.home_page, name='home'),
    url(r'^lists/new$', views.new_list, name='new_list'),
    url(r'^lists/(.+)$', views.view_list, name='view_list'),
]
```

Adaptamos a expressão regular para o nosso URL de modo a incluir um *grupo de captura*, (.+), que fará a correspondência com qualquer caractere até a próxima /. O texto capturado será passado para a view como argumento.

Em outras palavras, se acessarmos o URL `/lists/1/`, `view_list` receberá um segundo argumento após o argumento `request` usual, que será a string "1". Se acessarmos `/lists/foo/`, teremos `view_list(request, "foo")`.

Contudo, nossa view ainda não espera um argumento! Sem dúvida, isso causa problemas:

```
ERROR: test_displays_only_items_for_that_list (lists.tests.ListViewTest)
[...]
TypeError: view_list() takes 1 positional argument but 2 were given
[...]
ERROR: test_uses_list_template (lists.tests.ListViewTest)
[...]
TypeError: view_list() takes 1 positional argument but 2 were given
[...]
ERROR: test_redirects_after_POST (lists.tests.NewListTest)
[...]
TypeError: view_list() takes 1 positional argument but 2 were given
FAILED (errors=3)
```


Podemos corrigir isso facilmente com um parâmetro dummy em *views.py*:

lists/views.py

```
def view_list(request, list_id):  
    [...]
```

Agora chegamos à nossa falha esperada:

```
FAIL: test_displays_only_items_for_that_list (lists.tests.ListViewTest)  
[...]  
AssertionError: 1 != 0 : Response should not contain 'other list item 1'
```

Vamos fazer nossa view discriminar quais itens ela envia para o template:

lists/views.py

```
def view_list(request, list_id):  
    list_ = List.objects.get(id=list_id)  
    items = Item.objects.filter(list=list_)  
    return render(request, 'list.html', {'items': items})
```

Adaptando new_list para o novo mundo

Opa, agora temos erros em outro teste:

```
ERROR: test_redirects_after_POST (lists.tests.NewListTest)  
ValueError: invalid literal for int() with base 10:  
'the-only-list-in-the-world'
```

Vamos analisar esse teste então, já que ele está reclamando:

lists/tests.py

```
class NewListTest(TestCase):  
    [...]  
    def test_redirects_after_POST(self):  
        response = self.client.post('/lists/new', data={'item_text': 'A new list item'})  
        self.assertRedirects(response, '/lists/the-only-list-in-the-world/')
```

Parece que ele não foi adaptado para o novo mundo de Lists e Items. O teste deve estar informando que essa view faz um redirecionamento para o URL da nova lista específica, que acabou

de ser criada:

lists/tests.py (ch07I036-1)

```
def test_redirects_after_POST(self):
    response = self.client.post('/lists/new', data={'item_text': 'A new list item'})
    new_list = List.objects.first()
    self.assertRedirects(response, f'/lists/{new_list.id}/')
```

Continuamos obtendo um erro de *literal inválido*. Observamos a view propriamente dita e a alteramos de modo que ela faça um redirecionamento para um local válido:

lists/views.py (ch07I036-2)

```
def new_list(request):
    list_ = List.objects.create()
    Item.objects.create(text=request.POST['item_text'], list=list_)
    return redirect(f'/lists/{list_.id}/')
```

Isso nos leva de volta ao estado em que os testes de unidade passam:

```
$ python3 manage.py test lists
```

```
[...]
```

```
.....
```

```
-----
Ran 6 tests in 0.033s
```

```
OK
```

E o que dizer dos testes funcionais? Será que estamos quase lá?

Os testes funcionais detectam outra regressão

Bem, quase:

```
F.
```

```
=====
=====
```

```
FAIL: test_can_start_a_list_for_one_user
(functional_tests.tests.NewVisitorTest)
```

```
-----
Traceback (most recent call last):
```

```
File "../superlists/functional_tests/tests.py", line 67, in
```

```
test_can_start_a_list_for_one_user
    self.wait_for_row_in_list_table('2: Use peacock feathers to make a fly')
[...]
AssertionError: '2: Use peacock feathers to make a fly' not found in ['1: Use
peacock feathers to make a fly']
```

Ran 2 tests in 8.617s

FAILED (failures=1)

Nosso novo teste na verdade está passando e diferentes usuários podem obter listas distintas, mas o teste antigo está nos avisando de uma regressão. Parece que não podemos mais adicionar um segundo item em uma lista. Isso se deve ao nosso hack grosseiro e rápido, no qual criamos uma nova lista para cada uma das submissões de POST. É exatamente por isso que temos testes funcionais!

E isso se correlaciona muito bem com o último item de nossa lista de tarefas:

- **Ajustar o modelo para que os itens sejam associados a listas distintas**
- **Adicionar URLs exclusivos para cada lista**
- **~~Adicionar um URL para criar uma nova lista via POST~~**
- **Adicionar URLs para acrescentar um novo item em uma lista existente via POST**

Mais uma view para tratar a adição de itens em uma lista existente

Precisamos de um URL e de uma view para tratar a adição de um novo item em uma lista existente (*/lists/<list_id>/add_item*). Estamos ficando muito bons nisso agora, então vamos montar um teste rapidamente:

lists/tests.py

```
classNewItemTest(TestCase):

    def test_can_save_a_POST_request_to_an_existing_list(self):
        other_list = List.objects.create()
        correct_list = List.objects.create()

        self.client.post(
            f'/lists/{correct_list.id}/add_item',
            data={'item_text': 'A new item for an existing list'}
        )

        self.assertEqual(Item.objects.count(), 1)
        new_item = Item.objects.first()
        self.assertEqual(new_item.text, 'A new item for an existing list')
        self.assertEqual(new_item.list, correct_list)

    def test_redirects_to_list_view(self):
        other_list = List.objects.create()
        correct_list = List.objects.create()

        response = self.client.post(
            f'/lists/{correct_list.id}/add_item',
            data={'item_text': 'A new item for an existing list'}
        )

        self.assertRedirects(response, f'/lists/{correct_list.id}/')
```



Você está se perguntando sobre `other_list`? De modo um pouco parecido com os testes para visualizar uma lista específica, é importante adicionar itens em uma lista específica. Adicionar esse segundo objeto no banco de dados evita que eu utilize um hack como `List.objects.first()` na implementação. Seria algo estúpido para fazer, e você poderia ir bem longe no caminho de testes para todas as implementações estúpidas que você não deve fazer (afinal de contas, há um número infinito delas). É uma questão de fazer uma avaliação, mas, nesse caso, parece que vale a pena. Há mais discussões sobre esse assunto em “Informações extras sobre quando testar a estupidez do desenvolvedor”.

Eis o que obtivemos:

```
AssertionError: 0 != 1
[...]
AssertionError: 301 != 302 : Response didn't redirect as expected: Response
code was 301 (expected 302)
```

Tome cuidado com expressões regulares gulosas!

Isso é um pouco estranho. Na verdade, ainda não havíamos especificado um URL para `/lists/1/add_item`, portanto nossa falha esperada seria `404 != 302`. Por que estamos obtendo um `301`?

Isso teve um quê de quebra-cabeça! É porque usamos uma expressão regular bem “gulosa” (greedy) em nosso URL:

superlists/urls.py

```
url(r'^lists/(.+)$', views.view_list, name='view_list'),
```

O Django apresenta um código embutido para gerar um redirecionamento permanente (`301`) sempre que alguém pedir um URL que seja *quase* certo, exceto por uma barra faltando. Nesse caso, haveria uma correspondência entre `/lists/1/add_item/` e `lists/(.+)`, com o `(.+)` capturando `1/add_item`. Assim, o Django “prestativamente” adivinhou que, na verdade, queríamos o URL com uma barra no final.

Podemos corrigir isso deixando nosso padrão de URL capturar explicitamente apenas os dígitos numéricos, usando a expressão regular `\d`:

superlists/urls.py

```
url(r'^lists/(\d+)/$', views.view_list, name='view_list'),
```

Isso nos dá a falha esperada:

```
AssertionError: 0 != 1
[...]
AssertionError: 404 != 302 : Response didn't redirect as expected: Response
code was 404 (expected 302)
```

Último URL novo

Agora que temos nosso 404 esperado, vamos adicionar um novo URL para o acréscimo de novos itens em listas existentes:

superlists/urls.py

```
urlpatterns = [  
    url(r'^$', views.home_page, name='home'),  
    url(r'^lists/new$', views.new_list, name='new_list'),  
    url(r'^lists/(\d+)/$', views.view_list, name='view_list'),  
    url(r'^lists/(\d+)/add_item$', views.add_item, name='add_item'),  
]
```

Há três URLs de aparência bastante semelhante aí. Vamos fazer uma anotação em nossa lista de tarefas; eles parecem bons candidatos para uma refatoração:

- ~~Ajustar o modelo para que os itens sejam associados a listas distintas~~
- ~~Adicionar URLs exclusivos para cada lista~~
- ~~Adicionar um URL para criar uma nova lista via POST~~
- Adicionar URLs para acrescentar um novo item em uma lista existente via POST
- Refatorar um pouco da duplicação em urls.py

De volta aos testes, como de costume, obtivemos a informação de objetos de view ausentes no módulo:

```
AttributeError: module 'lists.views' has no attribute 'add_item'
```

Última nova view

Vamos tentar isto:

lists/views.py

```
def add_item(request):  
    pass
```

A-ha:

TypeError: add_item() takes 1 positional argument but 2 were given

lists/views.py

```
def add_item(request, list_id):  
    pass
```

E, então, temos:

ValueError: The view lists.views.add_item didn't return an HttpResponse object. It returned None instead.

Podemos copiar o redirect de new_list e o List.objects.get de view_list:

lists/views.py

```
def add_item(request, list_id):  
    list_ = List.objects.get(id=list_id)  
    return redirect(f'/lists/{list_.id}/')
```

Isso nos leva a:

```
self.assertEqual(Item.objects.count(), 1)  
AssertionError: 0 != 1
```

Por fim, vamos fazer nosso novo item de lista ser salvo:

lists/views.py

```
def add_item(request, list_id):  
    list_ = List.objects.get(id=list_id)  
    Item.objects.create(text=request.POST['item_text'], list=list_)  
    return redirect(f'/lists/{list_.id}/')
```

E voltamos a ter testes que passam.

Ran 8 tests in 0.050s

OK

Testando os objetos de contexto de resposta diretamente

Temos nossa nova view e o novo URL para adição de itens em listas existentes; agora só precisamos realmente a usar em nosso template *list.html*. Portanto, vamos abri-lo para ajustar a tag de formulário...

lists/templates/list.html

```
<form method="POST" action="but what should we put here?">
```

...oh. Para obter o URL para a adição dos dados à lista atual, o template precisa saber qual lista está renderizando, assim como quais são os itens. Queremos ter a capacidade de fazer algo como:

lists/templates/list.html

```
<form method="POST" action="/lists/{{ list.id }}/add_item">
```

Para que isso funcione, a view terá que passar a lista para o template. Vamos criar um novo teste de unidade em ListViewTest:

lists/tests.py (ch07I041)

```
def test_passes_correct_list_to_template(self):
    other_list = List.objects.create()
    correct_list = List.objects.create()
    response = self.client.get(f'/lists/{correct_list.id}/')
    self.assertEqual(response.context['list'], correct_list) ❶
```

❶ `response.context` representa o contexto que passaremos para a função de renderização – o Django Test Client o coloca no objeto `response` para nós, a fim de ajudar nos testes.

Isso resulta em:

```
KeyError: 'list'
```

porque não estamos passando `list` para o template. Na verdade, temos a oportunidade de simplificar um pouco:

lists/views.py

```
def view_list(request, list_id):
    list_ = List.objects.get(id=list_id)
    return render(request, 'list.html', {'list': list_})
```

Isso, é claro, causará falhas em um de nossos testes antigos, pois o template precisava de itens:

```
FAIL: test_displays_only_items_for_that_list (lists.tests.ListViewTest)
[...]
AssertionError: False is not true : Couldn't find 'itemey 1' in response
```


No entanto, podemos corrigir isso em *list.html*, assim como ajustar a ação de POST do formulário:

lists/templates/list.html (ch07I043)

```
<form method="POST" action="/lists/{{ list.id }}/add_item"> ❶
```

```
[...]
```

```
{% for item in list.item_set.all %} ❷
```

```
<tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>
```

```
{% endfor %}
```

❶ Essa é a nova ação de nosso formulário.

❷ `.item_set` é chamado de lookup reverso (<https://docs.djangoproject.com/en/1.11/topics/db/queries/#following-relationships-backward>). É uma das partes incrivelmente úteis do ORM do Django, que permite consultar os itens relacionados a um objeto a partir de uma tabela diferente...

Com isso, conseguimos fazer os testes de unidade passarem:

```
Ran 9 tests in 0.040s
```

OK

E quanto aos FTs?

```
$ python manage.py test functional_tests
```

```
[...]
```

```
..
```

```
-----  
Ran 2 tests in 9.771s
```

OK

VIVA! Ah, vamos fazer uma rápida verificação em nossa lista de tarefas:

- ~~Ajustar o modelo para que os itens sejam associados a listas distintas~~
- ~~Adicionar URLs exclusivos para cada lista~~

- ~~Adicionar um URL para criar uma nova lista via POST~~
- ~~Adicionar URLs para acrescentar um novo item em uma lista existente via POST~~
- Refatorar um pouco da duplicação em `urls.py`

Irritantemente o Testing Goat também é um defensor de amarrar pontas soltas, portanto devemos executar essa última tarefa.

Antes de começar, faremos um commit – sempre garanta que tenha um commit de um estado funcional antes de embarcar em uma refatoração:

```
$ git diff
$ git commit -am "new URL + view for adding to existing lists. FT passes :-)"
```

Uma última refatoração usando inclusões de URL

`superlists/urls.py` serve realmente para URLs que se aplicam a todo o seu site. Para URLs que sejam somente para a aplicação `lists`, o Django nos incentiva a usar um `lists/urls.py` separado, a fim de deixar a aplicação mais autocontida. O modo mais simples de criar um é usar uma cópia do `urls.py` existente:

```
$ cp superlists/urls.py lists/
```

Então, substituímos três linhas em `superlists/urls.py` por um `include`:

`superlists/urls.py`

```
from django.conf.urls import include, url
from lists import views as list_views ❶
from lists import urls as list_urls ❶

urlpatterns = [
    url(r'^$', list_views.home_page, name='home'),
    url(r'^lists/', include(list_urls)), ❷
]
```

- ❶ Aproveitando a ocasião, usamos a sintaxe `import x as y` para criar aliases de `views` e `urls`. Essa é uma boa prática no `urls.py` de nível

mais alto, pois nos permitirá importar views e urls de várias aplicações, se quisermos – e, realmente, precisaremos fazer isso mais adiante no livro.

- ② Eis o `include`. Observe que ele pode usar parte de uma regex de URL como prefixo, que será aplicado a todos os URLs incluídos (é nessa parte que reduzimos a duplicação, assim como conferimos melhor estrutura ao nosso código).

De volta a `lists/urls.py`, podemos reduzi-lo de modo que inclua somente a última parte de nossos três URLs, e nenhuma das demais informações do `urls.py` pai:

lists/urls.py (ch07I046)

```
from django.conf.urls import url
from lists import views

urlpatterns = [
    url(r'^new$', views.new_list, name='new_list'),
    url(r'^(d+)/$', views.view_list, name='view_list'),
    url(r'^(d+)/add_item$', views.add_item, name='add_item'),
]
```

Execute os testes de unidade novamente para verificar se tudo funcionou.

Quando fiz isso, quase não pude acreditar que tudo estava correto na primeira tentativa. Sempre vale a pena ser cético quanto às próprias habilidades, portanto alterei levemente um dos URLs de modo proposital, apenas para verificar se um teste falharia. O teste falhou. Estamos cobertos.

Sinta-se à vontade para testar por conta própria! Lembre-se de desfazer a mudança, verifique se todos os testes passam novamente e então execute um commit final:

```
$ git status
$ git add lists/urls.py
$ git add superlists/urls.py
$ git diff --staged
$ git commit
```

Ufa! Um capítulo que foi uma maratona. No entanto, abordamos uma série de tópicos importantes, começando pelo isolamento dos testes e então algumas ideias sobre design. Discutimos algumas regras gerais, por exemplo, “YAGNI” e “três acertos e então refatorar”. Acima de tudo, porém, vimos como adaptar um site existente passo a passo, indo de um estado funcional a outro, a fim de iterar em direção a um novo design.

Eu diria que estamos bem perto de lançar esse site como o primeiríssimo beta do site de superlistas que vai dominar o mundo. Talvez ele precise de um pouco de embelezamento antes... vamos ver o que precisaremos fazer para a sua implantação nos próximos dois capítulos.

Um pouco mais sobre a filosofia do TDD

De estado funcional para estado funcional (também conhecido como Testing Goat versus Refactoring Cat)

Nosso impulso natural geralmente é mergulhar de cabeça e corrigir tudo de uma só vez... porém, se não tomarmos cuidado, acabaremos como o Refactoring Cat, em uma situação com muitas alterações em nosso código, sem que nada funcione. O Testing Goat nos incentiva a dar um passo da cada vez, e ir de um estado funcional a outro.

Divida o trabalho em tarefas pequenas e viáveis

Às vezes, isso significa começar com um trabalho “maçante” em vez de mergulhar direto na parte divertida, mas você deverá acreditar que aquele você-YOLO no universo paralelo provavelmente está passando por maus bocados, tendo causado falhas em tudo, e luta para fazer a aplicação funcionar novamente.

YAGNI

You ain't gonna need it (Você não vai precisar disso)! Evite a tentação de escrever um código que você *ache* que poderá ser útil, somente porque a ocasião sugeria isso. É provável que você não use esse código, ou que não tenha previsto seus futuros requisitos corretamente. Leia o Capítulo 22 para ver uma metodologia que nos ajudará a evitar essa armadilha.

PARTE II

Sine qua nons do desenvolvimento web

Desenvolvedores de verdade fazem lançamentos.

– JEFF ATWOOD

Se este fosse apenas um guia para TDD em um campo usual de programação, poderíamos nos parabenizar agora. Afinal de contas, temos na manga um pouco de uma base sólida sobre TDD e Django; dispomos de tudo que é necessário para começar a construir um site.

No entanto, desenvolvedores de verdade fazem lançamentos, e, para isso, teremos que enfrentar alguns dos aspectos mais intrincados, porém inevitáveis, do desenvolvimento web: arquivos estáticos, validação de dados de formulários, o temível JavaScript, e o mais cabeludo de todos, a implantação em um servidor de produção.

Em cada etapa, o TDD pode nos ajudar a fazer tudo isso de forma correta também.

Nesta seção, continuarei tentando manter a curva de aprendizado relativamente suave, mas veremos vários novos conceitos e tecnologias importantes. Poderei descrever cada um deles apenas superficialmente – espero demonstrar o suficiente sobre cada tópico para que você comece quando estiver trabalhando em seu projeto, mas será necessário fazer suas próprias leituras por aí quando passar a aplicar esses tópicos na “vida real”.

Por exemplo, se você não tinha familiaridade com o Django antes de

começar a ler este livro, talvez ache que investir um pouco de tempo para dar uma olhada no tutorial oficial do Django a essa altura complementarará muito bem o que você aprendeu até agora, e o deixará mais confiante sobre assuntos relacionados ao Django nos próximos capítulos, de modo a se concentrar nos conceitos essenciais.

Ah, mas há muitos assuntos divertidos pela frente! Espera e verá!

CAPÍTULO 8

Embelezamento: layout e estilização, e o que testar sobre eles

Estamos começando a pensar em lançar a primeira versão de nosso site, mas sentimos um pouco de vergonha de sua deselegância no momento. Neste capítulo, discutiremos um pouco do básico sobre estilização, incluindo a integração de um framework HTML/CSS chamado Bootstrap. Veremos como os arquivos estáticos funcionam no Django e o que precisamos fazer para testá-los.

O que testar funcionalmente quanto ao layout e ao estilo

Sem dúvida, nosso site está um tanto quanto pouco atraente no momento (Figura 8.1).



Se você iniciar seu servidor de desenvolvimento com `manage.py runserver`, talvez depare com um erro de banco de dados: “table lists_item has no column named list_id” (a tabela `lists_item` não tem nenhuma coluna chamada `list_id`). Atualize seu banco de dados local para que ele reflita as alterações que fizemos em *models.py*. Utilize `manage.py migrate`. Se ele apresentar alguma reclamação sobre `IntegrityErrors`, basta apagar¹ o arquivo do banco de dados e tentar novamente.

Não podemos contribuir para piorar a reputação de Python por ser feio (<http://grokcode.com/746/dear-python-why-are-you-so-ugly/>), portanto vamos dar uma leve polida na aparência do site. Eis alguns

itens que podemos desejar:

- um bom campo de entrada grande para adicionar listas novas e existentes;
- uma caixa grande, chamativa e centralizada para colocá-lo.

Como podemos aplicar o TDD nesses itens? A maioria das pessoas dirá que você não deve testar a estética, e elas estão certas. É um pouco como testar uma constante, no sentido de que os testes geralmente não lhe acrescentarão nada.

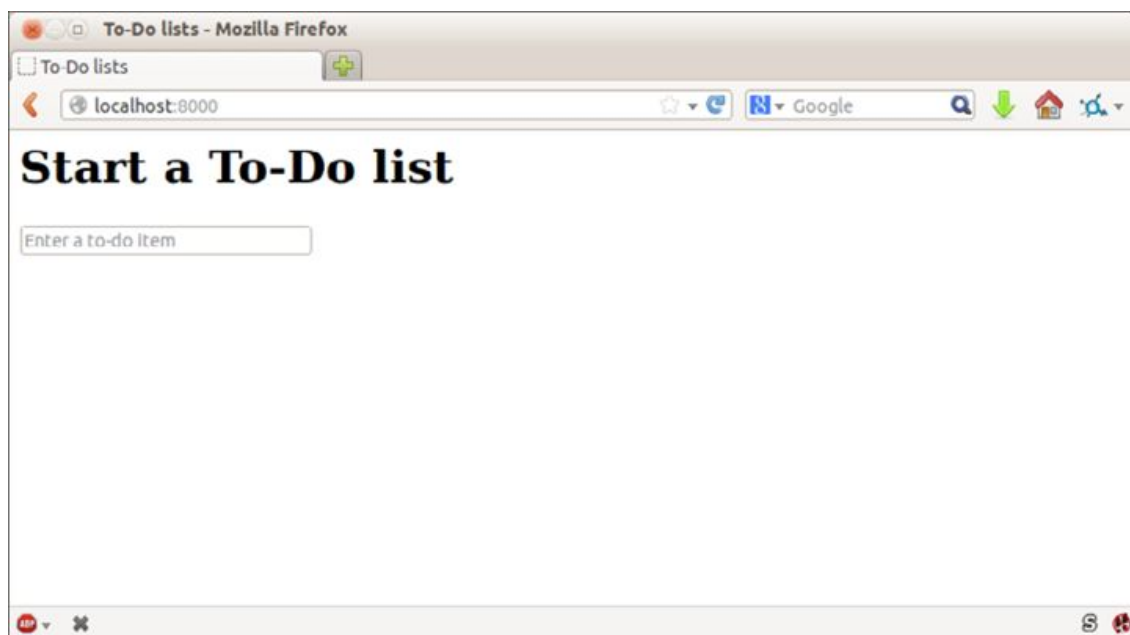


Figura 8.1 – Nossa página inicial, parecendo um pouco feia...

Contudo, podemos testar a implementação de nossa estética – apenas o suficiente para nos garantir que tudo está funcionando. Por exemplo, usaremos CSS (Cascading Style Sheets, ou Folhas de Estilo em Cascata) para a nossa estilização, e eles serão carregados como arquivos estáticos. Arquivos estáticos podem ser um pouco complicados de configurar (especialmente, como veremos mais adiante, quando passamos de nosso próprio microcomputador para um site de hospedagem), portanto queremos ter algum tipo de “teste de fumaça” (smoke test) simples para verificar se o CSS foi carregado. Não precisamos testar fontes nem cores ou cada pixel único, mas podemos realizar uma verificação rápida para saber se a

caixa de entrada principal está alinhada do modo como gostaríamos em cada página, e isso nos deixará confiantes de que o restante da estilização dessa página provavelmente estará carregado também.

Começaremos com um novo método de teste em nosso teste funcional:

functional_tests/tests.py (ch08I001)

```
class NewVisitorTest(LiveServerTestCase):
    [...]

    def test_layout_and_styling(self):
        # Edith acessa a página inicial
        self.browser.get(self.live_server_url)
        self.browser.set_window_size(1024, 768)

        # Ela percebe que a caixa de entrada está elegantemente centralizada
        inputbox = self.browser.find_element_by_id('id_new_item')
        self.assertAlmostEqual(
            inputbox.location['x'] + inputbox.size['width'] / 2,
            512,
            delta=10
        )
```

Há algumas novidades aqui. Começamos definindo o tamanho da janela com um tamanho fixo. Então, encontramos o elemento de entrada, vemos o seu tamanho e a localização e fazemos alguns cálculos matemáticos para verificar se ele parece posicionado no meio da página. `assertAlmostEqual` nos ajuda a lidar com erros de arredondamento e o ocasional resultado inusitado como consequência das barras de rolagem e itens semelhantes, deixando-nos especificar que queremos que nossa aritmética trabalhe com 10 pixels para mais ou para menos.

Se os testes funcionais forem executados, veremos o seguinte:

```
$ python manage.py test functional_tests
[...]
.F.
```

```
=====
=====
FAIL: test_layout_and_styling (functional_tests.tests.NewVisitorTest)
-----
Traceback (most recent call last):
  File ".../superlists/functional_tests/tests.py", line 129, in
test_layout_and_styling
    delta=10
AssertionError: 107.0 != 512 within 10 delta

-----
Ran 3 tests in 9.188s
```

FAILED (failures=1)

Essa é a falha esperada. No entanto, esse tipo de FT pode facilmente dar errado, portanto vamos usar uma solução “trapaceira” rápida e grosseira para verificar se o FT também passa quando a caixa de entrada está centralizada. Apagaremos esse código mais uma vez, logo depois de tê-lo usado para verificar o FT:

lists/templates/home.html (ch08I002)

```
<form method="POST" action="/lists/new">
  <p style="text-align: center;">
    <input name="item_text" id="id_new_item" placeholder="Enter a to-do item" />
  </p>
  {% csrf_token %}
</form>
```

O teste passa, o que significa que o FT funciona. Vamos estendê-lo para garantir que a caixa de entrada também está centralizada na página com uma lista nova:

functional_tests/tests.py (ch08I003)

```
# Ela inicia uma nova lista e vê que a entrada está elegantemente
# centralizada aí também
inputbox.send_keys('testing')
inputbox.send_keys(Keys.ENTER)
self.wait_for_row_in_list_table('1: testing')
inputbox = self.browser.find_element_by_id('id_new_item')
```

```
self.assertAlmostEqual(
    inputbox.location['x'] + inputbox.size['width'] / 2,
    512,
    delta=10
)
```

Isso resulta em outra falha de teste:

```
File ".../superlists/functional_tests/tests.py", line 141, in
test_layout_and_styling
    delta=10
AssertionError: 107.0 != 512 within 10 delta
```

Vamos fazer commit somente do FT:

```
$ git add functional_tests/tests.py
$ git commit -m "first steps of FT for layout + styling"
```

Agora temos a sensação de que há uma justificativa para encontrar uma solução “apropriada” e atender às nossas necessidades de ter uma estilização melhor para o site. Podemos eliminar o nosso hack `<p style="text-align: center">`:

```
$ git reset --hard
```



`git reset --hard` é o comando do Git para “lançar um míssil e tirar o site de órbita”, portanto tome cuidado com ele – o comando destruirá todas as suas alterações para as quais um commit não tenha sido efetuado. De modo diferente de quase tudo o mais que você possa fazer com o Git, não há maneiras de voltar atrás nesse caso.

Embelezamento: usando um framework CSS

Design é uma tarefa difícil, e é duplamente mais complicado agora que temos de lidar com dispositivos móveis, tablets, entre outros. É por isso que muitos programadores, particularmente os preguiçosos como eu, estão lançando mão de frameworks CSS para resolver alguns desses problemas para eles. Há muitos frameworks por aí, mas um dos mais antigos e populares é o Bootstrap do Twitter. Vamos usá-lo.

O Bootstrap pode ser encontrado em <http://getbootstrap.com/>.

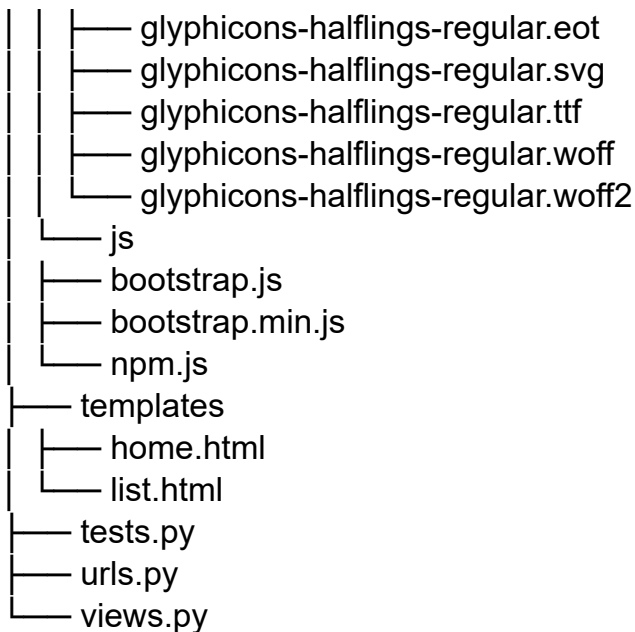
Faremos o download e o colocaremos em uma nova pasta chamada *static* na aplicação *lists*:²

```
$ wget -O bootstrap.zip
https://github.com/twbs/bootstrap/releases/download/v3.3.4/bootstrap-3.3.4-dist.zip
$ unzip bootstrap.zip
$ mkdir lists/static
$ mv bootstrap-3.3.4-dist lists/static/bootstrap
$ rm bootstrap.zip
```

O Bootstrap vem com uma instalação simples, não personalizada, na pasta *dist*. Vamos usá-la por enquanto, mas você jamais deve fazer isso em um site de verdade – o Bootstrap básico é instantaneamente reconhecível, e um sinal evidente para qualquer um com conhecimento no assunto de que você não se deu ao trabalho de estilizar o seu site. Aprenda a usar LESS e, no mínimo, mude a fonte! Há informações na documentação do Bootstrap, ou você pode encontrar um bom guia em <http://coding.smashingmagazine.com/2013/03/12/customizing-bootstrap/>.

Nossa pasta *lists* acabará com o seguinte aspecto:

```
$ tree lists
lists
├── __init__.py
├── __pycache__
│   └── [...]
├── admin.py
├── models.py
├── static
│   └── bootstrap
│       ├── css
│       │   ├── bootstrap.css
│       │   ├── bootstrap.css.map
│       │   ├── bootstrap.min.css
│       │   ├── bootstrap-theme.css
│       │   ├── bootstrap-theme.css.map
│       │   └── bootstrap-theme.min.css
│       └── fonts
```



Observe a seção “Getting Started” (Introdução) da documentação do Bootstrap (<http://bit.ly/2u1IROA>); você verá que ele quer que nosso template HTML inclua algo como:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Bootstrap 101 Template</title>
    <!-- Bootstrap -->
    <link href="css/bootstrap.min.css" rel="stylesheet">
  </head>
  <body>
    <h1>Hello, world!</h1>
    <script src="http://code.jquery.com/jquery.js"></script>
    <script src="js/bootstrap.min.js"></script>
  </body>
</html>
```

Já temos dois templates HTML. Não queremos adicionar um volume enorme de código boilerplate em cada um, portanto esta parece uma boa hora para aplicar a regra “Don’t repeat yourself” (Não se repita) e colocar todas as partes comuns em um só lugar.

Felizmente a linguagem de template do Django facilita fazer isso usando algo chamado herança de template.

Herança de templates do Django

Vamos fazer uma pequena revisão sobre as diferenças entre *home.html* e *list.html*:

```
$ diff lists/templates/home.html lists/templates/list.html
< <h1>Start a new To-Do list</h1>
< <form method="POST" action="/lists/new">
---
> <h1>Your To-Do list</h1>
> <form method="POST" action="/lists/{{ list.id }}/add_item">
[...]
```

<table id="id_list_table">
<{% for item in list.item_set.all %>
<tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>
<{% endfor %>
> </table>

Eles têm textos de cabeçalho diferentes e seus formulários usam URLs distintos. Além disso, *list.html* tem o elemento adicional `<table>`.

Agora que está claro o que é comum e o que não é, podemos fazer com que os dois templates herdem de um template que é uma “superclasse” comum. Começaremos fazendo uma cópia de *home.html*:

```
$ cp lists/templates/home.html lists/templates/base.html
```

Faremos com que esse seja um template-base contendo apenas o boilerplate comum e marcaremos os “blocos”, que são os lugares em que os templates-filhos poderão personalizá-lo:

lists/templates/base.html

```
<html>
  <head>
    <title>To-Do lists</title>
  </head>
```

```

<body>
  <h1>{% block header_text %}{% endblock %}</h1>
  <form method="POST" action="{% block form_action %}{% endblock %}">
    <input name="item_text" id="id_new_item" placeholder="Enter a to-do item"
  />
  {% csrf_token %}
</form>
{% block table %}
{% endblock %}
</body>
</html>

```

O template-base define uma série de áreas chamadas “blocos”, que serão locais em que outros templates poderão se engatar e adicionar o próprio conteúdo. Vamos ver como isso funciona na prática, alterando *home.html* de modo que ele “herde de” *base.html*:

lists/templates/home.html

```

{% extends 'base.html' %}

{% block header_text %}Start a new To-Do list{% endblock %}

{% block form_action %}/lists/new{% endblock %}

```

É possível ver que muito do boilerplate HTML desaparece, e podemos nos concentrar somente nas partes que queremos personalizar. Fazemos o mesmo com *list.html*:

lists/templates/list.html

```

{% extends 'base.html' %}

{% block header_text %}Your To-Do list{% endblock %}

{% block form_action %}/lists/{{ list.id }}/add_item{% endblock %}

{% block table %}
<table id="id_list_table">
  {% for item in list.item_set.all %}
  <tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>

```

```
{% endfor %}  
</table>  
{% endblock %}
```

É uma refatoração do modo como nossos templates funcionam. Vamos executar os FTs novamente para garantir que não causamos problemas em nada...

```
AssertionError: 107.0 != 512 within 10 delta
```

Sem dúvida, eles estão chegando exatamente até o ponto em que estavam antes. Vale a pena fazer um commit:

```
$ git diff -b  
# -b significa ignorar espaços em branco, e é conveniente, pois alteramos  
# algumas indentações de html  
$ git status  
$ git add lists/templates # leave static, for now  
$ git commit -m "refactor templates to use a base template"
```

Integrando o Bootstrap

Agora é muito mais fácil integrar o código boilerplate que o Bootstrap quer – ainda não acrescentaremos o JavaScript, apenas o CSS:

lists/templates/base.html (ch08I006)

```
<!DOCTYPE html>  
<html lang="en">  
  
  <head>  
    <meta charset="utf-8">  
    <meta http-equiv="X-UA-Compatible" content="IE=edge">  
    <meta name="viewport" content="width=device-width, initial-scale=1">  
    <title>To-Do lists</title>  
    <link href="css/bootstrap.min.css" rel="stylesheet">  
  </head>  
  [...]
```

Linhas e colunas

Por fim, vamos realmente usar um pouco da mágica do Bootstrap!

Você terá que ler a documentação por conta própria, mas poderemos utilizar uma combinação do sistema de grade e da classe `text-center` para obter o que queremos:

lists/templates/base.html (ch08I007)

```
<body>
  <div class="container">

    <div class="row">
      <div class="col-md-6 col-md-offset-3">
        <div class="text-center">
          <h1>{% block header_text %}{% endblock %}</h1>
          <form method="POST" action="{% block form_action %}{% endblock
%}">
            <input name="item_text" id="id_new_item"
              placeholder="Enter a to-do item" />
            {% csrf_token %}
          </form>
        </div>
      </div>
    </div>

    <div class="row">
      <div class="col-md-6 col-md-offset-3">
        {% block table %}
        {% endblock %}
      </div>
    </div>

  </div>
</body>
```

(Se você nunca viu uma tag HTML dividida em várias linhas, esse `<input>` talvez seja um pouco chocante. Sem dúvida, ele é válido, mas você não precisará usá-lo se julgá-lo ofensivo.)



Reserve tempo para navegar pela documentação do Bootstrap (<http://getbootstrap.com/>), caso não a tenha visto antes. É um carrinho de compras cheio de ferramentas úteis para usar em seu site.

Isso funciona?

```
AssertionError: 107.0 != 512 within 10 delta
```

Humm. Não. Por que o nosso CSS não está carregando?

Arquivos estáticos no Django

O Django e, na verdade, qualquer servidor web precisam de duas informações para lidar com arquivos estáticos:

1. Como saber se uma requisição de URL é para um arquivo estático, em oposição a algum HTML que será servido por meio de uma função de view.
2. Onde encontrar o arquivo estático que o usuário quer.

Em outras palavras, os arquivos estáticos são um mapeamento de URLs para arquivos em disco.

Para o item 1, o Django nos permite definir um “prefixo” de URL para dizer que qualquer URL que comece com esse prefixo deve ser tratado como requisições de arquivos estáticos. Por padrão, o prefixo é */static/*. Ele está definido em *settings.py*:

superlists/settings.py

```
[...]
```

```
# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/1.11/howto/static-files/
```

```
STATIC_URL = '/static/'
```

Todo o restante das configurações que adicionarmos nesta seção estará associado ao item 2: encontrar os arquivos estáticos propriamente ditos no disco.

Enquanto estivermos usando o servidor de desenvolvimento do Django (`manage.py runserver`), podemos contar com esse framework para encontrar os arquivos estáticos para nós como num passe de mágica – ele simplesmente procurará em qualquer subpasta chamada *static* de uma de nossas aplicações.

Agora você perceberá por que colocamos todos os arquivos estáticos do Bootstrap em *lists/static*. Então, por que eles não estão funcionando no momento? Porque não estamos usando o prefixo de URL */static/*. Observe novamente o link para o CSS em *base.html*:

```
<link href="css/bootstrap.min.css" rel="stylesheet">
```

Para que funcione, precisamos alterar isso para:

lists/templates/base.html

```
<link href="/static/bootstrap/css/bootstrap.min.css" rel="stylesheet">
```

Quando runserver vir a requisição, ele saberá que é para um arquivo estático, pois começa com */static/*. Então tentará encontrar um arquivo chamado *bootstrap/css/bootstrap.min.css*, procurando em cada pasta de nossa aplicação em busca de subpastas de nome *static*, e deverá encontrá-lo em *lists/static/bootstrap/css/bootstrap.min.css*.

Assim, se você verificar manualmente, deverá ver que isso funciona, como na Figura 8.2.

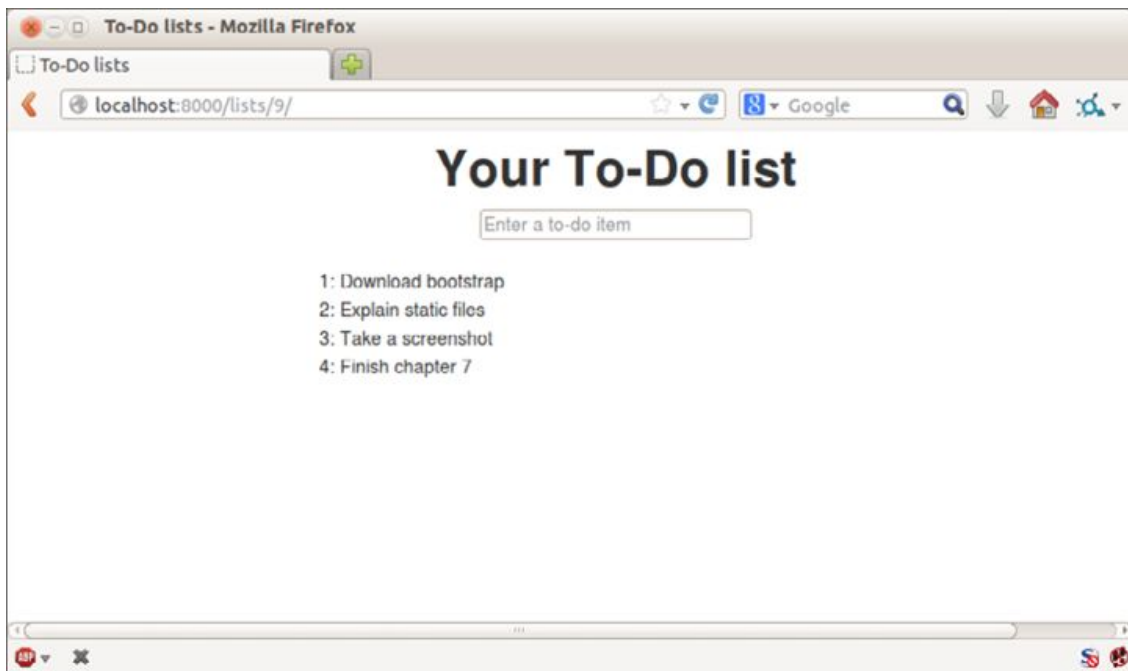


Figura 8.2 – Nosso site começa a ficar com uma aparência um pouco melhor...

Mudando para StaticLiveServerTestCase

Se você executar o FT, porém, ele não passará:

```
AssertionError: 107.0 != 512 within 10 delta
```

Isso ocorre porque, embora runserver encontre os arquivos estáticos automaticamente, como num passe de mágica, LiveServerTestCase não o faz. Contudo, não tenha medo: os desenvolvedores do Django criaram uma classe de teste mais mágica ainda chamada StaticLiveServerTestCase (veja a documentação em <http://bit.ly/Suv4lp>).

Vamos começar a usá-la:

functional_tests/tests.py

```
@@ -1,14 +1,14 @@
-from django.test import LiveServerTestCase
+from django.contrib.staticfiles.testing import StaticLiveServerTestCase
 from selenium import webdriver
 from selenium.common.exceptions import WebDriverException
 from selenium.webdriver.common.keys import Keys
 import time
```

```
MAX_WAIT = 10
```

```
-class NewVisitorTest(LiveServerTestCase):
+class NewVisitorTest(StaticLiveServerTestCase):
```

```
    def setUp(self):
```

Agora o novo CSS será encontrado e o nosso teste passará:

```
$ python manage.py test functional_tests
Creating test database for alias 'default'...
```

```
...
```

```
-----
Ran 3 tests in 9.764s
```



Nesse ponto, os usuários de Windows talvez vejam algumas mensagens de erro (inócuas, porém desconcertantes) que informam o seguinte:

socket.error: [WinError 10054] An existing connection was forcibly closed by the remote host (Uma conexão existente foi forçosamente encerrada pelo host remoto). Adicione um `self.browser.refresh()` imediatamente antes de `self.browser.quit()` em `tearDown` a fim de se livrar deles. O problema está sendo acompanhado no sistema de monitoração de bugs do Django (<https://code.djangoproject.com/ticket/21227>).

Viva!

Usando componentes do Bootstrap para melhorar a aparência do site

Vamos ver se podemos fazer algo melhor ainda usando algumas das outras ferramentas da parafernália do Bootstrap.

Jumbotron!

O Bootstrap possui uma classe chamada `jumbotron` para itens que devam ser particularmente proeminentes na página. Vamos usá-la para ampliar o cabeçalho da página principal e o formulário de entrada:

lists/templates/base.html (ch08I009)

```
<div class="col-md-6 col-md-offset-3 jumbotron">
  <div class="text-center">
    <h1>{% block header_text %}{% endblock %}</h1>
    <form method="POST" action="{% block form_action %}{% endblock %}">
      [...]
```



Quando estiver trabalhando com design e layout, é melhor ter uma janela aberta na qual possamos acionar uma atualização com frequência. Utilize `python manage.py runserver` para iniciar o servidor de desenvolvimento e então acesse `http://localhost:8000` para ver o seu trabalho à medida que progredir.

Entradas maiores

O `jumbotron` é um bom ponto de partida, mas, no momento, a caixa de entrada apresenta um texto minúsculo quando comparada a tudo

o mais. Felizmente as classes de controle de formulários do Bootstrap oferecem uma opção para configurar uma entrada como “grande” (large):

lists/templates/base.html (ch08l010)

```
<input name="item_text" id="id_new_item"
      class="form-control input-lg"
      placeholder="Enter a to-do item" />
```

Estilização de tabelas

O texto da tabela também parece pequeno demais quando comparado ao restante da página agora. A adição da classe `table` do Bootstrap melhora a situação:

lists/templates/list.html (ch08l011)

```
<table id="id_list_table" class="table">
```

Usando o nosso próprio CSS

Por fim, eu gostaria simplesmente de afastar um pouco a entrada em relação ao texto do título. Não há nenhuma correção pronta para isso no Bootstrap, portanto vamos fazer a nossa própria implementação. Isso exigirá a especificação de um arquivo CSS nosso:

lists/templates/base.html

```
[...]
<title>To-Do lists</title>
<link href="/static/bootstrap/css/bootstrap.min.css" rel="stylesheet">
<link href="/static/base.css" rel="stylesheet">
</head>
```

Criamos um novo arquivo em `lists/static/base.css` com nossa nova regra de CSS. Usaremos o id do elemento de entrada, `id_new_item`, para encontrá-lo e aplicar-lhe um pouco de estilização:

lists/static/base.css

```
#id_new_item {  
    margin-top: 2ex;  
}
```

Tudo isso exigiu que eu realizasse algumas tentativas, mas estou razoavelmente satisfeito com o resultado agora (Figura 8.3).

Se quiser ir além na personalização do Bootstrap, deverá se envolver com a compilação de LESS. *Definitivamente*, recomendo que você invista tempo para isso algum dia. LESS e outros recursos semelhantes a um pseudoCSS, como o Sass, representam uma grande melhoria em relação ao velho CSS básico, e são uma ferramenta útil mesmo que você não utilize o Bootstrap. Não discutirei esse assunto neste livro, mas é possível encontrar recursos na internet. Eis um deles: <http://coding.smashingmagazine.com/2013/03/12/customizing-bootstrap/>, por exemplo.

Vamos fazer uma última execução dos testes funcionais para ver se tudo continua executando sem problemas:

```
$ python manage.py test functional_tests
```

```
[...]
```

```
...
```

```
-----  
Ran 3 tests in 10.084s
```

```
OK
```

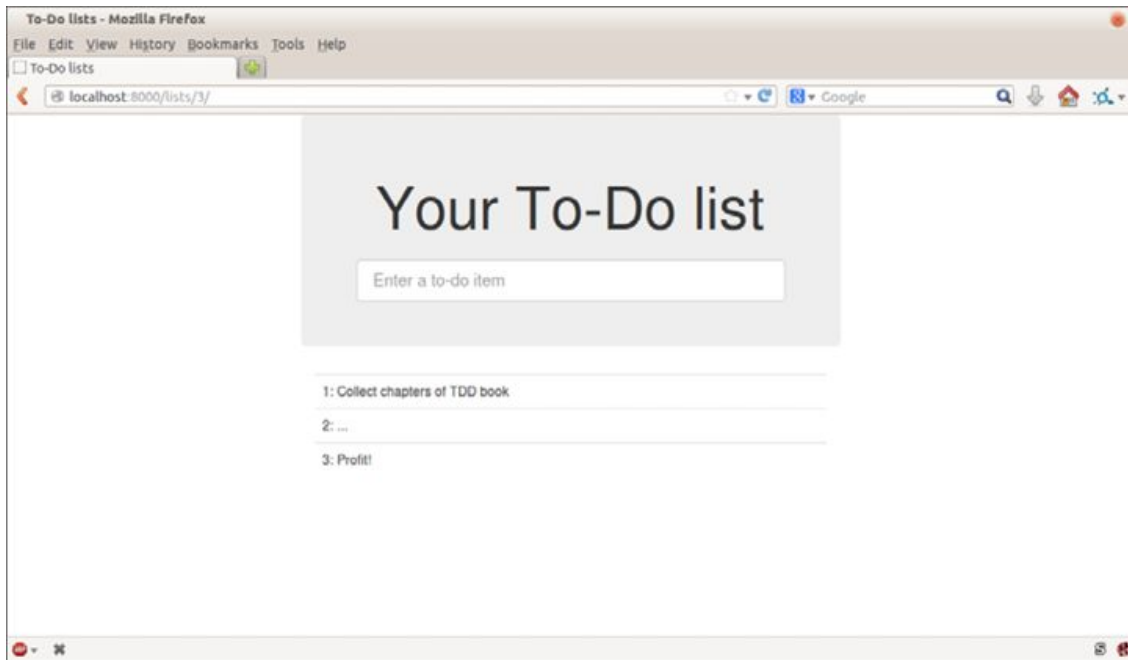


Figura 8.3 – A página de listas, com todas as partes maiores...

É isso! Com certeza é hora de fazer um commit:

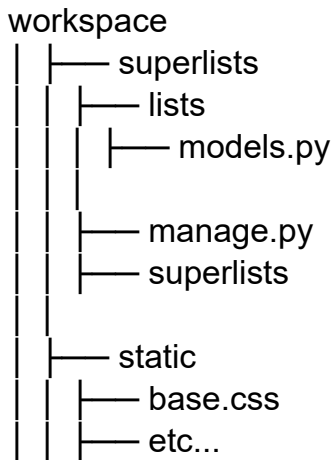
```
$ git status # mudanças em tests.py, base.html, list.html + lists/static
              # ainda não controlado
$ git add .
$ git status # mostrará agora todos os acréscimos do bootstrap
$ git commit -m "Use Bootstrap to improve layout"
```

O que não havíamos revelado: collectstatic e outros diretórios estáticos

Vimos antes que o servidor de desenvolvimento do Django encontrará todos os seus arquivos estáticos nas pastas da aplicação em um passe de mágica, e os servirá para você. Isso não é um problema durante a fase de desenvolvimento, mas, quando a execução ocorrer em um servidor web de verdade, você não vai querer que o Django sirva o seu conteúdo estático – usar Python para servir arquivos brutos é lento e ineficaz, e um servidor web como o Apache ou o Nginx pode fazer tudo isso para você. É possível até mesmo decidir fazer upload de todos os seus arquivos estáticos para um CDN, em vez de hospedá-los por conta própria.

Por esses motivos, você deverá ser capaz de reunir todos os seus arquivos estáticos das várias pastas da aplicação e copiá-los para um único local, deixando-os prontos para a implantação. É para isso que serve o comando `collectstatic`.

O destino – o lugar em que os arquivos estáticos reunidos serão colocados – é definido em `settings.py` com `STATIC_ROOT`. No próximo capítulo, faremos um pouco do processo de implantação, portanto vamos realizar alguns experimentos com isso agora. Alteraremos seu valor para uma pasta fora de nosso repositório – farei com que seja uma pasta ao lado da pasta principal de códigos-fontes:



A lógica é que a pasta de arquivos estáticos não deve fazer parte de seu repositório – não queremos colocá-la no sistema de controle de versões, pois ela é uma duplicata de todos os arquivos que estão em `lists/static`.

Eis uma forma organizada de especificar essa pasta, deixando sua localização relativa ao diretório-base do projeto:

superlists/settings.py (ch08l018)

```
# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/1.11/howto/static-files/
```

```
STATIC_URL = '/static/'
STATIC_ROOT = os.path.abspath(os.path.join(BASE_DIR, '../static'))
```

Dê uma olhada no início do arquivo de configurações e você verá

como a variável `BASE_DIR` está convenientemente definida para nós, usando `__file__` (que, por si só, é um recurso embutido de Python realmente muito, muito útil).

De qualquer modo, vamos experimentar executar `collectstatic`:

```
$ python manage.py collectstatic
```

```
[...]
```

```
Copying '/.../superlists/lists/static/bootstrap/css/bootstrap-theme.css'
```

```
Copying '/.../superlists/lists/static/bootstrap/css/bootstrap.min.css'
```

```
76 static files copied to '/.../static'.
```

Se observarmos `../static`, veremos todos os nossos arquivos CSS:

```
$ tree ../static/
```

```
../static/
```

```
|— admin
|   |— css
|   |   |— base.css
```

```
[...]
```

```
|   |— xregexp.min.js
|   |— base.css
|   |— bootstrap
|       |— css
|           |— bootstrap.css
|           |— bootstrap.css.map
|           |— bootstrap.min.css
|           |— bootstrap-theme.css
|           |— bootstrap-theme.css.map
|           |— bootstrap-theme.min.css
|       |— fonts
|           |— glyphsicons-halflings-regular.eot
|           |— glyphsicons-halflings-regular.svg
|           |— glyphsicons-halflings-regular.ttf
|           |— glyphsicons-halflings-regular.woff
|           |— glyphsicons-halflings-regular.woff2
|       |— js
|           |— bootstrap.js
|           |— bootstrap.min.js
```

└─ npm.js

14 directories, 76 files

collectstatic também reuniu todo o CSS para o site de administração. É um dos recursos mais eficazes do Django, e descobriremos tudo sobre ele um dia, mas não estamos prontos para usá-lo ainda, portanto vamos desativá-lo por enquanto:

superlists/settings.py

```
INSTALLED_APPS = [  
    #'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'lists',  
]
```

Vamos tentar novamente:

```
$ rm -rf ../static/  
$ python manage.py collectstatic --noinput  
Copying '/.../superlists/lists/static/base.css'  
[...]  
Copying '/.../superlists/lists/static/bootstrap/css/bootstrap-theme.css'  
Copying '/.../superlists/lists/static/bootstrap/css/bootstrap.min.css'
```

15 static files copied to '/.../static'.

Muito melhor.

De qualquer forma, agora sabemos como reunir todos os arquivos estáticos em uma única pasta, e esses arquivos poderão ser facilmente encontrados por um servidor web. Descobriremos tudo sobre esse assunto, incluindo como fazer os testes, no próximo capítulo!

Por ora, vamos salvar nossas alterações em *settings.py*:


```
$ git diff # deve mostrar alterações em settings.py*
```

```
$ git commit -am "set STATIC_ROOT in settings and disable admin"
```

Alguns pontos que ficaram de fora

Inevitavelmente esse foi apenas um tour rápido sobre estilização e CSS, e houve vários tópicos que considerei discutir, mas que acabaram ficando de fora. Eis alguns candidatos para um estudo mais aprofundado:

- personalizar o bootstrap com LESS ou SASS;
- a tag de template `{% static %}` para mais DRY e menos URLs fixos;
- ferramentas para empacotamento do lado do cliente, como `npm` e `bower`.

Revisão: sobre testar design e layout

A resposta curta e grossa é: você não deveria escrever testes para design e layout *per se*. É muito semelhante a testar uma constante, e os testes que você escrever, com frequência, serão frágeis.

Apesar do que dissemos, a *implementação* do design e do layout envolve aspectos um tanto quanto intrincados: CSS e arquivos estáticos. Como resultado, é importante ter algum tipo de “teste de fumaça” (smoke test) mínimo, que verifique se seus arquivos estáticos e o CSS estão funcionando. Como veremos no próximo capítulo, esses testes poderão ajudar a identificar problemas quando seu código for implantado no ambiente de produção.

De modo semelhante, se uma parte específica da estilização exigir muito código JavaScript do lado do cliente para funcionar (redimensionamento dinâmico é um ponto no qual gastei certo tempo), definitivamente você vai querer ter alguns testes para isso.

Procure escrever os testes mínimos que lhe darão a confiança de que seu design e o layout estão funcionando, sem testar *o que* eles são realmente. Procure se colocar em uma posição em que você possa realizar alterações livremente no design e no layout sem ter de voltar e adaptar os testes o tempo todo.

- 1 O quê? Apagar o banco de dados? Você ficou maluco? Não totalmente. O banco de dados local de desenvolvimento com frequência perde a sincronização com suas migrações à medida que retrocedemos e avançamos em nosso desenvolvimento, e ele não contém nenhum dado importante, portanto não haverá problemas em apagá-lo ocasionalmente. Seremos muito mais cuidadosos quando tivermos um banco de dados de “produção” no servidor. Há mais informações sobre esse assunto no Apêndice D.
- 2 No Windows, talvez você não tenha wget nem unzip, mas estou certo de que poderá descobrir como fazer download do Bootstrap, descompactá-lo e colocar o conteúdo da pasta *dist* na pasta *lists/static/bootstrap*.

CAPÍTULO 9

Testando a implantação usando um site de staging

É tudo diversão e entretenimento até você ter que colocar em produção.

– DEVOPS BORAT ([HTTP://BIT.LY/2UHCXNH](http://bit.ly/2UHCXNH))

É hora de fazer a implantação da primeira versão de nosso site e torná-lo público. Dizem que, se você esperar até se sentir pronto para fazer o lançamento, então terá esperado demais.

O nosso site é utilizável? É melhor que nada? Podemos criar listas nele? Sim, sim, sim.

Não, ainda não podemos fazer login. Não, não podemos marcar as tarefas como concluídas. Mas precisamos realmente de tudo isso? Na verdade, não – e você não poderá jamais ter certeza do que seus usuários *realmente* farão com seu site depois que o tiverem nas mãos. Achamos que nossos usuários querem usar o site para listas de tarefas, mas talvez eles queiram usá-lo para criar listas dos “10 melhores locais para pesca com fly”, para as quais não precisaremos de nenhum tipo de função para “marcar como concluída”. Não saberemos até colocarmos o site lá fora.

Neste capítulo, descreveremos e faremos realmente a implantação de nosso site em um servidor web live (ao vivo) e real.

Você pode se sentir tentado a pular este capítulo – há muitas informações que podem parecer excessivas, e talvez você ache que não é isso que está procurando. No entanto, eu insisto *fortemente* que você faça uma tentativa. Esta é uma das seções do livro com a

qual estou mais satisfeito, e é uma seção sobre a qual as pessoas me escrevem com frequência dizendo que ficaram realmente felizes por terem insistido nela.

Caso você não tenha feito nenhuma implantação de servidor antes, o capítulo desmistificará um mundo todo para você, e não há nada como a sensação de ver o seu site ao vivo na internet. Dê-lhe um nome da moda, como “DevOps”, se esse for o preço para convencer você de que a tarefa vale a pena.



Por que você não me avisa quando seu site estiver ao vivo na web, enviando-me o URL? Isso sempre me dá uma sensação feliz e agradável...
obeythetestinggoat@gmail.com.

TDD e as áreas perigosas da implantação

Fazer a implantação de um site em um servidor web live pode ser um assunto complicado. “*Mas funciona em minha máquina!*” é uma queixa em tom de consternação, ouvida com frequência.

Algumas das áreas perigosas da implantação incluem:

Arquivos estáticos (CSS, JavaScript, imagens, etc.)

Os servidores web geralmente precisam de uma configuração especial para servir esses dados.

O banco de dados

Pode haver problemas de permissões e de path, e devemos ter cuidado quanto à preservação dos dados entre as implantações.

Dependências

Devemos garantir que os pacotes dos quais nosso software depende estejam instalados no servidor e que tenham as versões corretas.

Contudo, há soluções para tudo isso. São elas, na sequência:

- Usar um *site de staging*, na mesma infraestrutura do site de

produção, pode nos ajudar a testar nossas implantações e deixar tudo funcionando corretamente antes de irmos para o site “real”.

- Também podemos *executar nossos testes funcionais no site de staging*. Isso nos deixará mais confiantes de que temos o código e os pacotes corretos no servidor, e, pelo fato de termos agora um “teste de fumaça” (smoke test) para o layout de nosso site, saberemos que o CSS será carregado corretamente.
- Assim como em nosso próprio computador, um *virtualenv* é útil no servidor para administrar pacotes e dependências caso você possa executar mais de uma aplicação Python.
- Por fim, *automação, automação, automação*. Ao usar um script automatizado para a implantação de novas versões, e usar o mesmo script para a implantação nos ambientes de staging e de produção, podemos nos sentir mais seguros de que o ambiente de staging é o mais parecido possível com o ambiente de produção.¹

Nas próximas páginas, descreverei *um* procedimento de implantação. Ele não tem o intuito de ser um procedimento de implantação *perfeito*, portanto, por favor, não o tome como a melhor prática ou uma recomendação – o objetivo é servir de ilustração para mostrar os tipos de problemas envolvidos na implantação, e em que ponto os testes se enquadram.

Visão geral dos capítulos sobre implantação

Há muitas informações nos próximos três capítulos, portanto eis uma visão geral para ajudar você a manter o controle da situação:

Neste capítulo: deixar o sistema ativo e executando

- Adaptar nossos FTs para que possam executar em um servidor de staging.
- Iniciar um servidor, instalar todo o software necessário aí e apontar nossos domínios de staging e do site live para ele.
- Fazer upload de nosso código para o servidor usando o Git.
- Obter e testar uma versão rápida e grosseira de nosso site executando no domínio de staging, usando o servidor de desenvolvimento do Django.

- Configurar manualmente um virtualenv no servidor (sem virtualenvwrapper) e garantir que o banco de dados e os arquivos estáticos estejam funcionando.
- À medida que prosseguirmos, vamos continuar executando nosso FT para saber o que está funcionando e o que não está.

Próximo capítulo: passando para uma configuração pronta para produção

- Passar de nossa versão rápida e grosseira para uma configuração pronta para produção; parar de usar o servidor de desenvolvimento do Django, configurar nossa aplicação para iniciar automaticamente no boot, configurar DEBUG com False, e assim por diante.

Terceiro capítulo de implantação: automatizar a implantação

1. Depois que tivermos uma configuração funcional, escreveremos um script para automatizar o processo que acabamos de executar manualmente; desse modo, poderemos implantar nosso site automaticamente no futuro.
2. Por fim, usaremos esse script para implantar a versão de produção de nosso site em seu domínio real.

Como sempre, comece com um teste

Vamos adaptar um pouco os nossos testes funcionais para que sejam executados em um site de staging. Faremos isso por meio de um pequeno hacking em um argumento que geralmente é usado para modificar o endereço em que o servidor temporário de testes é executado:

functional_tests/tests.py (ch08l001)

```
import os
[...]
```

```
class NewVisitorTest(StaticLiveServerTestCase):
```

```
    def setUp(self):
        self.browser = webdriver.Firefox()
        staging_server = os.environ.get('STAGING_SERVER') ❶
        if staging_server:
            self.live_server_url = 'http://' + staging_server ❷
```

Você se lembra que eu disse que `LiveServerTestCase` tinha certas limitações? Bem, uma delas é que ela sempre supõe que você quer usar seu próprio servidor de testes, que ele disponibiliza em `self.live_server_url`. Continuo querendo ter a possibilidade de fazer isso às vezes, mas também quero poder lhe dizer seletivamente que não se importe com isso e use um servidor real.

- ❶ O modo como decidi fazer isso é usando uma variável de ambiente chamada `STAGING_SERVER`.
- ❷ Eis o hack: substituímos `self.live_server_url` com o endereço de nosso servidor “real”.

Testamos se o mencionado hack não causou problemas em nada executando os testes funcionais “normalmente”:

```
$ python manage.py test functional_tests
[...]
Ran 3 tests in 8.544s
```

OK

Agora podemos testá-los no URL de nosso servidor de staging. Planejo hospedar meu servidor de staging em *superlists-staging.ottg.eu*:

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test functional_tests
```

```
=====
=====
FAIL: test_can_start_a_list_for_one_user
(functional_tests.tests.NewVisitorTest)
-----
Traceback (most recent call last):
  File "../superlists/functional_tests/tests.py", line 49, in
test_can_start_a_list_and_retrieve_it_later
    self.assertIn('To-Do', self.browser.title)
AssertionError: 'To-Do' not found in 'Domain name registration | Domain names
| Web Hosting | 123-reg'
[...]
```



```
=====
=====
```

```
FAIL: test_multiple_users_can_start_lists_at_different_urls
(functional_tests.tests.NewVisitorTest)
```

```
-----
Traceback (most recent call last):
```

```
File
"/.../superlists/functional_tests/tests.py", line 86, in
test_layout_and_styling
    inputbox = self.browser.find_element_by_id('id_new_item')
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate
element: [id="id_new_item"]
[...]
```

```
=====
=====
```

```
FAIL: test_layout_and_styling (functional_tests.tests.NewVisitorTest)
```

```
-----
Traceback (most recent call last):
```

```
File
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate
element: [id="id_new_item"]
[...]
```

```
Ran 3 tests in 19.480s:
```

```
FAILED (failures=3)
```



No Windows, se você vir um erro informando algo como "STAGING_SERVER is not recognized as a command" (STAGING_SERVER não é reconhecido como um comando), provavelmente é porque você não está usando o Git-Bash. Consulte novamente a seção "Pré-requisitos e suposições".

Podemos ver que os dois testes estão falhando conforme esperado,

pois, na verdade, ainda não configurei meu domínio. De fato, podemos ver, com base no primeiro traceback, que o teste está terminando na página inicial do agente de registro (registrar) de meu domínio.

O FT, porém, parece testar os aspectos corretos, portanto vamos fazer um commit:

```
$ git diff # deve mostrar alterações em functional_tests.py  
$ git commit -am "Hack FT runner to be able to test staging"
```



Não use `export` para configurar a variável de ambiente `STAGING_SERVER`; do contrário, todas as suas execuções subsequentes de testes nesse terminal serão feitas no staging (e isso poderá ser muito confuso se você não estiver esperando). Configurá-lo explicitamente inline sempre que você executar o FT é melhor.

Obtendo um nome de domínio

Vamos precisar de dois nomes de domínio neste ponto do livro – ambos podem ser subdomínios de um único domínio. Usarei *superlists.ottg.eu* e *superlists-staging.ottg.eu*. Caso você ainda não seja dono de um domínio, é hora de registrar um! Novamente, é algo que de fato quero que você faça *de verdade*. Caso não tenha registrado um domínio antes, basta escolher qualquer agente de registro e comprar um domínio barato – deve custar apenas cinco dólares ou algo assim, mas você pode até mesmo encontrar alguns gratuitos. Prometo que ver seu trabalho em um site “de verdade” será empolgante.

Provisionamento manual de um servidor para hospedar o nosso site

Podemos separar a “implantação” em duas tarefas:

- *provisionamento* (provisioning) de um novo servidor para ser capaz de hospedar o código;

- *implantação* (deploying) de uma nova versão do código em um servidor existente.

Algumas pessoas gostam de usar um servidor totalmente novo a cada implantação – é o que fazemos na PythonAnywhere. Contudo, isso é necessário somente para sites maiores e mais complexos, ou para alterações significativas em um site existente. Para um site simples como o nosso, faz sentido separar as duas tarefas. Além disso, embora desejemos que, em algum momento, as duas tarefas sejam totalmente automatizadas, provavelmente poderemos conviver com um sistema de provisionamento manual por enquanto.

À medida que ler este capítulo, você deverá estar ciente de que o provisionamento é algo que varia bastante e que, como resultado, há poucas boas práticas universais para a implantação. Então, em vez de tentar se lembrar das especificidades do que estou fazendo aqui, você deve tentar compreender o raciocínio para que possa aplicar o mesmo tipo de ideias nas circunstâncias específicas que você encontrar no futuro.

Escolhendo o lugar para hospedar o nosso site

Há diversas soluções distintas por aí atualmente, mas, de modo geral, elas se enquadram em dois campos:

- executar o seu próprio servidor (possivelmente virtual);
- usar uma oferta de PaaS (Platform-As-A-Service, ou Plataforma como Serviço), como Heroku, OpenShift ou PythonAnywhere.

No caso particular de sites pequenos, um PaaS oferece muitas vantagens e, definitivamente, recomendo que você dê uma olhada neles. Entretanto, por diversos motivos, não usaremos um PaaS neste livro. Em primeiro lugar, tenho um conflito de interesses, pois acho que o PythonAnywhere é o melhor, mas, novamente, eu diria isso porque trabalho lá. Em segundo lugar, todas as ofertas de PaaS são bem diferentes, e os procedimentos para implantação em cada um deles variam bastante – conhecer um não necessariamente diz

algo sobre os outros. Qualquer um deles pode modificar seu processo de forma radical ou poderá apenas estar fora do mercado quando você estiver lendo este livro.

Em vez de fazer isso, conheceremos apenas um pouquinho da boa e velha administração de servidores, incluindo SSH e configuração de servidor web. É improvável que essas tarefas vão desaparecer algum dia, e saber um pouco sobre o assunto fará com que você conquiste certo respeito de todos os velhos dinossauros por aí.

O que eu fiz foi tentar configurar um servidor de modo que seja bem semelhante ao ambiente que você terá com um PaaS; assim, você deverá ser capaz de aplicar as lições aprendidas na seção de implantação, independentemente da solução de provisionamento que escolher.

Iniciando um servidor

Não vou descrever exatamente como fazer isso – independentemente de você escolher Amazon AWS, Rackspace, Digital Ocean, seu próprio servidor em seu próprio data center ou um Raspberry Pi em um armário sob as escadas, qualquer solução será apropriada, desde que:

- seu servidor esteja executando Ubuntu 16.04 (também conhecido como “Xenial/LTS”);
- você tenha acesso de root a ele;
- está na internet pública;
- você pode acessá-lo via SSH.

Recomendo o Ubuntu como uma distro porque é fácil ter o Python 3.6 aí e ele possui algumas maneiras específicas de configurar o Nginx, das quais farei uso a seguir. Se você souber o que está fazendo, provavelmente poderá achar outra solução diferente, mas estará por sua conta e risco.

Caso não tenha iniciado um servidor Linux antes e não tenha a mínima ideia de por onde começar, escrevi um guia bem rápido no

GitHub

(<https://github.com/hjwp/Book-TDD-Web-Dev-Python/blob/master/server-quickstart.md>).



Algumas pessoas chegam a este capítulo e se sentem tentadas a pular a parte sobre o domínio e sobre “obter um servidor de verdade”, e simplesmente usam uma VM em seu próprio computador. Não faça isso. **Não** é a mesma coisa, e você terá mais dificuldade em seguir as instruções, que já são suficientemente complicadas como são. Se estiver preocupado com o custo, pesquise por aí e você encontrará opções gratuitas para ambos. Envie-me um email se precisar de outras referências; sempre fico satisfeito em ajudar.

Contas de usuário, SSH e privilégios

Nessas instruções, estou supondo que você tenha configurado uma conta de usuário diferente de root, com privilégios “sudo”; assim, sempre que precisar fazer algo que exija acesso de root, usaremos sudo, e deixarei isso explícito nas várias instruções apresentadas a seguir.

Meu usuário se chama “elspeth”, mas você pode dar o nome que quiser ao seu!

Instalando o Nginx

Precisaremos de um servidor web, e todo o pessoal interessante está usando o Nginx atualmente, portanto faremos isso também. Depois de ter lutado por muitos anos com o Apache, posso dizer que é um alívio e uma bênção, pelo menos quanto à legibilidade de seus arquivos de configuração!

Instalar o Nginx em meu servidor foi uma questão de executar um apt-get, e então pude ver a tela default “Hello World” do Nginx:

```
elspeth@server:$ sudo apt-get install nginx
elspeth@server:$ sudo systemctl start nginx
```

(Talvez você precise executar um apt-get update e/ou um apt-get upgrade antes.)



Preste atenção no `elspeth@server` nas listagens de linha de comando neste capítulo. Ele indica comandos que devem ser executados no servidor, em oposição a comandos executados em seu próprio computador.

Você deverá ser capaz de acessar o endereço IP de seu servidor e ver a página “Welcome to nginx” (Bem-vindo ao nginx) nesse ponto, como vemos na Figura 9.1.



Se você não vir essa página, talvez seja porque o seu firewall não abra a porta 80 para o mundo. No AWS, por exemplo, pode ser que seja necessário configurar o “security group” (grupo de segurança) para o seu servidor abrir a porta 80.

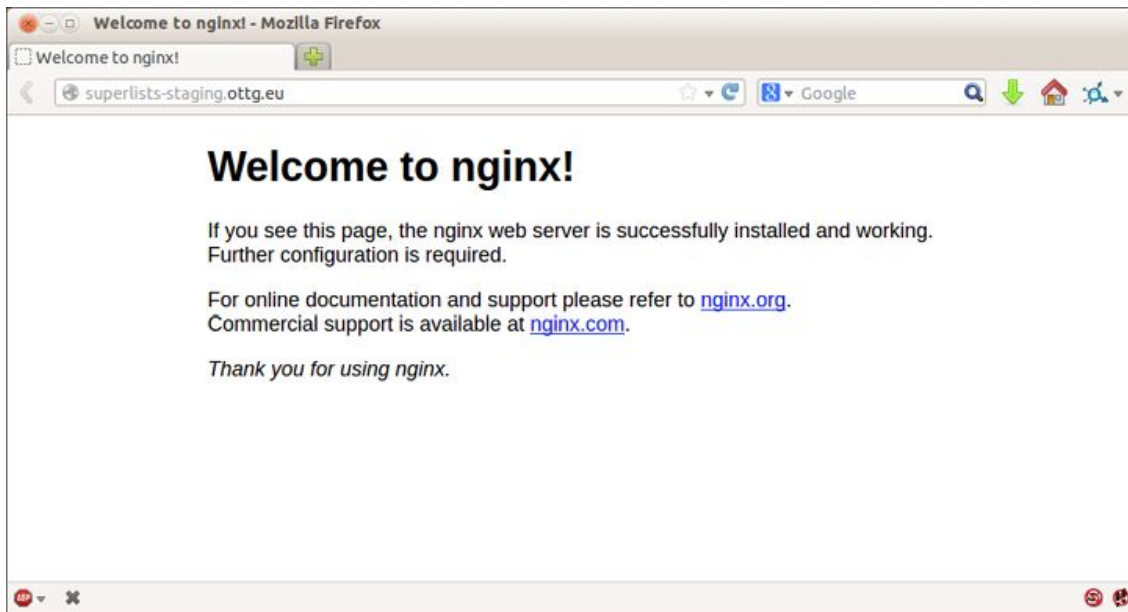


Figure 9.1 – Nginx – funcionando!

Instalando o Python 3.6

O Python 3.6 não estava disponível nos repositórios-padrões do Ubuntu quando escrevi este livro, mas o “Deadsnakes PPA” (<https://launchpad.net/~fkrull/+archive/ubuntu/deadsnakes>), com a contribuição de usuários, tem essa versão. Apresentaremos a seguir o modo de instalá-lo.

Com o acesso de root, vamos garantir que o servidor tenha os softwares essenciais de que precisamos no nível do sistema: Python, Git, pip e virtualenv.

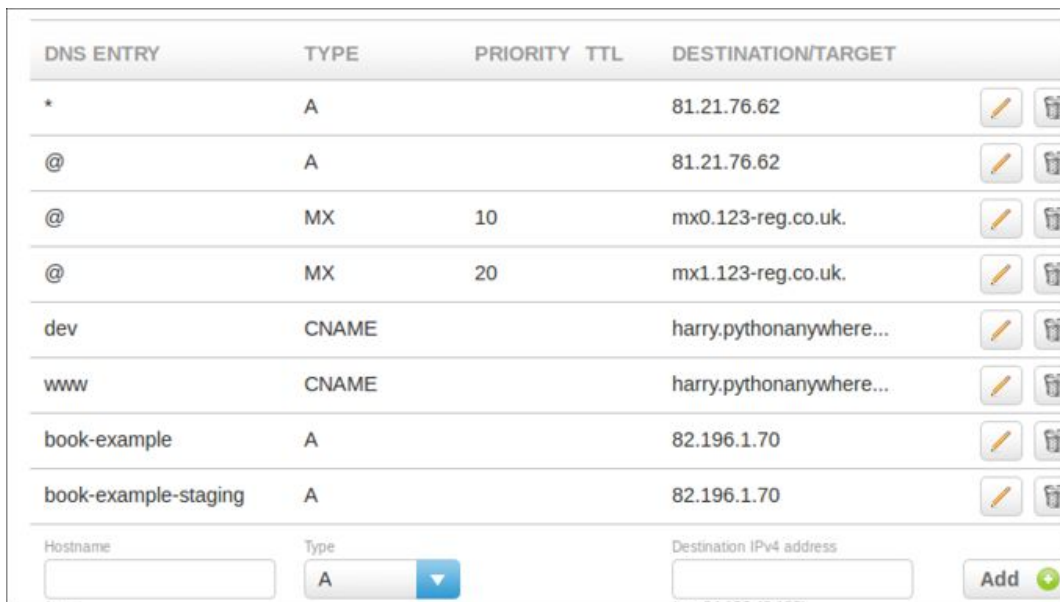
```
elspeth@server:$ sudo add-apt-repository ppa:fkruill/deadsnakes
elspeth@server:$ sudo apt-get update
elspeth@server:$ sudo apt-get install python3.6 python3.6-venv
```

















Aproveitando a ocasião, vamos garantir também que o Git esteja instalado.

```
elspeth@server:$ sudo apt-get install git
```

Configurando domínios para o ambiente de staging e o ambiente live

Não queremos ficar lidando com endereços IP o tempo todo, portanto devemos apontar os domínios de staging e do ambiente live para o servidor. Em meu agente de registro (registrar), as telas de controle tinham uma aparência semelhante àquela mostrada na Figura 9.2.



DNS ENTRY	TYPE	PRIORITY	TTL	DESTINATION/TARGET	
*	A			81.21.76.62	 
@	A			81.21.76.62	 
@	MX	10		mx0.123-reg.co.uk.	 
@	MX	20		mx1.123-reg.co.uk.	 
dev	CNAME			harry.pythonanywhere...	 
www	CNAME			harry.pythonanywhere...	 
book-example	A			82.196.1.70	 
book-example-staging	A			82.196.1.70	 



Hostname: Type: A  Destination IPv4 address: Add 

Figura 9.2 – Configuração do domínio.

No sistema de DNS, apontar um domínio para um endereço IP específico é chamado de “A-Record” (Registro A). Todo agente de

registro é um pouco diferente, mas alguns cliques aqui e ali deverão levar você até a tela correta em seu agente.

Usando o FT para confirmar que o domínio funciona e que o Nginx está executando

Para essa confirmação, podemos executar novamente nossos testes funcionais e ver se suas mensagens de falha mudaram um pouco – uma delas em particular deve agora mencionar o Nginx:

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test
functional_tests
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate
element: [id="id_new_item"]
[...]
AssertionError: 'To-Do' not found in 'Welcome to nginx!'
```

Progresso! Dê a si mesmo um tapinha nas costas e, quem sabe, uma boa xícara de café e um biscoito de chocolate (https://en.wikipedia.org/wiki/Digestive_biscuit).

Implantando o nosso código manualmente

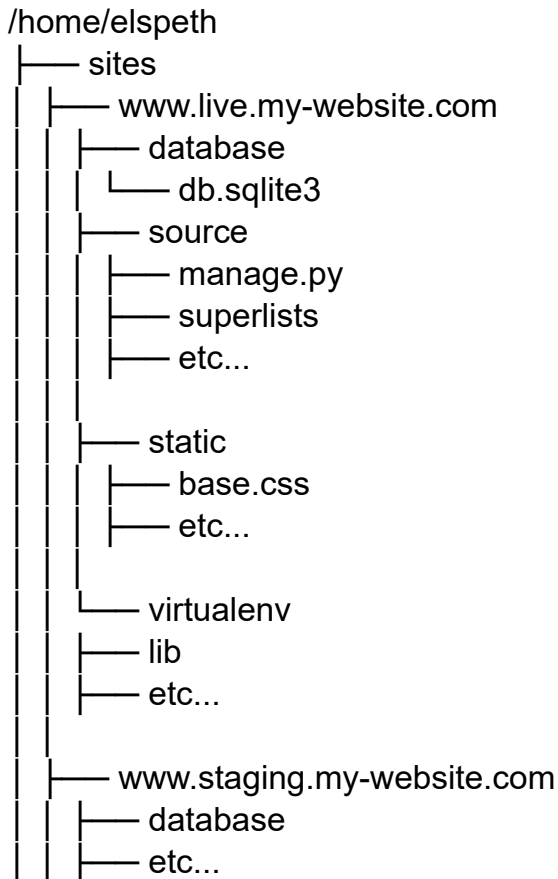
O próximo passo é fazer com que uma cópia do site de staging esteja no ar e executando, somente para ver se podemos fazer o Nginx e o Django conversarem um com o outro. À medida que fizermos isso, começaremos a sair do provisionamento e a entrar na “implantação”, portanto devemos pensar em como poderemos automatizar o processo enquanto o executamos.



Uma regra geral para distinguir provisionamento de implantação é que há uma tendência de precisarmos de permissões de root para o primeiro, mas não para o último.

Precisamos de um diretório para o código-fonte. Vamos colocá-lo em algum lugar na pasta home de nosso usuário não root; no meu

caso, seria em `/home/elspeth` (provavelmente essa será a configuração em qualquer sistema de hosting compartilhado, mas, de qualquer modo, você sempre deve executar suas aplicações web com um usuário diferente de root). Configurarei meus sites assim:



Cada site (staging, live ou qualquer outro) tem sua própria pasta. Dentro deles, temos uma pasta separada para o código-fonte, o banco de dados e os arquivos estáticos. A lógica é que, embora o código-fonte possa mudar de uma versão do site para a próxima, o banco de dados permanecerá o mesmo. A pasta estática está na mesma localidade relativa, `../static`, que configuramos no final do último capítulo. Por fim, o `virtualenv` tem também a sua própria subpasta (no servidor, não há necessidade de usar `virtualenvwrapper`; criaremos um `virtualenv` manualmente).

Acertando a localização do banco de dados

Inicialmente vamos alterar a localização de nosso banco de dados

em *settings.py* e garantir que possamos fazê-lo funcionar em nosso computador local:

superlists/settings.py (ch08l003)

```
# Build paths inside the project like this: os.path.join(BASE_DIR, ...)
import os
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
[...]

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, '../database/db.sqlite3'),
    }
}
```



Dê uma olhada no modo como `BASE_DIR` está definido, mais no início de *settings.py*. Observe que `abspath` vem antes (isto é, o mais interno). Sempre siga esse padrão quando estiver lidando com paths; caso contrário, você poderá ver coisas estranhas acontecendo, conforme o modo como o arquivo é importado. Obrigado a Green Nathan (<https://github.com/CleanCut/green>) por essa dica!

Vamos agora testar localmente:

```
$ mkdir ../database
$ python manage.py migrate --noinput
Operations to perform:
Apply all migrations: auth, contenttypes, lists, sessions
Running migrations:
[...]
$ ls ../database/
db.sqlite3
```

Parece que funciona. Vamos fazer um commit:

```
$ git diff # deve mostrar alterações em settings.py
$ git commit -am "move sqlite database outside of main source tree"
```

Para termos nosso código no servidor, usaremos o Git e um dos sites de compartilhamento de código. Se você ainda não o fez, envie seu código para o GitHub, o BitBucket ou algo semelhante. Todos

eles têm instruções excelentes para iniciantes acerca de como fazer isso.

A seguir, apresentamos alguns comandos bash que farão a configuração de tudo isso. Caso não tenha familiaridade com eles, observe o comando `export` que me permite configurar uma “variável local” no bash:

```
elspeth@server:$ export SITENAME=superlists-staging.ottg.eu
elspeth@server:$ mkdir -p ~/sites/$SITENAME/database
elspeth@server:$ mkdir -p ~/sites/$SITENAME/static
elspeth@server:$ mkdir -p ~/sites/$SITENAME/virtualenv
# você deve substituir o URL da próxima linha pelo URL de seu próprio
repositório
elspeth@server:$ git clone https://github.com/hjwp/book-example.git \
~/sites/$SITENAME/source
Resolving deltas: 100% [...]
```



Uma variável bash definida com `export` dura apenas o tempo da sessão de console. Se você fizer `logout` do servidor e `login` novamente, será necessário redefini-la. Pode ser confuso, pois o Bash não mostrará um erro, mas simplesmente substituirá a variável com a string vazia, o que levará a resultados estranhos... se estiver em dúvida, execute um rápido `echo $SITENAME`.

Agora que temos o site instalado, vamos tentar executar o servidor de desenvolvimento – esse é um teste de fumaça para ver se todas as partes estão conectadas:

```
elspeth@server:$ $ cd ~/sites/$SITENAME/source
$ python manage.py runserver
Traceback (most recent call last):
  File "manage.py", line 8, in <module>
    from django.core.management import execute_from_command_line
ImportError: No module named django.core.management
```

Ah. O Django não está instalado no servidor.

Criando um virtualenv manualmente e usando requirements.txt

Para “salvar” a lista de pacotes de que precisamos em nosso virtualenv, e poderemos recriá-la no servidor, devemos criar um arquivo *requirements.txt*:

```
$ echo "django==1.11" > requirements.txt
$ git add requirements.txt
$ git commit -m "Add requirements.txt for virtualenv"
```



Talvez você esteja se perguntando por que não acrescentamos nossa outra dependência, o Selenium, aos nossos requisitos. Isso se deve ao fato de o Selenium ser uma dependência somente para os testes, e não para o código da aplicação. Algumas pessoas também gostam de criar um arquivo chamado *test-requirements.txt*.

Agora executamos um git push para enviar as atualizações ao nosso site de compartilhamento de código:

```
$ git push
```

Então podemos descer essas alterações para o servidor:

```
elspeth@server:$ git pull # talvez peça para você fazer algumas configuração
# no git antes
```

Criar um virtualenv “manualmente” (isto é, sem virtualenvwrapper) envolve usar o módulo “venv” da biblioteca-padrão e especificar o path no qual você quer colocar o virtualenv:

```
elspeth@server:$ pwd
/home/elspeth/sites/staging.superlists.com/source
elspeth@server:$ python3.6 -m venv ../virtualenv
elspeth@server:$ ls ../virtualenv/bin
activate activate.fish easy_install-3.6 pip3 python
activate.csh easy_install pip pip3.6 python3
```

Se quiséssemos ativar o virtualenv, poderíamos fazê-lo com `source ../virtualenv/bin/activate`, mas isso não é necessário. Podemos, na verdade, fazer tudo que queremos chamando as versões de Python, pip e outros executáveis no diretório *bin* de virtualenv, conforme veremos.

Para instalar nossos requisitos no virtualenv, usamos o pip do virtualenv:

```
elspeth@server:$ ../virtualenv/bin/pip install -r requirements.txt
Downloading/unpacking Django==1.11 (from -r requirements.txt (line 1))
[...]
Successfully installed Django
```

Para executar o Python no virtualenv, utilize o binário python do virtualenv:

```
elspeth@server:$ ../virtualenv/bin/python manage.py runserver
Validating models...
0 errors found
[...]
```



Conforme a configuração de seu firewall, talvez você possa até mesmo acessar o seu site manualmente neste ponto. Será necessário executar `runserver 0.0.0.0:8000` para ouvir o endereço IP público bem como o privado, e então acessar `http://your.domain.com:8000`.

Parece que tudo está executando satisfatoriamente. Por ora, podemos teclar Ctrl-C.

Mais progressos! Temos um sistema para obter e enviar código ao servidor (git push e git pull), e temos um virtualenv configurado para que corresponda ao nosso ambiente local, além de um único arquivo, *requirements.txt*, para mantê-los sincronizados.

Em seguida, vamos configurar o servidor web Nginx, para que converse com o Django, e deixar o nosso site ativo na porta padrão 80.

Configuração simples do Nginx

Vamos criar um arquivo de configuração do Nginx para lhe dizer que envie requisições ao nosso site de staging com o Django. Eis a aparência de uma configuração mínima:

```
server: /etc/nginx/sites-available/superlists-
staging.ottg.eu
```

```
server {
    listen 80;
```

```
server_name superlists-staging.ottg.eu;

location / {
    proxy_pass http://localhost:8000;
}
}
```

Essa configuração informa que apenas o nosso domínio de staging será ouvido, e será feito um “proxy” de todas as requisições para a porta local 8000, onde se espera que o Django seja encontrado à espera para responder.

Salvei essa configuração em um arquivo chamado *superlists-staging.ottg.eu* na pasta */etc/nginx/sites-available*.



Você tem dúvidas sobre como editar um arquivo no servidor? Sempre temos o vi, que incentivo você a conhecer um pouco, mas hoje talvez já seja um dia muito cheio de novidades para isso. Experimente usar o nano (<http://www.howtogeek.com/howto/42980/the-beginners-guide-to-nano-the-linux-command-line-text-editor/>), que é relativamente mais agradável para os iniciantes. Observe que você também precisará usar sudo porque o arquivo está em uma pasta do sistema.

Então adicionamos o arquivo nos sites permitidos ao servidor, criando um link simbólico (symlink) para ele:

```
elspeth@server:$ echo $SITENAME # verifica se ainda contém o nosso site
                  # superlists-staging.ottg.eu
elspeth@server:$ sudo ln -s ../sites-available/$SITENAME /etc/nginx/sites-
enabled/$SITENAME
elspeth@server:$ ls -l /etc/nginx/sites-enabled # verifica se nosso link
                  # simbólico está aí
```

Essa é a maneira preferida do Debian/Ubuntu para salvar configurações de Nginx – o verdadeiro arquivo de configuração em *sites-available* e um link simbólico em *sites-enabled*; a ideia é que isso facilita ativar ou desativar os sites.

Também podemos muito bem remover a configuração default “Welcome to nginx” a fim de evitar qualquer confusão:

```
elspeth@server:$ sudo rm /etc/nginx/sites-enabled/default
```

Vamos testar agora:

```
elspeth@server:$ sudo systemctl reload nginx
elspeth@server:$ ../virtualenv/bin/python manage.py runserver
```



Também tive que editar `/etc/nginx/nginx.conf` e remover o caractere de comentário de uma linha contendo `server_names_hash_bucket_size 64`; para que meu nome de domínio longo pudesse funcionar. Talvez você não tenha esse problema; o Nginx avisará quando você fizer um reload caso haja algum problema com seus arquivos de configuração.

Uma rápida inspeção visual confirma: o site está no ar (Figura 9.3)!

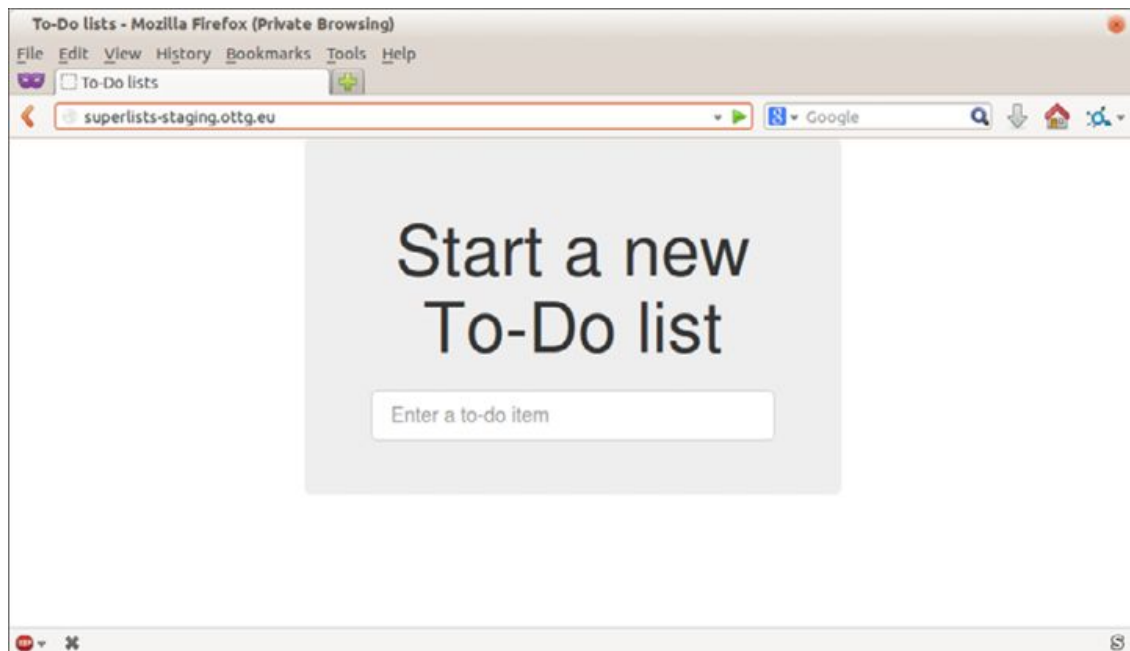


Figura 9.3 – O site de staging está no ar!



Se, em algum momento, você achar que o Nginx não está se comportando conforme esperado, experimente executar o comando `sudo nginx -t`, que faz um teste de configuração, e você será avisado se houver algum problema em seus arquivos de configuração.

Vamos ver o que dizem os nossos testes funcionais:

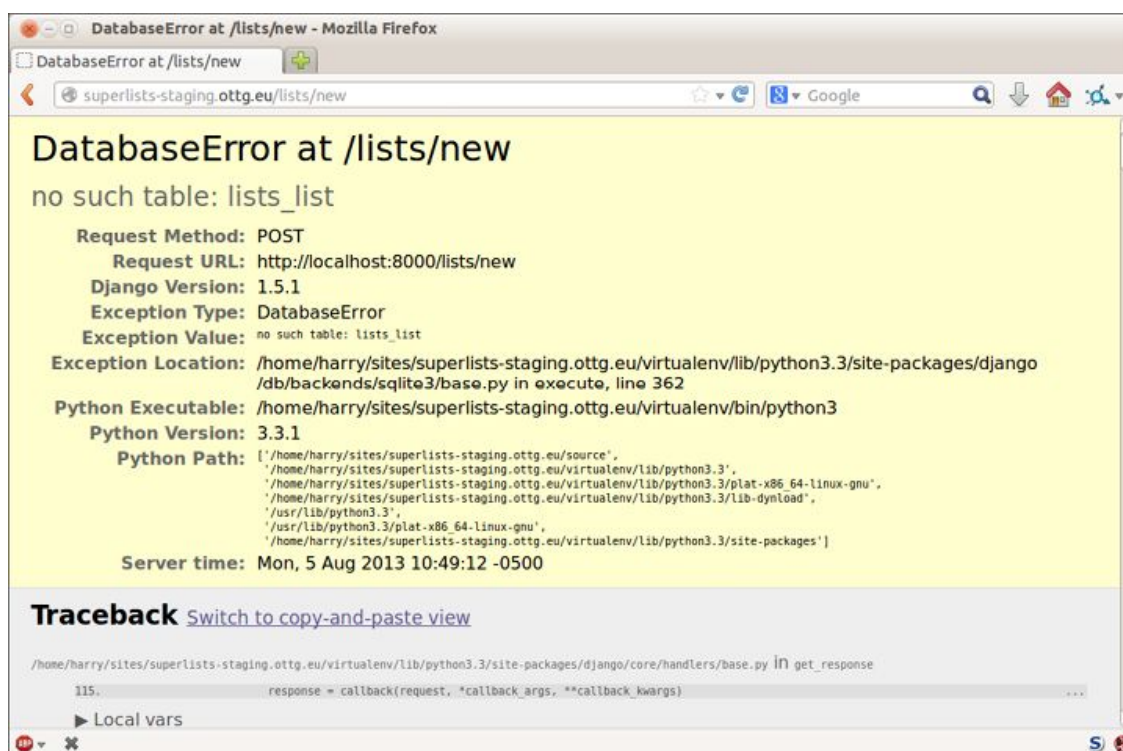
```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test
functional_tests
[...]
```

selenium.common.exceptions.NoSuchElementException: Message: Unable to locate

[...]

AssertionError: 0.0 != 512 within 3 delta

Os testes falham assim que tentam submeter um novo item, pois ainda não configuramos o banco de dados. Provavelmente você deve ter percebido a página de depuração amarela do Django (Figura 9.4) nos informando até que ponto os testes chegaram, ou se você tentou manualmente.



```
DatabaseError at /lists/new
no such table: lists_list

Request Method: POST
Request URL: http://localhost:8000/lists/new
Django Version: 1.5.1
Exception Type: DatabaseError
Exception Value: no such table: lists_list
Exception Location: /home/harry/sites/superlists-staging.ottg.eu/virtualenv/lib/python3.3/site-packages/django/db/backends/sqlite3/base.py in execute, line 362
Python Executable: /home/harry/sites/superlists-staging.ottg.eu/virtualenv/bin/python3
Python Version: 3.3.1
Python Path: ['/home/harry/sites/superlists-staging.ottg.eu/source',
'/home/harry/sites/superlists-staging.ottg.eu/virtualenv/lib/python3.3',
'/home/harry/sites/superlists-staging.ottg.eu/virtualenv/lib/python3.3/plat-x86_64-linux-gnu',
'/home/harry/sites/superlists-staging.ottg.eu/virtualenv/lib/python3.3/lib-dynload',
'/usr/lib/python3.3',
'/usr/lib/python3.3/plat-x86_64-linux-gnu',
'/home/harry/sites/superlists-staging.ottg.eu/virtualenv/lib/python3.3/site-packages']

Server time: Mon, 5 Aug 2013 10:49:12 -0500

Traceback Switch to copy-and-paste view

/home/harry/sites/superlists-staging.ottg.eu/virtualenv/lib/python3.3/site-packages/django/core/handlers/base.py in get_response
115.         response = callback(request, *callback_args, **callback_kwargs)
▶ Local vars
```

Figura 9.4 – O banco de dados, porém, ainda não está pronto.



Os testes evitaram que possivelmente passássemos vergonha nesse caso. O site *parecia* não ter problemas quando carregamos a página frontal. Se tivéssemos sido um pouco apressados demais, poderíamos achar que havíamos terminado e teriam sido os primeiros usuários que descobririam aquela página amarela desagradável de DEBUG do Django. Tudo bem, estou sendo um pouco exagerado para causar efeito; talvez *tivéssemos* verificado, mas o que acontecerá à medida que o site se tornar cada vez maior e mais complexo? Você não é capaz de verificar tudo. Os testes são.

Criando o banco de dados com migrate

Executamos migrate usando o argumento `--noinput` a fim de eliminar os dois pequenos prompts “are you sure” (você tem certeza):

```
elspeth@server:$ ../virtualenv/bin/python manage.py migrate --noinput
Creating tables ...
[...]
elspeth@server:$ ls ../database/
db.sqlite3
elspeth@server:$ ../virtualenv/bin/python manage.py runserver
```

Vamos experimentar os FTs novamente:

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test
functional_tests
[...]
```

...

```
-----
Ran 3 tests in 10.718s
```

OK

É ótimo ver que o site está no ar e executando! A essa altura, podemos nos recompensar com um intervalo merecido para um café, antes de passarmos para a próxima seção...



Se você vir um “502 - Bad Gateway”, provavelmente é porque você se esqueceu de reiniciar o servidor de desenvolvimento com `manage.py runserver` após o `migrate`.

Há mais algumas dicas para depuração na caixa de texto que apresentamos a seguir.

Dicas para depuração do servidor

Implantações são capciosas! Se o processo não ocorrer exatamente conforme esperado, eis algumas dicas e aspectos nos quais você deve prestar atenção:

- Tenho certeza de que você já fez isso, mas verifique com atenção se cada arquivo está exatamente no lugar em que deveria e se seu conteúdo está

correto – um único caractere perdido pode fazer toda a diferença.

- Os logs de erro do Nginx estão em `/var/log/nginx/error.log`.
- Você pode pedir ao Nginx que “verifique” sua configuração usando a flag `-t`:
`nginx -t`.
- Garanta que seu navegador não esteja fazendo caching de uma resposta desatualizada. Utilize Ctrl-Refresh ou inicie uma nova janela de navegador privada.
- Pode parecer uma tentativa desesperada, mas, às vezes, já vi comportamentos inexplicáveis no servidor que só foram resolvidos quando eu o reiniciei totalmente com um `sudo reboot`.

Se você se vir totalmente sem saber o que fazer, há sempre a opção de acabar com o seu servidor e começar novamente do zero! A segunda vez deverá ser mais rápida...

Sucesso! Nossa implantação hack funciona

Ufa. Supondo que você tenha conseguido deixar o site no ar e executando, no mínimo temos certeza de que a engrenagem básica funciona, mas não podemos usar o servidor de desenvolvimento do Django em produção. Também não podemos depender de iniciar o servidor manualmente com `runserver`. No próximo capítulo, deixaremos nossa implantação hack mais preparada para o ambiente de produção.

Testando a configuração e a implantação do servidor

Os testes eliminam parte da incerteza da implantação.

Para os desenvolvedores, a administração do servidor é sempre “divertida” e, com isso, quero dizer que é um processo cheio de incertezas e surpresas. Meu objetivo durante este capítulo foi mostrar que uma suíte de testes funcionais pode eliminar parte da incerteza do processo.

Pontos complicados típicos – banco de dados, arquivos estáticos, dependências, configurações personalizadas

Os itens em que você precisa ficar de olho em qualquer implantação incluem a configuração de seu banco de dados, arquivos estáticos, dependências de software e configurações personalizadas que diferem entre os ambientes de desenvolvimento e de produção. Você precisará pensar em cada um deles em suas próprias implantações.

Os testes nos permitem fazer experimentos

Sempre que fizermos uma alteração na configuração de nosso servidor, podemos executar a suíte de testes novamente e ter confiança de que tudo está funcionando tão bem quanto antes. Isso nos permite fazer experimentos com nossa configuração sem tanto temor (como veremos no próximo capítulo).

1 O que estou chamando de servidor de “staging” algumas pessoas chamariam de servidor de “desenvolvimento”, enquanto outras também vão querer distinguir os servidores de “pré-produção”. Independentemente de como o chamamos, a questão é ter algum lugar em que possamos testar o nosso código em um ambiente que seja o mais parecido possível com o servidor de produção real.

CAPÍTULO 10

Chegando a uma implantação pronta para produção

Neste capítulo, faremos algumas alterações em nosso site para passarmos para uma configuração mais preparada para produção. À medida que fizermos cada mudança, usaremos os testes para nos dizer se tudo continua funcionando.

O que há de errado com nossa implantação hack? Bem, não podemos usar o servidor de desenvolvimento do Django para produção; ele não foi projetado para cargas da “vida real”. Em seu lugar, usaremos algo chamado Gunicorn para executar nosso código Django, e teremos o Nginx para servir nossos arquivos estáticos.

Atualmente, nosso *settings.py* tem `DEBUG=True`, e isso é fortemente não recomendado em ambiente de produção (por exemplo, você não vai querer seus usuários encarando tracebacks de depuração de seu código quando seu site errar). Também teremos que definir `ALLOWED_HOSTS` para segurança.

Queremos que nosso site inicie automaticamente sempre que o servidor reiniciar. Para isso, criaremos um arquivo de configuração para Systemd.

Por fim, se deixarmos a porta 8000 fixa, não poderemos executar vários sites nesse servidor, portanto passaremos a usar “sockets unix” para comunicação entre Nginx e Django.

Passando a usar o Gunicorn

Você sabe por que o mascote do Django é um pônei? Segundo a história, o Django tem muitos recursos desejados: um ORM, todo tipo de middleware, o site de administração... “O que mais você quer, um pônei?” Bem, Gunicorn quer dizer “Green Unicorn” (Unicórnio verde), o que você iria querer depois, se já tivesse um pônei...

```
elspeth@server:$ ../virtualenv/bin/pip install gunicorn
```

O Gunicorn precisará conhecer um path para um servidor WSGI, que geralmente é uma função chamada `application`. O Django disponibiliza um em `superlists/wsgi.py`:

```
elspeth@server:$ ../virtualenv/bin/gunicorn superlists.wsgi:application
2013-05-27 16:22:01 [10592] [INFO] Starting gunicorn 0.19.6
2013-05-27 16:22:01 [10592] [INFO] Listening at: http://127.0.0.1:8000 (10592)
[...]
```

Se observar o site agora, você verá que o CSS está todo com problemas, como vemos na Figura 10.1.

Se executarmos os testes funcionais, você verá que eles confirmarão que há algo errado. O teste para adicionar itens de lista passa satisfatoriamente, porém o teste para layout e estilização falha. Bom trabalho, testes!

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test
functional_tests
[...]
```

AssertionError: 125.0 != 512 within 3 delta
FAILED (failures=1)

O motivo pelo qual o CSS apresenta falhas é que, embora o servidor de desenvolvimento do Django sirva arquivos estáticos em um passe de mágica para você, o Gunicorn não faz isso. Agora é hora de dizer ao Nginx que o faça.

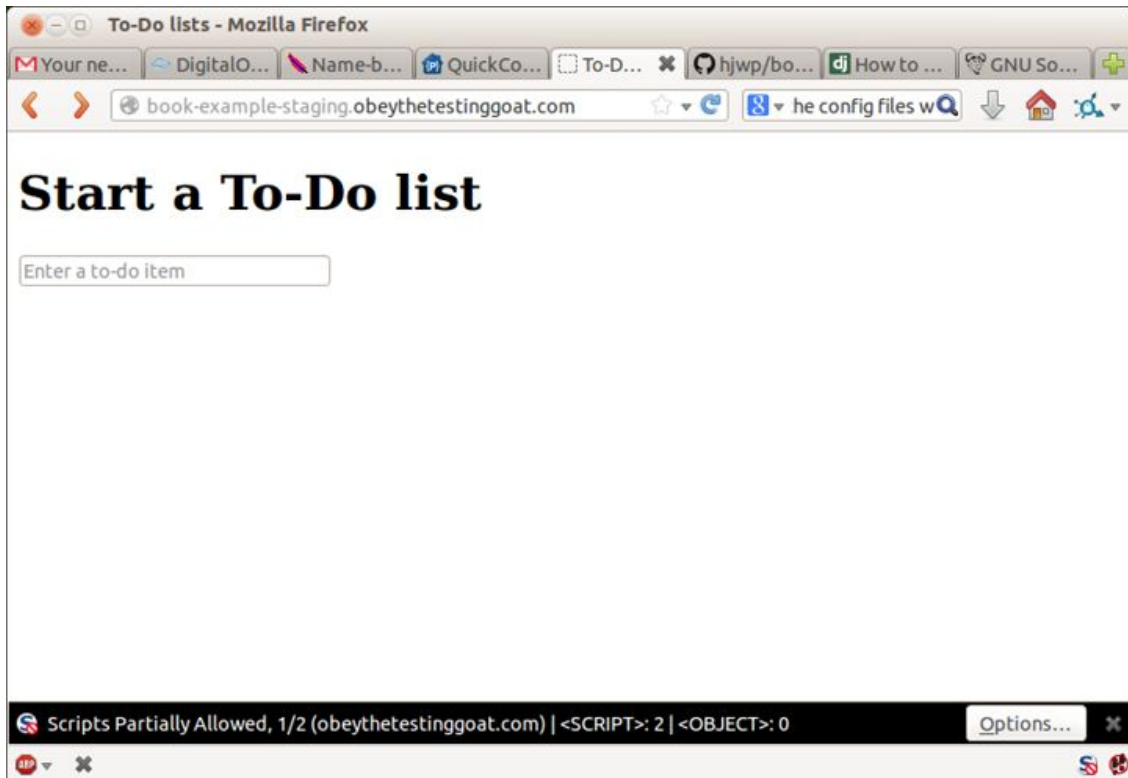


Figura 10.1 – CSS com problema.

Um passo para a frente, outro para trás, mas, pelo menos, os testes estão aí para nos ajudar. Vamos em frente!

Fazendo o Nginx servir arquivos estáticos

Inicialmente executamos `collectstatic` a fim de copiar todos os arquivos estáticos para uma pasta em que o Nginx possa encontrá-los:

```
elspeth@server:~$ ../virtualenv/bin/python manage.py collectstatic --noinput
elspeth@server:~$ ls ../static/
base.css bootstrap
```

Agora, dizemos ao Nginx que comece a servir esses arquivos estáticos para nós:

```
server {
    listen 80;
    server_name superlists-staging.ottg.eu;

    location /static {
```

```
    alias /home/elspeth/sites/superlists-staging.ottg.eu/static;
  }
  location / {
    proxy_pass http://localhost:8000;
  }
}
```

Recarregue o Nginx e reinicie o Gunicorn...

```
elspeth@server:$ sudo systemctl reload nginx
elspeth@server:$ ../virtualenv/bin/gunicorn superlists.wsgi:application
```

Se observarmos o site novamente, sua aparência será muito mais saudável. Podemos executar nossos FTs de novo:

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test
functional_tests
[...]
...
-----
Ran 3 tests in 10.718s
```

OK

Ufa.

Passando a usar sockets Unix

Se quisermos servir dados tanto no ambiente de staging quanto no ambiente live, não podemos ter os dois servidores tentando usar a porta 8000. Poderíamos decidir alocar portas diferentes, mas isso seria um pouco arbitrário, além de ser perigosamente fácil cometer um erro e iniciar o servidor de staging na porta do servidor live, ou vice-versa.

Uma solução melhor é usar sockets de domínio Unix – eles são como arquivos em disco, mas podem ser utilizados pelo Nginx e pelo Gunicorn para conversarem um com o outro. Colocaremos nossos sockets em */tmp*. Vamos alterar as configurações de proxy no Nginx:

```
server:                /etc/nginx/sites-available/superlists-
```

staging.ottg.eu

```
[...]  
    location / {  
        proxy_set_header Host $host;  
        proxy_pass http://unix:/tmp/superlists-staging.ottg.eu.socket;  
    }  
}
```

`proxy_set_header` é usado para garantir que o Gunicorn e o Django saibam em qual domínio estão executando. Precisaremos disso para o recurso de segurança `ALLOWED_HOSTS`, que estamos prestes a usar.

Vamos agora reiniciar o Gunicorn, porém, desta vez, dizendo-lhe que ouça um socket em vez de escutar a porta default:

```
elspeth@server:$ sudo systemctl reload nginx  
elspeth@server:$ ../virtualenv/bin/gunicorn --bind \  
    unix:/tmp/superlists-staging.ottg.eu.socket superlists.wsgi:application
```

Novamente, executamos os testes funcionais para garantir que continuam passando:

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test  
functional_tests  
[...]  
OK
```

Mais alguns passos!

Alterando `DEBUG` para `false` e configurando `ALLOWED_HOSTS`

O modo de `DEBUG` do Django é muito apropriado para hacking em seu próprio servidor, mas deixar todas essas páginas cheias de `tracebacks` disponíveis não é seguro (<http://bit.ly/SuvluV>).

Você encontrará a configuração de `DEBUG` no início de `settings.py`. Quando definimos seu valor com `False`, devemos definir também outro parâmetro chamado `ALLOWED_HOSTS`. Esse valor foi adicionado como um recurso de segurança (<http://bit.ly/2u0R2d6>) no

Django 1.5. Infelizmente ele não tem um comentário prestativo no `settings.py` default, mas podemos acrescentar nossos próprios comentários. Faça o seguinte no servidor:

server: superlists/settings.py

```
# AVISO DE SEGURANÇA: não execute com debug ativado em produção!  
DEBUG = False
```

```
TEMPLATE_DEBUG = DEBUG
```

```
# Necessário quando DEBUG=False  
ALLOWED_HOSTS = ['superlists-staging.ottg.eu']  
[...]
```

Novamente, vamos reiniciar o Gunicorn e executar o FT para verificar se tudo continua funcionando.



Não faça commit dessas alterações no servidor. No momento, isso é apenas um hack para fazer tudo funcionar, e não uma alteração que desejamos manter em nosso repositório. Em geral, para simplificar, farei commits no Git somente a partir do computador local, e usarei git push e git pull quando precisar sincronizá-los com o servidor.

Vamos fazer mais um teste para termos a garantia de que tudo continua funcionando?

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test  
functional_tests  
[...]  
OK
```

Muito bom.

Usando Systemd para garantir que o Gunicorn inicie no boot

Nosso último passo será garantir que o servidor iniciará o Gunicorn automaticamente no boot e o recarregará também de modo automático caso haja falhas. No Ubuntu, o modo de fazer isso é usando o Systemd:

server: /etc/systemd/system/gunicorn-superlists-staging.ottg.eu.service

[Unit]

Description=Gunicorn server for superlists-staging.ottg.eu

[Service]

Restart=on-failure ❶

User=elspeth ❷

WorkingDirectory=/home/elspeth/sites/superlists-staging.ottg.eu/source ❸

ExecStart=/home/elspeth/sites/superlists-staging.ottg.eu/virtualenv/bin/gunicorn \
--bind unix:/tmp/superlists-staging.ottg.eu.socket \
superlists.wsgi:application ❹

[Install]

WantedBy=multi-user.target ❺

O Systemd é agradavelmente simples de configurar (em especial se você já teve o prazer duvidoso de escrever um script init.d) e é razoavelmente autoexplicativo.

- ❶ Restart=on-failure reiniciará o processo automaticamente se houver falhas.
- ❷ User=elspeth faz o processo executar com o usuário “elspeth”.
- ❸ WorkingDirectory define o diretório de trabalho atual.
- ❹ ExecStart é o processo a ser executado. Usamos o caractere \
de continuação de linha para separar o comando completo em várias linhas por questões de legibilidade, mas ele poderia estar todo em uma só linha.
- ❺ WantedBy na seção [Install] é o que diz ao Systemd que queremos que esse serviço inicie no boot.

Os scripts de Systemd ficam em */etc/systemd/system*, e seus nomes devem terminar com *.service*.

Agora, dizemos ao Systemd que inicie o Gunicorn com o comando `systemctl`:

```
# este comando é necessário para dizer ao Systemd que carregue o nosso  
# novo arquivo de configuração
```

```
elspeth@server:$ sudo systemctl daemon-reload  
# este comando diz ao Systemd para sempre carregar o nosso serviço no boot  
elspeth@server:$ sudo systemctl enable gunicorn-superlists-  
staging.ottg.eu  
# este é o comando que inicia o nosso serviço  
elspeth@server:$ sudo systemctl start gunicorn-superlists-staging.ottg.eu
```

(A propósito, você perceberá que o comando `systemctl` responde à tabulação para preenchimento automático, incluindo o nome do serviço.)

Agora, podemos executar novamente os FTs para ver se tudo continua funcionando. Você pode até mesmo testar se o site volta a executar caso reinicie o servidor!

Mais dicas de depuração

- Verifique os logs de Systemd para ver o uso de `sudo journalctl -u gunicorn-superlists-staging.ottg.eu`.
- Você pode pedir ao Systemd que verifique a validade da configuração de seu serviço: `systemd-analyze verify /path/to/my.service`.
- Lembre-se de reiniciar os dois serviços sempre que fizer alterações.
- Se você fizer alterações no arquivo de configuração do Systemd, será necessário executar `daemon-reload` antes de `systemctl restart` para ver os efeitos de suas modificações.

Salvando nossas alterações: adicionando o Gunicorn ao nosso requirements.txt

De volta à cópia *local* de nosso repositório, devemos adicionar o Gunicorn à lista de pacotes que precisamos ter em nossos `virtualenvs`:

```
$ pip install gunicorn  
$ pip freeze | grep gunicorn >> requirements.txt  
$ git commit -am "Add gunicorn to virtualenv requirements"  
$ git push
```



No Windows, quando escrevemos este livro, o Gunicorn era instalado sem problemas com `pip install`, mas não executava se você tentasse usá-lo. Felizmente nós só o executamos no servidor, portanto isso não é um problema. O suporte para Windows está sendo discutido (<http://stackoverflow.com/questions/11087682/does-gunicorn-run-on-windows>)...

Pensando em automatizar

Vamos recapitular nossos procedimentos de provisionamento e de implantação:

Provisionamento

1. Suponha que tenhamos uma conta de usuário e uma pasta home
2. `add-apt-repository ppa:fkruill/deadsnakes`
3. `apt-get install nginx git python3.6 python3.6-venv`
4. Adicione a configuração do Nginx para host virtual
5. Adicione o job no Systemd para o Gunicorn

Implantação

1. Crie a estrutura de diretórios em `~/sites`
2. Obtenha o código-fonte e coloque-o em uma pasta chamada *source*
3. Inicie o virtualenv em `../virtualenv`
4. `pip install -r requirements.txt`
5. `manage.py migrate` para o banco de dados
6. `collectstatic` para arquivos estáticos
7. Defina `DEBUG = False` e `ALLOWED_HOSTS` em *settings.py*
8. Reinicie o job do Gunicorn
9. Execute os FTs para verificar se tudo está funcionando

Supondo que não estamos prontos para automatizar completamente

o nosso processo de provisionamento, como podemos salvar os resultados de nossa investigação até agora? Eu diria que os arquivos de configuração do Nginx e de Systemd provavelmente devem ser salvos em algum lugar, de um modo que facilite sua reutilização no futuro. Vamos salvá-los em uma nova subpasta em nosso repositório.

Salvando templates para os arquivos de configuração de nosso provisionamento

Inicialmente vamos criar a subpasta:

```
$ mkdir deploy_tools
```

Eis um template genérico para a nossa configuração do Nginx:

deploy_tools/nginx.template.conf

```
server {
    listen 80;
    server_name SITENAME;

    location /static {
        alias /home/elspeth/sites/SITENAME/static;
    }

    location / {
        proxy_set_header Host $host;
        proxy_pass http://unix:/tmp/SITENAME.socket;
    }
}
```

E eis um template para o serviço Gunicorn no Systemd:

deploy_tools/gunicorn-systemd.template.service

```
[Unit]
Description=Gunicorn server for SITENAME

[Service]
Restart=on-failure
User=elspeth
```



```
WorkingDirectory=/home/elspeth/sites/SITENAME/source
ExecStart=/home/elspeth/sites/SITENAME/virtualenv/bin/gunicorn \
  --bind unix:/tmp/SITENAME.socket \
  superlists.wsgi:application
```

[Install]

```
WantedBy=multi-user.target
```

Agora se tornou fácil para nós usar esses dois arquivos a fim de gerar um novo site, fazendo uma operação de localizar e substituir em SITENAME.

Para o restante, basta manter algumas anotações. Por que não as manter em um arquivo no repositório também?

deploy_tools/provisioning_notes.md

Provisionamento de um novo site

=====

Pacotes necessários:

- * nginx
- * Python 3.6
- * virtualenv + pip
- * Git

Por exemplo, no Ubuntu:

```
sudo add-apt-repository ppa:frull/deadsnakes
sudo apt-get install nginx git python36 python3.6-venv
```

Config do Nginx Virtual Host

- * veja nginx.template.conf
- * substitua SITENAME, por exemplo, por staging.my-domain.com

Serviço Systemd

- * veja gunicorn-systemd.template.service
- * substitua SITENAME, por exemplo, por staging.my-domain.com

Estrutura de pastas:

Suponha que temos uma conta de usuário em /home/username

```
/home/username
├── sites
│   └── SITENAME
│       ├── database
│       ├── source
│       ├── static
│       └── virtualenv
```

Podemos fazer um commit desses arquivos:

```
$ git add deploy_tools
```

```
$ git status # você verá três novos arquivos
```

```
$ git commit -m "Notes and template config files for provisioning"
```

Nossa árvore de códigos-fontes terá agora uma aparência semelhante a esta:

```
.
├── deploy_tools
│   ├── gunicorn-systemd.template.service
│   ├── nginx.template.conf
│   └── provisioning_notes.md
├── functional_tests
│   └── [...]
├── lists
│   ├── __init__.py
│   ├── models.py
│   ├── [...]
│   └── static
│       ├── base.css
│       └── bootstrap
│           └── [...]
├── templates
│   ├── base.html
│   └── [...]
├── tests.py
├── urls.py
├── views.py
├── manage.py
└── requirements.txt
```

└─ superlists
└─ [...]

Salvando o nosso progresso

Ser capaz de executar nossos FTs em um servidor de staging pode ser muito tranquilizador. Na maioria dos casos, porém, você não vai querer executar seus FTs em seu servidor “real”. Para “salvar o seu trabalho” e termos a garantia de que o servidor de produção funcionará tão bem quanto o servidor real, precisamos fazer com que o nosso processo de implantação seja repetível.

A automação é a resposta, e será o assunto do próximo capítulo.

Deixando as implantações de servidores prontas para produção

Eis alguns pontos em que devemos pensar quando tentarmos criar um ambiente de servidor pronto para produção:

Não use o servidor de desenvolvimento do Django em produção

Algo como o Gunicorn ou o uWSGI são uma ferramenta melhor para executar o Django; elas permitirão que você execute vários workers, por exemplo.

Não use o Django para servir seus arquivos estáticos

Não faz sentido usar um processo Python para a simples tarefa de servir arquivos estáticos. O Nginx pode fazer isso, mas o mesmo vale também para outros servidores web como Apache ou uWSGI.

Verifique seu settings.py para conferir as configurações que são exclusivas para desenvolvimento

DEBUG=True e ALLOWED_HOSTS foram as duas configurações que analisamos, mas provavelmente você terá outras (veremos mais quando começarmos a enviar emails a partir do servidor).

Segurança

Uma discussão séria sobre segurança do servidor está além do escopo deste livro, e gostaria de alertar você contra a execução de seus próprios servidores sem conhecer bem mais a respeito desse assunto. (Um dos motivos pelos quais as pessoas preferem usar um PaaS para hospedar seu código é que isso implica um pouco menos de problemas de segurança com que se preocupar.) Se quiser um lugar para começar, eis uma boa opção: My first 5 minutes on a server (Meus primeiros cinco minutos em um servidor, em <https://plusbryan.com/my-first-5-minutes-on-a-server-or-essential-security-for-linux-servers>). Definitivamente, recomendo instalar o fail2ban para uma experiência que vai abrir seus olhos; observe seus arquivos de log para ver a rapidez com que ele detecta tentativas aleatórias de drive-by para uso de força bruta em seu login de SSH. A internet é um lugar perigoso!

CAPÍTULO 11

Automatizando a implantação com o Fabric

Automatize, automatize, automatize.

– CAY HORSTMAN

Automatizar a implantação é fundamental para nossos testes de staging terem algum sentido. Ao garantir que o procedimento de implantação é repetível, damos a nós mesmos a garantia de que tudo correrá bem quando fizermos a implantação em produção.

O Fabric é uma ferramenta que permite automatizar comandos que queremos executar em servidores. O “fabric3” é o fork para Python 3:

```
$ pip install fabric3
```



É seguro ignorar qualquer erro que informe “failed building wheel” (falha na construção de wheel) durante a instalação do fabric3, desde que haja a informação “Successfully installed...” (Instalado com sucesso) no final.

A configuração usual é ter um arquivo chamado *fabfile.py* contendo uma ou mais funções que poderão, mais tarde, ser chamadas a partir de uma ferramenta de linha de comando chamada *fab*, assim:

```
fab function_name:host=SERVER_ADDRESS
```

Esse comando chamará *function_name*, passando uma conexão com o servidor em *SERVER_ADDRESS*. Há várias outras opções para especificar nomes de usuário e senhas, e você poderá encontrar informações sobre elas usando *fab --help*.

Detalhando um script do Fabric para a nossa implantação

A melhor maneira de ver como isso funciona é por meio de um exemplo. Eis um que criei antes (<http://www.bbc.co.uk/cult/classic/bluepeter/valpetejohn/trivia.shtml>), o qual automatiza todos os passos de implantação que descrevemos. A função principal se chama `deploy`; é ela que chamaremos a partir da linha de comando. Então, ela chamará várias funções auxiliares, que implementaremos juntos, uma a uma, explicando à medida que fizermos isso.

`deploy_tools/fabfile.py` (ch09I001)

```
from fabric.contrib.files import append, exists, sed
from fabric.api import env, local, run
import random

REPO_URL = 'https://github.com/hjwp/book-example.git' ❶

def deploy():
    site_folder = f'/home/{env.user}/sites/{env.host}' ❷❸
    source_folder = site_folder + '/source'
    _create_directory_structure_if_necessary(site_folder)
    _get_latest_source(source_folder)
    _update_settings(source_folder, env.host) ❷
    _update_virtualenv(source_folder)
    _update_static_files(source_folder)
    _update_database(source_folder)
```

- ❶ Você vai querer atualizar a variável `REPO_URL` com o URL de seu próprio repositório Git em seu site de compartilhamento de código.
- ❷ `env.host` conterá o endereço do servidor que especificamos na linha de comando (por exemplo, *superlists.ottg.eu*).
- ❸ `env.user` conterá o nome do usuário que você está usando para fazer login no servidor.

Felizmente todas essas funções auxiliares têm nomes razoavelmente descritivos por si só. Como qualquer função em um

fabfile teoricamente pode ser chamada a partir da linha de comando, usei a convenção de um underscore na frente para indicar que elas não foram criadas para fazer parte da “API pública” do fabfile. Vamos ver cada uma delas, em ordem cronológica.

Criando a estrutura de diretórios

Eis o modo como construímos nossa estrutura de diretórios, de forma que ela não se desestruture caso já exista:

deploy_tools/fabfile.py (ch09I002)

```
def _create_directory_structure_if_necessary(site_folder):
    for subfolder in ('database', 'static', 'virtualenv', 'source'):
        run(f'mkdir -p {site_folder}/{subfolder}') ❶❷
```

- ❶ run é o comando mais comum de Fabric. Ele diz: “execute esse comando de shell no servidor”. Os comandos run neste capítulo reproduzirão muitos dos comandos que executamos manualmente nos dois últimos capítulos.
- ❷ mkdir -p é uma variante útil de mkdir, sendo melhor em dois aspectos: pode criar diretórios com vários níveis de profundidade e somente os criará se forem necessários. Assim, mkdir -p /tmp/foo/bar criará o diretório *bar*, mas também criará o seu diretório-pai *foo*, se necessário. O comando também não reclamará se *bar* já existir.¹

Obtendo nosso código-fonte com o Git

A seguir, queremos fazer download da versão mais recente de nosso código-fonte para o servidor, como fizemos com git pull nos capítulos anteriores:

deploy_tools/fabfile.py (ch09I003)

```
def _get_latest_source(source_folder):
    if exists(source_folder + '/.git'): ❶
        run(f'cd {source_folder} && git fetch') ❷❸
    else:
        run(f'git clone {REPO_URL} {source_folder}') ❹
    current_commit = local("git log -n 1 --format=%H", capture=True) ❺
```

```
run('cd {source_folder} && git reset --hard {current_commit}') ⑥
```

- ① `exists` verifica se um diretório ou arquivo já existe no servidor. Procuramos a pasta oculta `.git` para ver se o repositório já foi clonado nessa pasta.
- ② Muitos comandos começam com um `cd` para definir o diretório de trabalho atual. O Fabric não tem nenhum estado, portanto não se lembrará do diretório em que você estiver de um `run` para o próximo.²
- ③ `git fetch` em um repositório existente extrai todos os commits mais recentes da web (é como `git pull`, mas sem atualizar imediatamente a árvore de fontes do ambiente live).
- ④ De modo alternativo, usamos `git clone` com o URL do repositório para trazer uma árvore de fontes atualizada.
- ⑤ O comando `local` do Fabric executa um comando em sua máquina local – na verdade, é apenas um wrapper em torno de `subprocess.Popen`, mas é bem conveniente. Nesse ponto, capturamos a saída da chamada a `git log` para obter o ID do commit atual que está em seu computador local. Isso significa que o servidor ficará com qualquer código cujo checkout tenha sido feito em sua máquina no momento (desde que você o tenha enviado ao servidor).
- ⑥ Fazemos um `reset --hard` para aquele commit, o que eliminará qualquer alteração no diretório de código do servidor.

O resultado final disso é que faremos um `git clone` se for uma nova implantação, ou executaremos um `git fetch + git reset --hard` se uma versão anterior do código já estiver presente; é o equivalente ao `git pull` que usamos quando fizemos o processo manualmente, mas com `reset --hard` para forçar a sobrescrita de qualquer alteração local.



Para que esse script funcione, é preciso ter feito um `git push` de seu commit local atual, de modo que o servidor extraia o código e faça um `reset` para ele. Se você vir um erro informando `Could not parse object` (Não foi possível fazer parse do objeto), experimente executar um `git push`.

Atualizando settings.py

Em seguida, vamos atualizar nosso arquivo de configurações para definir as variáveis `ALLOWED_HOSTS` e `DEBUG` e criar uma nova `SECRET_KEY`:

deploy_tools/fabfile.py (ch09I004)

```
def _update_settings(source_folder, site_name):
    settings_path = source_folder + '/superlists/settings.py'
    sed(settings_path, "DEBUG = True", "DEBUG = False") ❶
    sed(settings_path,
        'ALLOWED_HOSTS = .+$',
        f'ALLOWED_HOSTS = [{"site_name}"]' ❷
    )
    secret_key_file = source_folder + '/superlists/secret_key.py'
    if not exists(secret_key_file): ❸
        chars = 'abcdefghijklmnopqrstuvwxyz0123456789!@#$%^&*(_=+)'
        key = "".join(random.SystemRandom().choice(chars) for _ in range(50))
        append(secret_key_file, f'SECRET_KEY = "{key}"')
        append(settings_path, '\nfrom .secret_key import SECRET_KEY') ❹❺
```

- ❶ O comando `sed` do Fabric faz uma substituição de string em um arquivo; nesse caso, está alterando `DEBUG` de `True` para `False`.
- ❷ Nesse ponto, o comando está acertando `ALLOWED_HOSTS`, usando uma regex para fazer a correspondência com a linha correta.
- ❸ O Django utiliza `SECRET_KEY` para parte de sua criptografia – para itens como cookies e proteção contra CSRF. Garantir que a chave secreta no servidor seja diferente daquela do repositório de seu código-fonte é uma boa prática, pois esse código poderá estar visível a estranhos. Essa seção gerará uma nova chave para ser importada nas configurações, caso ainda não haja nenhuma ali (depois que você tiver uma chave secreta, ela permanecerá a mesma entre as implantações). Encontre mais informações nas documentações do Django (<https://docs.djangoproject.com/en/1.11/topics/signing/>).
- ❹ `append` simplesmente adiciona uma linha no final de um arquivo.

(É inteligente o suficiente para não se preocupar caso a linha já esteja presente, mas não o suficiente para adicionar uma quebra de linha automaticamente caso o arquivo não termine com uma. Daí o “barra n”.)

- 5 Estou usando uma *importação relativa* (from .secret_key em vez de from secret_key) para ter absoluta certeza de que estamos importando o módulo local, em vez de outro módulo de um lugar diferente em sys.path. Discutirei um pouco mais as importações relativas no próximo capítulo.



Fazer hacking do arquivo de configurações dessa forma é uma maneira de alterar a configuração no servidor. Outro padrão comum é usar variáveis de ambiente. Veremos isso no Capítulo 21. Avalie qual das opções você prefere.

Atualizando o virtualenv

A seguir, vamos criar ou atualizar o virtualenv:

deploy_tools/fabfile.py (ch09I005)

```
def _update_virtualenv(source_folder):
    virtualenv_folder = source_folder + '/../virtualenv'
    if not exists(virtualenv_folder + '/bin/pip'): ❶
        run(f'python3.6 -m venv {virtualenv_folder}')
        run(f'{virtualenv_folder}/bin/pip install -r {source_folder}/requirements.txt') ❷
```

- ❶ Observamos a pasta virtualenv em busca do executável do pip como uma maneira de verificar se ela já existe.
- ❷ Então, usamos pip install -r, como fizemos antes.

A atualização de arquivos estáticos é feita com um único comando:

deploy_tools/fabfile.py (ch09I006)

```
def _update_static_files(source_folder):
    run(
        f'cd {source_folder}' ❶
        ' && ../virtualenv/bin/python manage.py collectstatic --noinput' ❷
    )
```

- ❶ Podemos separar strings longas em várias linhas dessa forma no Python; elas são concatenadas como uma única string. É uma causa comum de bugs se o que você realmente queria era uma lista de strings, porém se esqueceu de uma vírgula!
- ❷ Usamos a pasta de binários do virtualenv sempre que precisarmos executar um comando do *manage.py* do Django a fim de garantir que teremos a versão do Django do virtualenv, e não do sistema.

Migrando o banco de dados se for necessário

Por fim, atualizamos o banco de dados com `manage.py migrate`:

deploy_tools/fabfile.py (ch09I007)

```
def _update_database(source_folder):
    run(
        f'cd {source_folder}'
        ' && ../virtualenv/bin/python manage.py migrate --noinput'
    )
```

O `--noinput` remove qualquer confirmação sim/não interativa com a qual o Fabric poderia ter dificuldade de lidar.

Pronto! É muita novidade para absorver, eu suponho, mas espero que você veja como tudo isso está reproduzindo o trabalho que fizemos manualmente antes, com um pouco de lógica para funcionar tanto para implantações totalmente novas quanto para as existentes, que precisem apenas de uma atualização. Se você gosta de palavras com raízes latinas, poderá descrever esse processo como *idempotente*, o que significa que ele tem o mesmo efeito, independentemente de ser executado uma ou várias vezes.

Testando

Vamos testar tudo isso em nosso site de staging e ver o Fabric em ação para atualizar uma implantação já existente:

```
$ cd deploy_tools
$ fab deploy:host=elspeth@superlists-staging.ottg.eu
```

```
[superlists-staging.ottg.eu] Executing task 'deploy'
[superlists-staging.ottg.eu] run: mkdir -p /home/elspeth/sites/superlists-stagin
[superlists-staging.ottg.eu] run: mkdir -p /home/elspeth/sites/superlists-stagin
[superlists-staging.ottg.eu] run: mkdir -p /home/elspeth/sites/superlists-stagin
[superlists-staging.ottg.eu] run: mkdir -p /home/elspeth/sites/superlists-stagin
[superlists-staging.ottg.eu] run: mkdir -p /home/elspeth/sites/superlists-stagin
[superlists-staging.ottg.eu] run: cd /home/elspeth/sites/superlists-staging.ottg
[localhost] local: git log -n 1 --format=%H
[superlists-staging.ottg.eu] run: cd /home/elspeth/sites/superlists-staging.ottg
[superlists-staging.ottg.eu] out: HEAD is now at 85a6c87 Add a fabfile for autom
[superlists-staging.ottg.eu] out:

[superlists-staging.ottg.eu] run: sed -i.bak -r -e 's/DEBUG = True/DEBUG = False
[superlists-staging.ottg.eu] run: echo 'ALLOWED_HOSTS = ["superlists-
staging.ott
[superlists-staging.ottg.eu] run: echo 'SECRET_KEY = "\\4p2u8fi6)bltep(6nd_3tt
[superlists-staging.ottg.eu] run: echo 'from .secret_key import SECRET_KEY' >>
"

[superlists-staging.ottg.eu] run: /home/elspeth/sites/superlists-staging.ottg.eu
[superlists-staging.ottg.eu] out: Requirement already satisfied (use --upgrade t
[superlists-staging.ottg.eu] out: Requirement already satisfied (use --upgrade t
[superlists-staging.ottg.eu] out: Cleaning up...
[superlists-staging.ottg.eu] out:

[superlists-staging.ottg.eu] run: cd /home/elspeth/sites/superlists-staging.ottg
[superlists-staging.ottg.eu] out:
[superlists-staging.ottg.eu] out: 0 static files copied, 11 unmodified.
[superlists-staging.ottg.eu] out:
[superlists-staging.ottg.eu] run: cd /home/elspeth/sites/superlists-staging.ottg
[superlists-staging.ottg.eu] out: Creating tables ...
[superlists-staging.ottg.eu] out: Installing custom SQL ...
[superlists-staging.ottg.eu] out: Installing indexes ...
[superlists-staging.ottg.eu] out: Installed 0 object(s) from 0 fixture(s)
[superlists-staging.ottg.eu] out:
Done.
Disconnecting from superlists-staging.ottg.eu... done.
```

Incrível. Adoro fazer os computadores cuspirem páginas e mais páginas de saída como essas (com efeito, acho difícil me conter e

deixar de fazer pequenos ruídos de computadores dos anos 1970 do tipo *<brrp, brrrp, brrrp>*, como a Mãe no filme *Alien, o oitavo passageiro*). Se observarmos a saída, veremos que ela faz o que determinamos: os comandos `mkdir -p` executam satisfatoriamente, apesar de os diretórios já existirem. Em seguida, `git pull` obtém os dois commits que acabamos de fazer. `sed` e `echo >>` modificam o nosso *settings.py*. Então, `pip install -r requirements.txt` termina sem problemas, percebendo que o `virtualenv` existente já tem todos os pacotes de que precisamos. `collectstatic` também percebe que os arquivos estáticos já estão presentes e, por fim, `migrate` conclui sem problemas.

Configuração do Fabric

Se você estiver usando uma chave SSH para login, estiver armazenando-a no local default e utilizando o mesmo nome de usuário no servidor se comparado com o nome usado localmente, o Fabric deverá “simplesmente funcionar”. Se não estiver, há vários ajustes que talvez você precise aplicar a fim de fazer o comando `fab` executar o que você deseja. Eles giram em torno do nome de usuário, da localização da chave SSH a ser usada ou da senha.

Você pode passá-los para o Fabric na linha de comando. Dê uma olhada com:

```
$ fab --help
```

Ou consulte a documentação do Fabric (<http://docs.fabfile.org>) para mais informações.

Implantação no ambiente live

Vamos então testar tudo isso em nosso site live!

```
$ fab deploy:host=elspeth@superlists.ottg.eu
```

```
$ fab deploy --host=superlists.ottg.eu
[superlists.ottg.eu] Executing task 'deploy'
[superlists.ottg.eu] run: mkdir -p /home/elspeth/sites/superlists.ottg.eu
[superlists.ottg.eu] run: mkdir -p /home/elspeth/sites/superlists.ottg.eu/databa
[superlists.ottg.eu] run: mkdir -p /home/elspeth/sites/superlists.ottg.eu/static
[superlists.ottg.eu] run: mkdir -p /home/elspeth/sites/superlists.ottg.eu/virtua
[superlists.ottg.eu] run: mkdir -p /home/elspeth/sites/superlists.ottg.eu/source
[superlists.ottg.eu] run: git clone https://github.com/hjwp/book-example.git /ho
[superlists.ottg.eu] out: Cloning into '/home/elspeth/sites/superlists.ottg.eu/s
[superlists.ottg.eu] out: remote: Counting objects: 3128, done.
[superlists.ottg.eu] out: Receiving objects: 0% (1/3128)
[...]
[superlists.ottg.eu] out: Receiving objects: 100% (3128/3128), 2.60 MiB | 829 Ki
[superlists.ottg.eu] out: Resolving deltas: 100% (1545/1545), done.
[superlists.ottg.eu] out:

[localhost] local: git log -n 1 --format=%H
[superlists.ottg.eu] run: cd /home/elspeth/sites/superlists.ottg.eu/source && gi
[superlists.ottg.eu] out: HEAD is now at 6c8615b use a secret key file
[superlists.ottg.eu] out:
[superlists.ottg.eu] run: sed -i.bak -r -e 's/DEBUG = True/DEBUG = False/g' "$(e
[superlists.ottg.eu] run: echo 'ALLOWED_HOSTS = ["superlists.ottg.eu"]' >>
"$$(ec
[superlists.ottg.eu] run: echo 'SECRET_KEY = "\\mq(fffwid5vleol%ke^jil*x1mkj-4
[superlists.ottg.eu] run: echo 'from .secret_key import SECRET_KEY' >> "$(echo
/
[superlists.ottg.eu] run: python3.6 -m venv /home/elspeth/sites/superl
[superlists.ottg.eu] out: Using interpreter /usr/bin/python3.6
[superlists.ottg.eu] out: Using base prefix '/usr'
[superlists.ottg.eu] out: New python executable in /home/elspeth/sites/superlist
```

[superlists.ottg.eu] out: Also creating executable in /home/elspeth/sites/superl
[superlists.ottg.eu] out: Installing Setuptools.....done.

[superlists.ottg.eu] out: Installing Pip.....done.
[superlists.ottg.eu] out:

[superlists.ottg.eu] run: /home/elspeth/sites/superlists.ottg.eu/source/./virtu
[superlists.ottg.eu] out: Downloading/unpacking Django==1.11 (from -r /home/el
[superlists.ottg.eu] out: Downloading Django-1.11.tar.gz (8.0MB):

[...]

[superlists.ottg.eu] out: Downloading Django-1.11.tar.gz (8.0MB): 100% 8.0M

[superlists.ottg.eu] out: Running setup.py egg_info for package Django

[superlists.ottg.eu] out:

[superlists.ottg.eu] out: warning: no previously-included files matching '___

[superlists.ottg.eu] out: warning: no previously-included files matching '*'.
[superlists.ottg.eu] out: Downloading/unpacking gunicorn==17.5 (from -r

/home/el

[superlists.ottg.eu] out: Downloading gunicorn-17.5.tar.gz (367kB): 100% 367k

[...]

[superlists.ottg.eu] out: Downloading gunicorn-17.5.tar.gz (367kB): 367kB down

[superlists.ottg.eu] out: Running setup.py egg_info for package gunicorn

[superlists.ottg.eu] out:

[superlists.ottg.eu] out: Installing collected packages: Django, gunicorn

[superlists.ottg.eu] out: Running setup.py install for Django

[superlists.ottg.eu] out: changing mode of build/scripts-3.3/django-admin.py

[superlists.ottg.eu] out:

[superlists.ottg.eu] out: warning: no previously-included files matching '___

[superlists.ottg.eu] out: warning: no previously-included files matching '*'.
[superlists.ottg.eu] out: changing mode of /home/elspeth/sites/superlists.ot

[superlists.ottg.eu] out: Running setup.py install for gunicorn

[superlists.ottg.eu] out:

[superlists.ottg.eu] out:

[superlists.ottg.eu] out: Installing gunicorn_paster script to /home/elspeth

[superlists.ottg.eu] out: Installing gunicorn script to /home/elspeth/sites/

[superlists.ottg.eu] out: Installing gunicorn_django script to /home/elspeth

[superlists.ottg.eu] out: Successfully installed Django gunicorn

[superlists.ottg.eu] out: Cleaning up...

[superlists.ottg.eu] out:

[superlists.ottg.eu] run: cd /home/elspeth/sites/superlists.ottg.eu/source && ..

[superlists.ottg.eu] out: Copying '/home/elspeth/sites/superlists.ottg.eu/source

[superlists.ottg.eu] out: Copying '/home/elspeth/sites/superlists.ottg.eu/source

```
[...]  
[superlists.ottg.eu] out: Copying '/home/elspeth/sites/superlists.ottg.eu/source  
[superlists.ottg.eu] out:  
[superlists.ottg.eu] out: 11 static files copied.  
[superlists.ottg.eu] out:  
[superlists.ottg.eu] run: cd /home/elspeth/sites/superlists.ottg.eu/source && ..  
[superlists.ottg.eu] out: Creating tables ...  
[superlists.ottg.eu] out: Creating table auth_permission  
[...]  
[superlists.ottg.eu] out: Creating table lists_item  
[superlists.ottg.eu] out: Installing custom SQL ...  
[superlists.ottg.eu] out: Installing indexes ...  
[superlists.ottg.eu] out: Installed 0 object(s) from 0 fixture(s)  
[superlists.ottg.eu] out:  
Done.  
Disconnecting from superlists.ottg.eu... done.
```

Brrp brrp brpp. Podemos ver que o script segue um caminho um pouco diferente, executando um git clone para trazer um repositório totalmente novo, em vez de fazer um git pull. Ele também precisa configurar um novo virtualenv do zero, incluindo uma nova instalação de pip e do Django. collectstatic na verdade cria novos arquivos dessa vez, e migrate parece ter funcionado também.

Configuração de Nginx e de Gunicorn usando sed

O que mais precisamos fazer para o nosso site live funcionar em produção? Vamos consultar nossas notas sobre provisionamento, as quais nos dizem para usar os arquivos de template a fim de criar o nosso host virtual de Nginx e o serviço em Systemd. Que tal um pouco da mágica da linha de comando do Unix?

```
elspeth@server:$ sed "s/SITENAME/superlists.ottg.eu/g" \  
    source/deploy_tools/nginx.template.conf \  
    | sudo tee /etc/nginx/sites-available/superlists.ottg.eu
```

O sed (“stream editor”, ou editor de stream) aceita um stream de texto e faz edições nele. Não é por acaso que o comando de substituição de string do Fabric tem o mesmo nome. Nesse caso,

pedimos a ele que substitua a string *SITENAME* pelo endereço de nosso site, usando a sintaxe `s/replaceme/withthis/g`.³ Fizemos um pipe (`|`) da saída para um processo de usuário root (`sudo`), que utiliza `tee` para escrever o resultado do pipe em um arquivo; nesse caso, é o arquivo de configuração do virtualhost do Nginx para os sites disponíveis.

Em seguida, vamos ativar esse arquivo com um link simbólico:

```
elspeth@server:$ sudo ln -s ../sites-available/superlists.ottg.eu \
/etc/nginx/sites-enabled/superlists.ottg.eu
```

Vamos, então, escrever o serviço de Systemd, com outro `sed`:

```
elspeth@server: sed "s/SITENAME/superlists.ottg.eu/g" \
source/deploy_tools/gunicorn-systemd.template.service \
| sudo tee /etc/systemd/system/gunicorn-superlists.ottg.eu.service
```

Por fim, vamos iniciar os dois serviços:

```
elspeth@server:$ sudo systemctl daemon-reload
elspeth@server:$ sudo systemctl reload nginx
elspeth@server:$ sudo systemctl enable gunicorn-superlists.ottg.eu
elspeth@server:$ sudo systemctl start gunicorn-superlists.ottg.eu
```

Vamos dar uma olhada em nosso site: Figura 11.1. Funciona – viva!

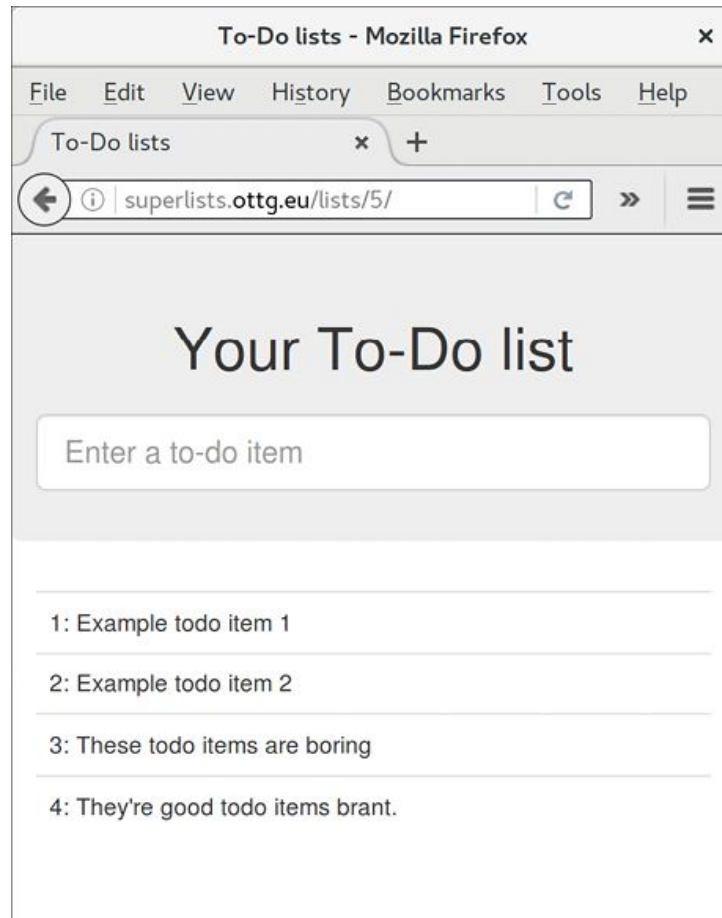


Figura 11.1 – Brrp, brrp, brrp... funcionou!

Foi um bom trabalho. Bom fabfile, você merece um biscoito. Conquistou o privilégio de ser adicionado ao repositório:

```
$ git add deploy_tools/fabfile.py  
$ git commit -m "Add a fabfile for automated deploys"
```

Atribua uma tag do Git à versão

Vamos a uma última dose pequenina de administração. Para preservar um marcador histórico, usaremos tags do Git para marcar o estado da base de código que reflete o que está atualmente ativo no servidor:

```
$ git tag LIVE  
$ export TAG=$(date +DEPLOYED-%F/%H%M) # gera um timestamp  
$ echo $TAG # deve mostrar "DEPLOYED-" e depois o timestamp  
$ git tag $TAG
```

```
$ git push origin LIVE $TAG # faz um push da tag
```

Agora é fácil, a qualquer momento, verificar qual é a diferença entre nossa base de código atual e o que está nos servidores. Esse recurso será útil em alguns capítulos, quando veremos as migrações de banco de dados. Consulte a tag no histórico:

```
$ git log --graph --oneline --decorate  
[...]
```

De qualquer modo, agora temos um site live! Conte a todos os seus amigos! Diga para a sua mãe, se não houver mais ninguém interessado! No próximo capítulo, vamos voltar a programar novamente.

Leituras complementares

Não há algo como uma Única Forma Verdadeira quando se trata de implantação, e, de qualquer maneira, não sou nenhum ancião expert. Tentei colocar você em um caminho razoavelmente sensato, mas há muitas tarefas que poderiam ser feitas de modo diferente, e muito, muito mais a aprender além do que foi apresentado. Eis alguns recursos que utilizei para me inspirar:

- Solid Python Deployments for Everybody (Implantações robustas de Python para todos, <http://hynek.me/talks/pythondeployments>), de Hynek Schlawack;
- Git-based fabric deployments are awesome (Implantações baseadas em Git com Fabric são incríveis, <http://bit.ly/U6tUo5>), de Dan Bravender;
- O capítulo sobre implantação no livro *Two Scoops of Django*, de Dan Greenfeld e Audrey Roy;
- The 12-factor App (A aplicação de 12 fatores, <http://12factor.net/>), da equipe do Heroku.

Para algumas ideias sobre como você poderá lidar com a automação no passo de provisionamento, além de ver uma alternativa ao Fabric, chamada Ansible, consulte o Apêndice C.

Implantações automatizadas

Fabric

O Fabric permite executar comandos em servidores a partir de scripts Python. É uma ótima ferramenta para automatizar tarefas de administração de servidores.

Idempotência

Se o seu script estiver fazendo uma implantação em servidores existentes, será necessário projetá-los de modo que funcionem em uma instalação nova e em um servidor que já esteja configurado.

Mantenha os arquivos de configuração no sistema de controle de versões

Certifique-se de que sua única cópia de um arquivo de configuração não seja a que está no servidor! Esses arquivos são fundamentais para a sua aplicação, e devem estar no sistema de controle de versões, como tudo o mais.

Automatize o provisionamento

Em última análise, *tudo* deve ser automatizado, e isso inclui iniciar servidores totalmente novos e garantir que tenham todos os softwares corretos instalados. Isso envolverá interagir com a API de seu provedor de hospedagem.

Ferramentas para gerenciamento de configuração

O Fabric é bem flexível, porém sua lógica continua baseada em scripting. Ferramentas mais avançadas adotam uma abordagem mais “declarativa” e podem facilitar mais ainda a sua vida. Ansible e Vagrant são duas ferramentas que valem a pena ver (consulte o Apêndice C), mas há muitas outras (Chef, Puppet, Salt, Juju...).

-
- 1 Se você estiver se perguntando por que estamos construindo paths manualmente com f-strings em vez de usar o comando `os.path.join` que vimos antes, é porque `path.join` usará barras invertidas se você executar o script no Windows, mas, definitivamente, queremos barras para a frente no servidor. É uma “sacada” comum!
 - 2 Há um comando `cd` no Fabric, mas achei que seria muita informação para acrescentar neste capítulo.
 - 3 Talvez você já tenha visto nerds usando essa notação estranha `s/mude-isto/para-isto/` na internet. Agora você sabe por quê!

CAPÍTULO 12

Separando testes em vários arquivos e criando método auxiliar genérico para espera

O próximo recurso que podemos implementar é um pouco de validação de dados de entrada. Contudo, à medida que começamos a escrever novos testes, percebemos que está ficando difícil nos localizar em um único *functional_tests.py* e *tests.py*, portanto vamos reorganizá-los de modo a ter vários arquivos – um pouco de refatoração em nossos testes, se você me permite.

Também implementaremos um método auxiliar genérico para espera explícita.

Comece com um FT de validação: evitando itens em branco

À medida que nossos primeiros usuários começam a usar o site, percebemos que, às vezes, eles cometem erros que desorganizam suas listas, como acidentalmente submeter itens de lista em branco ou inserir dois itens idênticos por engano em uma lista. Os computadores foram criados para nos ajudar a parar de cometer erros tolos, portanto vamos ver se podemos fazer com que nosso site nos ajude.

Eis o esboço de um FT:

`functional_tests/tests.py (ch11I001)`

```
def test_cannot_add_empty_list_items(self):
```

```
# Edith acessa a página inicial e acidentalmente tenta submeter  
# um item vazio na lista. Ela tecla Enter na caixa de entrada vazia
```

```
# A página inicial é atualizada e há uma mensagem de erro informando  
# que itens da lista não podem estar em branco
```

```
# Ela tenta novamente com um texto para o item, e isso agora funciona
```

```
# De forma perversa, ela agora decide submeter um segundo item em  
# branco na lista
```

```
# Ela recebe um aviso semelhante na página da lista
```

```
# E ela pode corrigir isso preenchendo o item com um texto  
self.fail('write me!')
```

Tudo está bem até agora, mas, antes de prosseguir – nosso arquivo de testes funcionais está começando a ficar um pouco carregado –, vamos separar os testes em vários arquivos, em que cada arquivo conterá um único método de teste.

Lembre-se de que os testes funcionais estão intimamente ligados às “histórias de usuário”. Se estivéssemos usando algum tipo de ferramenta para gerenciamento de projetos, como um sistema de rastreamento de problemas (issue tracker), poderíamos fazer a mudança de forma que cada arquivo correspondesse a um problema ou um ticket, e seu nome contivesse o ID do ticket. Ou, se preferir pensar em termos de “funcionalidades”, em que cada funcionalidade pode ter várias histórias de usuário, então você poderia ter um arquivo e uma classe para a funcionalidade, e vários métodos para cada uma de suas histórias de usuário.

Também teremos uma classe-base de teste da qual todas as demais poderão herdar. A seguir, descreveremos o modo de fazer isso, passo a passo.

Ignorando um teste

Ao fazer uma refatoração, é sempre bom ter uma suíte de testes

que esteja passando em sua totalidade. Acabamos de escrever um teste com uma falha proposital. Vamos desativá-lo temporariamente usando um decorador chamado “skip”, de unittest:

functional_tests/tests.py (ch11I001-1)

```
from unittest import skip  
[...]
```

```
@skip  
def test_cannot_add_empty_list_items(self):
```

Esse decorador informa ao executor de testes que ele deve ignorar esse teste. Você pode ver que ele funciona: se executarmos os testes novamente, veremos que eles passam:

```
$ python manage.py test functional_tests  
[...]  
Ran 4 tests in 11.577s  
OK
```



Skips são perigosos – você deve se lembrar de removê-los antes de fazer commit de suas alterações no repositório. É por isso que revisões linha a linha de cada um de seus diffs são uma boa ideia!

Não se esqueça do “Refatorar” em “Vermelho, Verde, Refatorar”

Uma crítica que às vezes é generalizada no TDD é que ele resulta em um código com péssima arquitetura, pois o desenvolvedor somente se concentra em fazer os testes passarem em vez de parar e pensar em como o sistema como um todo deveria ser projetado. Acho isso um pouco injusto.

O TDD não é uma solução para todos os problemas. Você ainda precisará investir tempo pensando em um bom design. Contudo, o que acontece com frequência é que as pessoas se esquecem do “Refatorar” no “Vermelho, Verde, Refatorar”. A metodologia permite lançar mão de qualquer código antigo para fazer seus testes passarem, mas ela *também* pede a você que invista tempo refatorando esse código a fim de melhorar o seu design. Caso contrário, será fácil permitir que um “débito técnico”

(<https://martinfowler.com/bliki/TechnicalDebtQuadrant.html>) se acumule.

Geralmente, porém, as melhores ideias sobre como refatorar o código não ocorrem a você de imediato. Elas podem surgir dias, semanas ou até mesmo meses depois que você escreveu uma porção de código, quando estiver trabalhando com algo totalmente não relacionado a essa tarefa e, por acaso, vir um código antigo novamente com um novo olhar. Entretanto, se você estiver no meio de outra tarefa, deve parar para refatorar o código antigo?

A resposta é: depende. No caso apresentado no início do capítulo, não havíamos nem mesmo começado a escrever o nosso novo código. Sabemos que estamos em um estado funcional, portanto podemos justificar o fato de colocar um skip em nosso novo FT (para voltar a um estado em que todos os testes estejam passando) e fazer um pouco de refatoração de imediato.

Mais adiante no capítulo, identificaremos outras porções de código que vamos querer alterar. Nesses casos, em vez de correr o risco de refatorar uma aplicação que não esteja em um estado funcional, faremos uma anotação em uma folha de rascunho sobre o que queremos mudar, e esperaremos até voltarmos a ter uma suíte de testes que esteja passando por completo, antes de fazer a refatoração.

Separando os testes funcionais em vários arquivos

Vamos começar colocando cada teste em sua própria classe, ainda no mesmo arquivo:

functional_tests/tests.py (ch11I002)

```
class FunctionalTest(StaticLiveServerTestCase):
```

```
    def setUp(self):
```

```
        [...]
```

```
    def tearDown(self):
```

```
        [...]
```

```
    def wait_for_row_in_list_table(self, row_text):
```

```
        [...]
```

```
class NewVisitorTest(FunctionalTest):
```

```
def test_can_start_a_list_for_one_user(self):
    [...]
def test_multiple_users_can_start_lists_at_different_urls(self):
    [...]
```

```
class LayoutAndStylingTest(FunctionalTest):
```

```
    def test_layout_and_styling(self):
        [...]
```

```
class ItemValidationTest(FunctionalTest):
```

```
    @skip
    def test_cannot_add_empty_list_items(self):
        [...]
```

Neste ponto, podemos executar os FTs novamente e ver que todos eles continuam passando:

```
Ran 4 tests in 11.577s
```

OK

Fazer isso é um pouco trabalhoso e, provavelmente, poderíamos resolver o problema com menos passos, mas, como repito sempre, exercitar o método passo a passo nos casos fáceis facilita bastante quando tivermos um caso complexo.

Agora vamos passar de um único arquivo de testes para um arquivo para cada classe, além de um arquivo “base” contendo a classe-base da qual todos os testes herdarão. Criaremos quatro cópias de *tests.py*, atribuindo-lhes nomes apropriados, e então apagaremos as partes desnecessárias em cada um:

```
$ git mv functional_tests/tests.py functional_tests/base.py
$ cp functional_tests/base.py functional_tests/test_simple_list_creation.py
$ cp functional_tests/base.py functional_tests/test_layout_and_styling.py
$ cp functional_tests/base.py functional_tests/test_list_item_validation.py
```

base.py pode ser reduzido de modo a conter apenas a classe `FunctionalTest`. Deixaremos o método auxiliar na classe-base, pois suspeitamos que estamos prestes a reutilizá-lo em nosso novo FT:

functional_tests/base.py (ch11I003)

```
from django.contrib.staticfiles.testing import StaticLiveServerTestCase
from selenium import webdriver
from selenium.common.exceptions import WebDriverException
import time
```

```
MAX_WAIT = 10
```

```
class FunctionalTest(StaticLiveServerTestCase):
```

```
    def setUp(self):
        [...]
    def tearDown(self):
        [...]
    def wait_for_row_in_list_table(self, row_text):
        [...]
```



Manter os métodos auxiliares em uma classe-base `FunctionalTest` é uma forma conveniente de evitar duplicação em FTs. Mais adiante no livro (no Capítulo 25), usaremos o “padrão Page”, que está relacionado a isso, mas que dá preferência à composição em vez da herança – o que é sempre bom.

Nosso primeiro FT agora está em seu próprio arquivo e deve ter apenas uma classe e um método de teste:

functional_tests/test_simple_list_creation.py (ch11I004)

```
from .base import FunctionalTest
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
```

```
class NewVisitorTest(FunctionalTest):

    def test_can_start_a_list_for_one_user(self):
        [...]
    def test_multiple_users_can_start_lists_at_different_urls(self):
        [...]
```

Utilizei uma importação relativa (`from .base`). Algumas pessoas gostam de usá-las com frequência em código Django (por exemplo, suas views podem importar modelos usando `from .models import List`, em vez de `from list.models`). Em última análise, é uma questão de preferência pessoal. Prefiro usar importações relativas somente quando tenho certeza absoluta de que a posição relativa do item que estou importando não mudará. Isso se aplica nesse caso, pois sei, com certeza, que todos os testes estarão ao lado de *base.py*, do qual eles herdam.

O FT de layout e estilização agora deve estar em um único arquivo e em uma só classe:

functional_tests/test_layout_and_styling.py (ch11I005)

```
from selenium.webdriver.common.keys import Keys
from .base import FunctionalTest
```

```
class LayoutAndStylingTest(FunctionalTest):
    [...]
```

Por fim, nosso novo teste de validação estará em um arquivo próprio também:

functional_tests/test_list_item_validation.py (ch11I006)

```
from selenium.webdriver.common.keys import Keys
from unittest import skip
from .base import FunctionalTest
```

```
class ItemValidationTest(FunctionalTest):

    @skip
    def test_cannot_add_empty_list_items(self):
        [...]
```

Podemos testar se tudo funcionou executando `manage.py test functional_tests` novamente e conferindo, mais uma vez, se todos os quatro testes são executados:

```
Ran 4 tests in 11.577s
```

OK

Agora podemos remover o nosso decorador `skip`:

`functional_tests/test_list_item_validation.py`
(ch11I007)

```
class ItemValidationTest(FunctionalTest):

    def test_cannot_add_empty_list_items(self):
        [...]
```

Executando um único arquivo de teste

Como bônus, agora podemos executar um arquivo de teste individual, assim:

```
$ python manage.py test functional_tests.test_list_item_validation
[...]  
AssertionError: write me!
```

Brilhante – não há necessidade de ficar sentado esperando que todos os FTs executem se estivermos interessados em um só teste. Contudo, precisamos nos lembrar de executar todos os testes ocasionalmente para verificar se não houve regressões. Mais adiante no livro, veremos como essa tarefa poderá ser feita com um laço automatizado de Integração Contínua (Continuous Integration). Por ora, vamos fazer um commit!

```
$ git status
$ git add functional_tests
```

```
$ git commit -m "Moved Fts into their own individual files"
```

Ótimo. Separamos nossos testes funcionais elegantemente em diferentes arquivos. Em seguida, começaremos a escrever nosso FT, mas em breve, como você já deve ter adivinhado, faremos algo semelhante com os arquivos de nossos testes de unidade.

Uma nova ferramenta para testes funcionais: um método auxiliar genérico para espera explícita

Inicialmente vamos começar a implementar o teste ou, pelo menos, o início dele:

```
functional_tests/test_list_item_validation.py  
(ch11I008)
```

```
def test_cannot_add_empty_list_items(self):  
    # Edith acessa a página inicial e acidentalmente tenta submeter  
    # um item vazio na lista. Ela tecla Enter na caixa de entrada vazia  
    self.browser.get(self.live_server_url)  
    self.browser.find_element_by_id('id_new_item').send_keys(Keys.ENTER)  
  
    # A página inicial é atualizada e há uma mensagem de erro informando  
    # que itens da lista não podem estar em branco  
    self.assertEqual(  
        self.browser.find_element_by_css_selector('.has-error').text, ❶  
        "You can't have an empty list item" ❷  
    )  
  
    # Ela tenta novamente com um texto para o item, e isso agora funciona  
    self.fail("finish this test!")  
    [...]
```

Eis o modo como podemos escrever o teste de forma ingênua:

- ❶ Definimos que vamos usar uma classe CSS chamada `.has-error` para marcar nosso texto com erro. Veremos que o Bootstrap tem algumas estilizações úteis para isso.
- ❷ Podemos verificar se nosso erro exibe a mensagem que queremos.

Todavia, você é capaz de adivinhar qual é o problema em potencial com o teste conforme está escrito agora?

Tudo bem, eu deixei transparecer no título da seção, mas, sempre que fizermos algo que provoque uma atualização de página, precisaremos de uma espera explícita; caso contrário, o Selenium poderá sair procurando o elemento `.has-error` antes de a página ter tido a chance de ser carregada.



Sempre que você submeter um formulário com `Keys.ENTER` ou clicar em algo que fará uma página ser carregada, provavelmente vai querer uma espera explícita para a sua próxima asserção.

Nossa primeira espera explícita foi implementada em um método auxiliar. Para essa espera, podemos decidir que implementar um método auxiliar específico é um exagero nessa etapa, mas poderia ser interessante ter em nossos testes alguma forma genérica de dizer “espere até essa asserção passar”. Algo como:

functional_tests/test_list_item_validation.py (ch11I009)

```
[...]
# A página inicial é atualizada e há uma mensagem de erro informando
# que itens da lista não podem estar em branco
self.wait_for(lambda: self.assertEqual( ❶
    self.browser.find_element_by_css_selector('.has-error').text,
    "You can't have an empty list item"
))
```

- ❶ Em vez de chamar a asserção diretamente, nós a encapsulamos em uma função lambda e a passamos para um novo método auxiliar que supomos que se chame `wait_for`.



Caso não tenha visto funções lambda em Python antes, consulte a seção *Funções lambda* na página XXX.

Como esse método `wait_for` mágico funcionaria? Vamos nos dirigir

para `base.py`, fazer uma cópia de nosso método `wait_for_row_in_list_table` existente e efetuar uma pequena adaptação:

functional_tests/base.py (ch11I010)

```
def wait_for(self, fn): ❶
    start_time = time.time()
    while True:
        try:
            table = self.browser.find_element_by_id('id_list_table') ❷
            rows = table.find_elements_by_tag_name('tr')
            self.assertIn(row_text, [row.text for row in rows])
            return
        except (AssertionError, WebDriverException) as e:
            if time.time() - start_time > MAX_WAIT:
                raise e
            time.sleep(0.5)
```

- ❶ Fazemos uma cópia do método, mas o chamamos de `wait_for` e alteramos o seu argumento. Espera-se que uma função `lhe` seja passada.
- ❷ Por enquanto, ainda estamos com o código antigo, que está verificando linhas da tabela. Como podemos transformar isso em algo que funcione para qualquer `fn` genérico que tenha sido passado?

Fazemos assim:

functional_tests/base.py (ch11I011)

```
def wait_for(self, fn):
    start_time = time.time()
    while True:
        try:
            return fn() ❶
        except (AssertionError, WebDriverException) as e:
            if time.time() - start_time > MAX_WAIT:
                raise e
            time.sleep(0.5)
```

- ❶ O corpo de nosso `try/except`, em vez de ser o código específico para analisar linhas da tabela, simplesmente se transformou em

uma chamada para a função que passamos. Também devolvemos o seu valor de retorno com `return` para podermos sair do laço imediatamente caso nenhuma exceção tenha sido lançada.

Funções lambda

lambda em Python é a sintaxe para criar uma função descartável de uma só linha – ela evita que você precise usar `def..()`: e um bloco indentado:

```
>>> myfn = lambda x: x+1
>>> myfn(2)
3
>>> myfn(5)
6
>>> adder = lambda x, y: x + y
>>> adder(3, 2)
5
```

Em nosso caso, estamos usando-a para transformar uma porção de código que, de outro modo, seria executada imediatamente, em uma função que possamos passar como argumento, e que poderá ser executada mais tarde, inclusive várias vezes.

```

>>> def addthree(x):
...     return x + 3
...
>>> addthree(2)
5
>>> myfn = lambda: addthree(2) # observe que addthree não é chamada
                                # imediatamente aqui

>>> myfn
<function <lambda> at 0x7f3b140339d8>
>>> myfn()
5
>>> myfn()
5

```

Vamos ver o nosso método auxiliar `wait_for` simples em ação:

```

$ python manage.py test functional_tests.test_list_item_validation
[...]
=====
ERROR: test_cannot_add_empty_list_items
(functional_tests.test_list_item_validation.ItemValidationTest)
-----
Traceback (most recent call last):
  File ".../superlists/functional_tests/test_list_item_validation.py", line
15, in test_cannot_add_empty_list_items
    self.wait_for(lambda: self.assertEqual( ❶
  File ".../superlists/functional_tests/base.py", line 37, in wait_for
    raise e ❷
  File ".../superlists/functional_tests/base.py", line 34, in wait_for
    return fn() ❸
  File ".../superlists/functional_tests/test_list_item_validation.py", line
16, in <lambda> ❹
    self.browser.find_element_by_css_selector('.has-error').text, ❺
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate
element: .has-error
-----
Ran 1 test in 10.575s

```

FAILED (errors=1)

A ordem do traceback é um pouco confusa, mas podemos, de certo modo, compreender o que aconteceu:

- ❶ Na linha 15 de nosso FT, chegamos ao nosso método auxiliar `self.wait_for`, passando-lhe a versão lambda de `assertEqual`.
- ❷ Chegamos até `self.wait_for` em `base.py`, e podemos ver aí que chamamos a lambda o número de vezes suficiente para termos executado `raise e`, pois o nosso timeout expirou.
- ❸ Para explicar de que lugar a exceção realmente veio, o traceback nos leva de volta para `test_list_item_validation.py` e para dentro do corpo da função lambda, e nos informa que estava tentando encontrar o elemento `.has-error` que falhou.

Estamos agora nos domínios da programação funcional, passando funções como argumentos para outras funções, e pode ser um pouco difícil de entender. Sei que precisei de um pouco de tempo para me acostumar com isso! Leia esse código algumas vezes, e o código lá no FT, para deixar que eles sejam absorvidos; se você continuar confuso, não se preocupe demais com esse código e permita que sua autoconfiança aumente à medida que trabalhar com ele. Vamos usá-lo mais algumas vezes neste livro e deixá-lo mais divertido ainda funcionalmente – você verá.

Finalizando o FT

Finalizaremos o FT assim:

`functional_tests/test_list_item_validation.py`
(ch11I012)

```
# A página inicial é atualizada e há uma mensagem de erro informando
# que itens da lista não podem estar em branco
self.wait_for(lambda: self.assertEqual(
    self.browser.find_element_by_css_selector('.has-error').text,
    "You can't have an empty list item"
))
```

```

# Ela tenta novamente com um texto para o item, e isso agora funciona
self.browser.find_element_by_id('id_new_item').send_keys('Buy milk')
self.browser.find_element_by_id('id_new_item').send_keys(Keys.ENTER)
self.wait_for_row_in_list_table('1: Buy milk')

# De forma perversa, ela agora decide submeter um segundo item em
# branco na lista
self.browser.find_element_by_id('id_new_item').send_keys(Keys.ENTER)

# Ela recebe um aviso semelhante na página da lista
self.wait_for(lambda: self.assertEqual(
    self.browser.find_element_by_css_selector('.has-error').text,
    "You can't have an empty list item"
))

# E ela pode corrigir isso preenchendo o item com um texto
self.browser.find_element_by_id('id_new_item').send_keys('Make tea')
self.browser.find_element_by_id('id_new_item').send_keys(Keys.ENTER)
self.wait_for_row_in_list_table('1: Buy milk')
self.wait_for_row_in_list_table('2: Make tea')

```

Métodos auxiliares em FTs

Temos dois métodos auxiliares agora: o nosso método auxiliar genérico `self.wait_for` e `wait_for_row_in_list_table`. O primeiro é um utilitário genérico – qualquer um de nossos FTs pode precisar fazer uma espera.

O segundo também ajuda a evitar duplicações no código de seus testes funcionais. No dia em que decidirmos alterar a implementação do funcionamento de nossa tabela com lista, queremos garantir que teremos de mudar o código de nosso FT em um só lugar, e não em dezenas deles em vários FTs...

Consulte também o Capítulo 25 e o Apêndice E para ver mais informações sobre como estruturar o código de seu FT.

Deixarei que você faça o commit de sua “primeira versão de FT”.

Refatorando os testes de unidade em vários

arquivos

Quando (finalmente!) começarmos a implementar a nossa solução, vamos adicionar outro teste para o nosso *models.py*. Antes de fazer isso, é hora de organizar nossos testes de unidade de modo semelhante aos testes funcionais.

Uma diferença será o fato de que, como a aplicação *lists* contém o verdadeiro código da aplicação bem como os testes, vamos separar os testes em sua própria pasta:

```
$ mkdir lists/tests
$ touch lists/tests/__init__.py
$ git mv lists/tests.py lists/tests/test_all.py
$ git status
$ git add lists/tests
$ python manage.py test lists
[...]
Ran 9 tests in 0.034s
```

OK

```
$ git commit -m "Move unit tests into a folder with single file"
```

Se vir uma mensagem informando “Ran 0 tests” (Nenhum teste executado), é provável que você tenha se esquecido de adicionar o *dunderinit* – ele deve estar presente; do contrário, a pasta de testes não será um pacote Python válido...¹

Vamos agora transformar *test_all.py* em dois arquivos, um chamado *test_views.py*, que conterá somente os testes para *view*, e outro chamado *test_models.py*. Começarei fazendo duas cópias:

```
$ git mv lists/tests/test_all.py lists/tests/test_views.py
$ cp lists/tests/test_views.py lists/tests/test_models.py
```

Reduza *test_models.py* de modo que contenha apenas um teste – isso significa que ele precisará de muito menos importações:

lists/tests/test_models.py (ch11i016)

```
from django.test import TestCase
from lists.models import Item, List
```

```
class ListAndItemModelsTest(TestCase):  
    [...]
```

Por outro lado, *test_views.py* perde apenas uma classe:

lists/tests/test_views.py (ch11I017)

```
--- a/lists/tests/test_views.py  
+++ b/lists/tests/test_views.py  
@@ -103,34 +104,3 @@ class ListViewTest(TestCase):  
     self.assertNotContains(response, 'other list item 1')  
     self.assertNotContains(response, 'other list item 2')  
  
-  
-  
-class ListAndItemModelsTest(TestCase):  
-  
- def test_saving_and_retrieving_items(self):  
    [...]
```

Vamos executar os testes novamente para verificar se tudo continua lá:

```
$ python manage.py test lists  
[...]  
Ran 9 tests in 0.040s
```

OK

Ótimo! Esse foi outro pequeno passo funcional:

```
$ git add lists/tests  
$ git commit -m "Split out unit tests into two files"
```



Algumas pessoas gostam de deixar seus testes de unidade em uma pasta de testes de imediato, assim que iniciam um projeto. É uma ideia perfeitamente boa; só achei que deveria esperar até que isso se tornasse necessário a fim de evitar fazer muita arrumação de casa, tudo no primeiro capítulo!

Bem, aqui estão nossos FTs e os testes de unidade elegantemente reorganizados. No próximo capítulo, vamos fazer um pouco de

validações apropriadas.

Dicas para organizar testes e refatorar

Utilize uma pasta para testes

Assim como você utiliza vários arquivos para armazenar o código de sua aplicação, deve separar seus testes em vários arquivos.

- Para os testes funcionais, agrupe-os em testes para uma funcionalidade ou para uma história de usuário específica.
- Para os testes de unidade, utilize uma pasta chamada *tests*, com um `__init__.py`.
- Provavelmente você vai querer um arquivo de teste separado para cada arquivo de código-fonte testado. Para o Django, geralmente são: *test_models.py*, *test_views.py* e *test_forms.py*.
- Tenha pelo menos um teste que sirva de placeholder para *cada* função e classe.

Não se esqueça do “Refatorar” em “Vermelho, Verde, Refatorar”

O ponto mais importante em ter testes é permitir que você refatore o seu código! Utilize-os e deixe seu código (incluindo os testes) o mais limpo possível.

Não refatore com testes em falha

- Em geral!
- Contudo, o FT em que você estiver trabalhando no momento não conta.
- Ocasionalmente você poderá colocar um skip em um teste que esteja verificando algo que você ainda não implementou.
- O mais comum é realizar uma anotação sobre a refatoração que você deseja fazer, finalizar aquilo em que você está trabalhando e fazer a refatoração um pouco mais tarde, quando estiver de volta a um estado funcional.
- Não se esqueça de remover qualquer skip antes de fazer commit de seu código! Sempre revise seus diffs linha a linha para identificar modificações como essa.

Experimente implementar um método auxiliar genérico wait_for

Ter métodos auxiliares específicos que façam esperas explícitas é ótimo e ajuda a deixar seus testes legíveis. Entretanto, com frequência, você precisará também de uma asserção *ad-hoc* de uma linha ou uma interação com o Selenium aos quais você vai querer adicionar uma espera. `self.wait_for` faz um bom trabalho para mim, mas você poderá descobrir um padrão um pouco diferente, que funcione para você.

¹ “Dunder” é a forma abreviada de underscore duplo, portanto “dunderinit” quer dizer `__init__.py`.

CAPÍTULO 13

Validação na camada do banco de dados

Nos próximos capítulos, discutiremos os testes e a implementação da validação de dados de entrada do usuário.

No que concerne ao conteúdo, o capítulo terá bastante material acima de tudo relacionado às especificidades do Django e menos discussão sobre a filosofia do TDD. Isso não significa que você não aprenderá nada sobre testes – há muitas informações aqui e ali sobre eles neste capítulo, porém talvez sejam mais sobre realmente como “pegar o jeito”, sobre o ritmo do TDD e como o trabalho é feito.

Depois que terminarmos esses três pequenos capítulos, reservarei um pouco de diversão com JavaScript (!) para o final da Parte II. Então, prosseguiremos para a Parte III, em que prometo que retornaremos diretamente às verdadeiras discussões relevantes sobre a metodologia do TDD – testes de unidade *versus* testes integrados, simulação (mocking) e outros assuntos. Continue sintonizado!

Por enquanto, porém, veremos um pouco de validação. Vamos simplesmente nos lembrar da direção em que nosso FT está apontando:

```
$ python3 manage.py test functional_tests.test_list_item_validation
[...]
=====
=====
ERROR: test_cannot_add_empty_list_items
(functional_tests.test_list_item_validation.ItemValidationTest)
```

Traceback (most recent call last):

File ".../superlists/functional_tests/test_list_item_validation.py", line 15, in test_cannot_add_empty_list_items

self.wait_for(lambda: self.assertEqual(

[...]

File ".../superlists/functional_tests/test_list_item_validation.py", line 16, in <lambda>

self.browser.find_element_by_css_selector('.has-error').text,

[...]

selenium.common.exceptions.NoSuchElementException: Message: Unable to locate

element: .has-error

Espera-se que uma mensagem de erro seja vista se o usuário tentar inserir um item vazio.

Validação na camada do modelo

Em uma aplicação web, há dois lugares em que podemos fazer uma validação: no lado do cliente (usando JavaScript ou propriedades do HTML5, como veremos mais adiante) e no lado do servidor. O lado do servidor é “mais seguro”, pois uma pessoa sempre pode passar pelo lado do cliente, seja maliciosamente, seja por causa de algum bug.

De modo semelhante, no lado do servidor, no Django, há dois níveis em que podemos fazer uma validação. Um deles é no nível do modelo, e o outro é acima, no nível dos formulários. Gosto de usar o nível mais baixo sempre que possível, parcialmente porque tenho uma queda pelos bancos de dados e pelas suas regras de integridade, e parcialmente porque, repetindo, é mais seguro – às vezes você pode se esquecer de qual formulário usou para validar a entrada, mas sempre utilizará o mesmo banco de dados.

Gerenciador de contexto `self.assertRaises`

Vamos começar a trabalhar e escrever um teste de unidade na camada dos modelos. Acrescente um novo método de teste em

ListAndItemModelsTest, que tentará criar um item em branco na lista. Esse teste é interessante porque verifica se o código deve lançar uma exceção:

lists/tests/test_models.py (ch11I018)

```
from django.core.exceptions import ValidationError
[...]
```

```
class ListAndItemModelsTest(TestCase):
    [...]
```

```
def test_cannot_save_empty_list_items(self):
    list_ = List.objects.create()
    item = Item(list=list_, text="")
    with self.assertRaises(ValidationError):
        item.save()
```



Se Python é novidade para você, talvez você não tenha visto antes a instrução `with`. Ela é usada com o que chamamos de “gerenciadores de contexto” (context managers), que encapsulam um bloco de código, geralmente com algum tipo de código de configuração, limpeza ou tratamento de erros. Há um bom texto nas notas de lançamento da versão 2.5 do Python (<http://docs.python.org/release/2.5/whatsnew/pep-343.html>).

Essa é uma nova técnica de teste de unidade: quando queremos verificar se uma ação lançará um erro, podemos utilizar o gerenciador de contexto `self.assertRaises`. Como alternativa, poderíamos ter usado algo como:

```
try:
    item.save()
    self.fail('The save should have raised an exception')
except ValidationError:
    pass
```

No entanto, a construção com `with` é mais limpa. Agora podemos tentar executar o teste e ver a falha esperada:

```
item.save()
AssertionError: ValidationError not raised
```

Idiossincrasia do Django: salvar o modelo não faz a validação ser executada

Descobriremos agora uma das pequenas idiossincrasias do Django. *Esse teste já deveria passar.* Se você consultar a documentação dos campos de modelos do Django (<http://bit.ly/SuxPJO>), verá que o default de `TextField`, na verdade, é `blank=False`, o que significa que valores vazios *deveriam* ser proibidos.

Então, por que o teste continua falhando? Bem, por motivos históricos um pouco contraintuitivos (<http://bit.ly/2v3SfRq>), os modelos do Django não executam uma validação completa ao salvar os dados. Conforme veremos depois, qualquer restrição que seja implementada no banco de dados lançará erros quando os dados forem salvos, mas o SQLite não oferece suporte para impor restrições de dados vazios em colunas de texto; desse modo, nosso método para salvar dados está deixando esse valor inválido passar silenciosamente.

Há uma maneira de verificar se a restrição ocorrerá ou não no nível do banco de dados: se fosse no nível do banco de dados, precisaríamos de uma migração para aplicar a restrição. Contudo, o Django sabe que o SQLite não aceita esse tipo de restrição; portanto, se tentarmos executar `makemigrations`, ele informará que não há nada a ser feito:

```
$ python manage.py makemigrations
No changes detected
```

No entanto, o Django tem um método para executar uma validação completa manualmente, chamado `full_clean` (mais informações podem ser obtidas na documentação em <http://bit.ly/2u5SlxA>). Vamos explorá-lo para ver como funciona:

lists/tests/test_models.py

```
with self.assertRaises(ValidationError):
    item.save()
    item.full_clean()
```

Com isso, conseguimos fazer o teste passar:

OK

Muito bom. Isso nos ensinou um pouco sobre a validação do Django, e o teste está aí para nos alertar se algum dia esquecermos o nosso requisito e definirmos `blank=True` no campo `text` (faça o teste!)

Mostrando erros de validação do modelo na view

Vamos tentar impor nossa validação de modelo na camada de views e levá-la até nossos templates para que o usuário possa vê-la. Eis o modo como podemos, opcionalmente, exibir um erro em nosso HTML – verificamos se o template recebeu uma variável de erro e, em caso afirmativo, nós a exibimos ao lado do formulário:

lists/templates/base.html (ch11l020)

```
<form method="POST" action="{% block form_action %}{% endblock %}">
  <input name="item_text" id="id_new_item"
    class="form-control input-lg"
    placeholder="Enter a to-do item" />
  {% csrf_token %}
  {% if error %}
    <div class="form-group has-error">
      <span class="help-block">{{ error }}</span>
    </div>
  {% endif %}
</form>
```

Dê uma olhada na documentação do Bootstrap (<http://getbootstrap.com/css/#forms>) para ver mais informações sobre controles de formulário.

Passar esse erro para o template é tarefa da função de view. Vamos observar os testes de unidade na classe `NewListTest`. Usarei aqui dois padrões levemente distintos de tratamento de erros.

No primeiro caso, nosso URL e a view para novas listas renderizarão opcionalmente o mesmo template da página inicial, porém com o acréscimo de uma mensagem de erro. Eis um teste de

unidade para isso:

lists/tests/test_views.py (ch11I021)

```
class NewListTest(TestCase):
    [...]

    def test_validation_errors_are_sent_back_to_home_page_template(self):
        response = self.client.post('/lists/new', data={'item_text': ''})
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'home.html')
        expected_error = "You can't have an empty list item"
        self.assertContains(response, expected_error)
```

Enquanto escrevemos esse teste, podemos nos sentir um pouco ofendidos com o URL `/lists/new`, que estamos inserindo manualmente como uma string. Temos muitos URLs fixos no código de nossos testes, em nossas views e nos templates, o que viola o princípio DRY. Não me importo com um pouco de duplicação nos testes, mas, definitivamente, devemos prestar atenção nos URLs fixos no código em nossas views e templates, e fazer uma anotação para lembrar de refatorá-los. Contudo, não faremos isso de imediato, pois, no momento, nossa aplicação se encontra em um estado de falha. Queremos retornar a um estado funcional antes.

Voltemos ao nosso teste, que falha porque a view no momento está devolvendo um redirecionamento 302 em vez de devolver uma resposta 200 “normal”:

```
AssertionError: 302 != 200
```

Vamos tentar chamar `full_clean()` na view:

lists/views.py

```
def new_list(request):
    list_ = List.objects.create()
    item = Item.objects.create(text=request.POST['item_text'], list=list_)
    item.full_clean()
    return redirect(f'/lists/{list_.id}/')
```

Enquanto observamos o código da view, identificamos um URL fixo

no código, que é um bom candidato para ser eliminado. Vamos adicionar essa informação em nossa folha de rascunho:

- **Remover os URLs fixos no código de views.py**

Agora a validação do modelo lança uma exceção, que chega até a nossa view:

```
[...]
File "../superlists/lists/views.py", line 11, in new_list
    item.full_clean()
[...]
django.core.exceptions.ValidationError: {'text': ['This field cannot be blank.']}
```

Desse modo, tentaremos nossa primeira abordagem: usar um try/except para detectar erros. Obedecendo ao Testing Goat, começaremos somente com try/except e nada mais. Os testes devem nos dizer o que devemos implementar em seguida...

lists/views.py (ch11I025)

```
from django.core.exceptions import ValidationError
[...]

def new_list(request):
    list_ = List.objects.create()
    item = Item.objects.create(text=request.POST['item_text'], list=list_)
    try:
        item.full_clean()
    except ValidationError:
        pass
    return redirect(f'/lists/{list_.id}/')
```

Isso nos leva de volta a 302 != 200:

```
AssertionError: 302 != 200
```

Vamos, então, devolver um template renderizado, que deve cuidar da verificação do template também:

lists/views.py (ch11I026)

```
except ValidationError:
    return render(request, 'home.html')
```

Os testes agora nos dizem para colocar a mensagem de erro no template:

```
AssertionError: False is not true : Couldn't find 'You can't have an empty list item' in response
```

Fazemos isso passando uma nova variável de template em:

lists/views.py (ch11I027)

```
except ValidationError:
    error = "You can't have an empty list item"
    return render(request, 'home.html', {"error": error})
```

Humm, parece que isso não funcionou exatamente:

```
AssertionError: False is not true : Couldn't find 'You can't have an empty list item' in response
```

Uma pequena depuração com print...

lists/tests/test_views.py

```
expected_error = "You can't have an empty list item"
print(response.content.decode())
self.assertContains(response, expected_error)
```

...nos mostrará a causa – o Django escapou o apóstrofo do HTML (<https://docs.djangoproject.com/en/1.11/ref/templates/builtins/#autoescape>):

```
[...]
<span class="help-block">You can&#39;t have an empty list
item</span>
```

Poderíamos fazer um hack como este em nosso teste:

```
expected_error = "You can&#39;t have an empty list item"
```

No entanto, usar uma função auxiliar do Django provavelmente é uma ideia melhor:

lists/tests/test_views.py (ch11I029)

```
from django.utils.html import escape
[...]
```

```
expected_error = escape("You can't have an empty list item")
```

```
self.assertContains(response, expected_error)
```

O teste passou!

```
Ran 11 tests in 0.047s
```

OK

Verificando se uma entrada inválida não é salva no banco de dados

Antes de continuar, porém, você notou um pequeno erro de lógica que permitimos que se esgueirasse em nossa implementação? Atualmente estamos criando um objeto, apesar de a validação falhar:

lists/views.py

```
item = Item.objects.create(text=request.POST['item_text'], list=list_)
try:
    item.full_clean()
except ValidationError:
    [...]
```

Vamos adicionar um novo teste de unidade para garantir que itens vazios de lista não sejam salvos:

lists/tests/test_views.py (ch11I030-1)

```
class NewListTest(TestCase):
    [...]

    def test_validation_errors_are_sent_back_to_home_page_template(self):
        [...]

    def test_invalid_list_items_arent_saved(self):
        self.client.post('/lists/new', data={'item_text': ''})
        self.assertEqual(List.objects.count(), 0)
        self.assertEqual(Item.objects.count(), 0)
```

Eis o resultado:

```
[...]
Traceback (most recent call last):
```

```
File "/.../superlists/lists/tests/test_views.py", line 40, in
test_invalid_list_items_arent_saved
    self.assertEqual(List.objects.count(), 0)
AssertionError: 1 != 0
```

Vamos fazer a correção assim:

lists/views.py (ch11I030-2)

```
def new_list(request):
    list_ = List.objects.create()
    item = Item(text=request.POST['item_text'], list=list_)
    try:
        item.full_clean()
        item.save()
    except ValidationError:
        list_.delete()
        error = "You can't have an empty list item"
        return render(request, 'home.html', {"error": error})
    return redirect(f'/lists/{list_.id}/')
```

Os FTs passam?

```
$ python manage.py test functional_tests.test_list_item_validation
[...]
File "/.../superlists/functional_tests/test_list_item_validation.py", line
29, in test_cannot_add_empty_list_items
    self.wait_for(lambda: self.assertEqual(
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate
element: .has-error
```

Não exatamente, mas eles avançaram um pouco. Ao verificar a *linha 29*, podemos ver que conseguimos passar pela primeira parte do teste e agora estamos na segunda verificação – a submissão de um segundo item vazio também faz um erro ser apresentado.

Entretanto, temos algum código funcionando, portanto vamos fazer um commit:

```
$ git commit -am "Adjust new list view to do model validation"
```


Padrão Django: processando requisições POST na mesma view em que o formulário é renderizado

Dessa vez, usaremos uma abordagem um pouco diferente: uma que, na verdade, é um padrão bem comum no Django, e consiste em utilizar a mesma view para processar requisições POST e renderizar o formulário do qual elas vieram. Embora isso não se enquadre muito bem no modelo de URL RESTful, tem a importante vantagem de que o mesmo URL pode exibir um formulário e mostrar qualquer erro encontrado no processamento da entrada do usuário.

De acordo com a situação atual, temos uma view e um URL para exibir uma lista, e uma view e um URL para processar acréscimos nessa lista. Vamos combiná-los em um só. Desse modo, em *list.html*, nosso formulário terá um alvo diferente:

lists/templates/list.html (ch11l030)

```
{% block form_action %}/lists/{{ list.id }}/{% endblock %}
```

A propósito, esse é outro URL fixo no código. Vamos adicioná-lo em nossa lista de tarefas e, enquanto estamos pensando nele, há um também em *home.html*:

- **Remover os URLs fixos no código de views.py**
- **Remover URLs fixos no código de formulários em list.html e home.html**

Isso fará com que nosso teste funcional original falhe de imediato, pois a página `view_list` ainda não sabe como processar requisições POST:

```
$ python manage.py test functional_tests
```

```
[...]
```

```
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
```

```
element: .has-error
```

```
[...]
```

AssertionError: '2: Use peacock feathers to make a fly' not found in ['1: Buy peacock feathers']



Nesta seção estamos fazendo uma refatoração no nível da aplicação. Executamos nossa refatoração no nível da aplicação alterando ou adicionando testes de unidade, e então adaptando nosso código. Utilizamos os testes funcionais para nos dizer quando nossa refatoração está completa e tudo voltou a funcionar como antes. Observe novamente o diagrama que está no final do Capítulo 4 caso precise se orientar.

Refatorar: transferindo a funcionalidade `new_item` para `view_list`

Vamos tomar todos os testes antigos de `NewItemTest`, aqueles relacionados a salvar requisições POST em listas existentes, e passá-las para `ListViewTest`. Enquanto trabalhamos nisso, também os fazemos apontar para o URL da lista-base, em vez de apontar para `.../add_item`:

`lists/tests/test_views.py` (ch11I031)

```
class ListViewTest(TestCase):

    def test_uses_list_template(self):
        [...]

    def test_passes_correct_list_to_template(self):
        [...]

    def test_displays_only_items_for_that_list(self):
        [...]

    def test_can_save_a_POST_request_to_an_existing_list(self):
        other_list = List.objects.create()
        correct_list = List.objects.create()

        self.client.post(
            f'/lists/{correct_list.id}/',
            data={'item_text': 'A new item for an existing list'})
```

```

)

self.assertEqual(Item.objects.count(), 1)
new_item = Item.objects.first()
self.assertEqual(new_item.text, 'A new item for an existing list')
self.assertEqual(new_item.list, correct_list)

def test_POST_redirects_to_list_view(self):
    other_list = List.objects.create()
    correct_list = List.objects.create()

    response = self.client.post(
        f'/lists/{correct_list.id}/',
        data={'item_text': 'A new item for an existing list'}
    )
    self.assertRedirects(response, f'/lists/{correct_list.id}/')

```

Observe que a classe `NewItemTest` desaparece totalmente. Também alterei o nome do teste de redirecionamento para deixar explícito que ele se aplica somente às requisições POST.

Eis o resultado:

```

FAIL: test_POST_redirects_to_list_view (lists.tests.test_views.ListViewTest)
AssertionError: 200 != 302 : Response didn't redirect as expected: Response
code was 200 (expected 302)
[...]
FAIL: test_can_save_a_POST_request_to_an_existing_list
(lists.tests.test_views.ListViewTest)
AssertionError: 0 != 1

```

Alteramos a função `view_list` para que trate dois tipos de requisição:

lists/views.py (ch11I032-1)

```

def view_list(request, list_id):
    list_ = List.objects.get(id=list_id)
    if request.method == 'POST':
        Item.objects.create(text=request.POST['item_text'], list=list_)
        return redirect(f'/lists/{list_.id}/')
    return render(request, 'list.html', {'list': list_})

```

Isso resulta em testes que passam:

```
Ran 12 tests in 0.047s
```

OK

Agora, podemos apagar a view `add_item`, pois ela não é mais necessária... opa, uma falha inesperada:

```
[...]
```

```
AttributeError: module 'lists.views' has no attribute 'add_item'
```

É porque apagamos a view, mas ela continua sendo referenciada em `urls.py`. Vamos removê-la dali:

`lists/urls.py` (ch111033)

```
urlpatterns = [  
    url(r'^new$', views.new_list, name='new_list'),  
    url(r'^(id+)/$', views.view_list, name='view_list'),  
]
```

Com isso, obtivemos o OK. Vamos testar uma execução completa do FT:

```
$ python manage.py test
```

```
[...]
```

```
ERROR: test_cannot_add_empty_list_items
```

```
[...]
```

```
Ran 16 tests in 15.276s
```

```
FAILED (errors=1)
```

Estamos de volta à única falha em nosso novo teste funcional. A refatoração da funcionalidade `add_item` está completa. Devemos fazer um commit neste ponto:

```
$ git commit -am "Refactor list view to handle new item POSTs"
```



Então eu quebrei a regra de jamais refatorar quando há testes em falha? Nesse caso, é permitido porque a refatoração é necessária para a nossa nova funcionalidade. Sem dúvida, você jamais deve refatorar com testes de *unidade* em falha. Em meu livro, porém, não há problemas se o FT para a história de usuário na qual você está trabalhando agora esteja com falha¹.

¹ Se realmente quiser uma execução de teste “limpa”, você pode adicionar um

skip ou fazer um return precoce do FT atual, mas é necessário garantir que você não vai se esquecer disso acidentalmente.

Impondo a validação do modelo em `view_list`

Continuamos querendo que a adição de itens em listas existentes esteja sujeita às nossas regras de validação do modelo. Vamos escrever um novo teste de unidade para isso; é muito semelhante àquele para a página inicial, apenas com duas adaptações:

`lists/tests/test_views.py` (ch11I034)

```
class ListViewTest(TestCase):
    [...]

    def test_validation_errors_end_up_on_lists_page(self):
        list_ = List.objects.create()
        response = self.client.post(
            f'/lists/{list_.id}',
            data={'item_text': ''}
        )
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'list.html')
        expected_error = escape("You can't have an empty list item")
        self.assertContains(response, expected_error)
```

Esse teste deve falhar, pois nossa view atualmente não faz nenhuma validação e apenas faz um redirecionamento para todos os POSTs:

```
self.assertEqual(response.status_code, 200)
AssertionError: 302 != 200
```

Eis uma implementação:

`lists/views.py` (ch11I035)

```
def view_list(request, list_id):
    list_ = List.objects.get(id=list_id)
    error = None

    if request.method == 'POST':
        try:
```

```
    item = Item(text=request.POST['item_text'], list=list_)
    item.full_clean()
    item.save()
    return redirect(f'/lists/{list_.id}/')
except ValidationError:
    error = "You can't have an empty list item"
```

```
return render(request, 'list.html', {'list': list_, 'error': error})
```

Não é profundamente satisfatório, não é mesmo? Sem dúvida, há um pouco de duplicação de código nesse caso; aquele `try/except` ocorre duas vezes em `views.py` e, em geral, o código parece desajeitado.

Ran 13 tests in 0.047s

OK

No entanto, vamos esperar um pouco antes de fazer mais refatorações, pois sabemos que estamos prestes a fazer um código de validação um pouco diferente para itens duplicados. Por enquanto, vamos apenas adicionar isso em nossa folha de rascunho:

- **Remover os URLs fixos no código de `views.py`**
- **Remover URLs fixos no código de formulários em `list.html` e `home.html`**
- **Remover duplicação na lógica de validação em `views`**



Um dos motivos para a existência da regra “três acertos e refatorar” é que, se você esperar até ter três casos de uso, cada um deles poderá ser um pouco diferente, e você terá uma visão melhor sobre o que é a funcionalidade comum. Se fizer a refatoração cedo demais, poderá descobrir que o terceiro caso de uso não se esquadra exatamente em seu código refatorado...

Pelo menos nossos testes funcionais voltam a passar:

```
$ python manage.py test functional_tests
[...]
OK
```

Retornamos a um estado funcional, portanto podemos dar uma olhada em alguns dos itens de nossa folha de rascunho. É uma boa hora para um commit. E possivelmente um intervalo para um café.

```
$ git commit -am "enforce model validation in list view"
```

Refatorar: removendo os URLs fixos no código

Você se lembra daqueles parâmetros `name=` em `urls.py`? Simplesmente os copiamos do exemplo default que o Django nos deu, e eu lhes dei alguns nomes razoavelmente descritivos. Vamos agora descobrir para que eles servem:

lists/urls.py

```
url(r'^new$', views.new_list, name='new_list'),
url(r'^(d+)/$', views.view_list, name='view_list'),
```

Tag de template {% url %}

Podemos substituir o URL fixo em `home.html` por uma tag de template Django que se refira ao “nome” do URL:

lists/templates/home.html (ch11I036-1)

```
{% block form_action %}{% url 'new_list' %}{% endblock %}
```

Verificamos se isso não causa falha nos testes de unidade:

```
$ python manage.py test lists
OK
```

Vamos fazer isso no outro template. Esse é mais interessante porque lhe passamos um parâmetro:

lists/templates/list.html (ch11I036-2)

```
{% block form_action %}{% url 'view_list' list.id %}{% endblock %}
```

Consulte a documentação do Django sobre resolução inversa de URL (<http://bit.ly/2uKaMzA>) para obter mais informações.

Executamos os testes novamente e conferimos se todos eles passam:

```
$ python manage.py test lists
OK
$ python manage.py test functional_tests
OK
```

Excelente:

```
$ git commit -am "Refactor hard-coded URLs out of templates"
```

- **Remover os URLs fixos no código de views.py**
- ~~Remover URLs fixos no código de formulários em list.html e home.html~~
- **Remover duplicação na lógica de validação em views**

Usando `get_absolute_url` para redirecionamentos

Vamos agora encarar *views.py*. Uma maneira de fazer isso é como no template, passando o nome do URL e um argumento posicional:

lists/views.py (ch11I036-3)

```
def new_list(request):
    [...]
    return redirect('view_list', list_.id)
```

Isso faria os testes de unidade e os testes funcionais passarem, mas a função `redirect` é capaz de uma mágica melhor do que essa! No Django, como objetos de modelo geralmente estão associados a um URL em particular, podemos definir uma função especial chamada `get_absolute_url` que informa qual página exibirá o item. Ela é conveniente nesse caso, mas também é útil na administração do Django (não será discutida no livro, mas, em breve, você descobrirá por conta própria): permitirá que você olhe um objeto na view de administração e passe a olhá-lo no site live. Sempre recomendo definir um `get_absolute_url` para um modelo se houver um que faça

sentido; isso não exige nenhum tempo adicional.

Tudo que é necessário é um teste de unidade supersimples em `test_models.py`:

lists/tests/test_models.py (ch11I036-4)

```
def test_get_absolute_url(self):
    list_ = List.objects.create()
    self.assertEqual(list_.get_absolute_url(), f'/lists/{list_.id}/')
```

O que resulta em:

AttributeError: 'List' object has no attribute 'get_absolute_url'

A implementação consiste em usar a função `reverse` do Django, que essencialmente faz o inverso do que o Django normalmente faz em `urls.py` (consulte a documentação em <https://docs.djangoproject.com/en/1.11/topics/http/urls/#reverse-resolution-of-urls>):

lists/models.py (ch11I036-5)

```
from django.core.urlresolvers import reverse
```

```
class List(models.Model):
```

```
    def get_absolute_url(self):
        return reverse('view_list', args=[self.id])
```

Agora podemos usá-la na view – a função `redirect` apenas toma o objeto para o qual queremos redirecionar e utiliza `get_absolute_url` internamente, de forma automática, como num passe de mágica!

lists/views.py (ch11I036-6)

```
def new_list(request):
    [...]
    return redirect(list_)
```

Há mais informações na documentação do Django (<https://docs.djangoproject.com/en/1.11/topics/http/shortcuts/#redirect>). Faça uma verificação rápida para ver se os testes de unidade

continuam passando:

OK

Então fazemos o mesmo com `view_list`:

lists/views.py (ch11I036-7)

```
def view_list(request, list_id):  
    [...]  
  
    item.save()  
    return redirect(list_)  
except ValidationError:  
    error = "You can't have an empty list item"
```

Executamos, então, um teste de unidade e um teste funcional completos para termos a garantia de que tudo continua funcionando:

```
$ python manage.py test lists  
OK  
$ python manage.py test functional_tests  
OK
```

Riscamos o item de nossa lista de tarefas...

- ~~Remover os URLs fixos no código de views.py~~
- ~~Remover URLs fixos no código de formulários em list.html e home.html~~
- **Remover duplicação na lógica de validação em views**

E fazemos um commit...

```
$ git commit -am "Use get_absolute_url on List model to DRY urls in views"
```

Terminamos essa parte! Temos uma validação funcionando na camada do modelo e aproveitamos a oportunidade para fazer algumas refatorações no decorrer do processo.

O último item da folha de rascunho será o assunto do próximo capítulo...

Sobre a validação na camada do banco de dados

Sempre gosto de empurrar minha lógica de validação para o nível mais baixo possível.

Validação na camada do banco de dados é a garantia definitiva de integridade dos dados

Ela pode garantir que, independentemente da complexidade que o código nas camadas superiores atingir, no nível mais baixo, seus dados serão válidos e consistentes.

Contudo o preço é a flexibilidade

Essa vantagem não é gratuita! Agora é impossível, mesmo que temporariamente, ter dados inconsistentes. Às vezes, você poderá ter um bom motivo para, temporariamente, armazenar dados que quebrem as regras, em vez de não armazenar nada. Talvez você esteja importando dados de uma origem externa em várias etapas, por exemplo.

Não foi projetada para ser amigável ao usuário

Tentar armazenar dados inválidos fará um `IntegrityError` desagradável ser devolvido pelo seu banco de dados e, possivelmente, o usuário verá uma página de erro 500 confusa. Como veremos em capítulos mais adiante, a validação no nível de formulários é projetada com o usuário em mente, prevendo os tipos de mensagens de erro úteis que queremos lhe enviar.

CAPÍTULO 14

Um formulário simples

No final do último capítulo, ficamos com a ideia de que havia muita duplicação de código nas partes que cuidavam da validação em nossas views. O Django nos incentiva a usar classes de formulário para fazer o trabalho de validar dados de entrada de usuários e escolher as mensagens de erro a serem exibidas. Vamos ver como isso funciona.

À medida que avançarmos no capítulo, também investiremos um pouco de tempo organizando nossos testes de unidade e garantindo que cada um deles teste apenas um aspecto de cada vez.

Passando a lógica de validação para um formulário



No Django, uma view complexa é um code smell. Parte dessa lógica poderia ser passada para um formulário? Ou para alguns métodos personalizados na classe de modelo? Ou quem sabe até mesmo para um módulo que não seja do Django e represente a sua lógica de negócios?

Os formulários têm vários superpoderes no Django:

- Podem processar entradas de usuário e validá-las para verificar erros.
- Podem ser usados em templates para renderizar elementos de entrada HTML e mensagens de erro também.
- Além disso, como veremos mais adiante, alguns deles podem até mesmo salvar dados no banco de dados para você.

Não é necessário usar os três superpoderes em cada um dos formulários. Talvez você prefira implementar o seu próprio HTML, ou salvar os dados por conta própria. Contudo, eles são um local excelente para manter uma lógica de validação.

Explorando a API dos formulários com um teste de unidade

Vamos fazer um pequeno experimento com formulários usando um teste de unidade. Meu plano é iterar em direção a uma solução completa; espero apresentar os formulários gradualmente, de modo que façam sentido caso você não os tenha visto antes.

Inicialmente vamos adicionar um novo arquivo para nossos testes de unidade de formulário, e começaremos com um teste que apenas olha o HTML do formulário:

lists/tests/test_forms.py

```
from django.test import TestCase

from lists.forms import ItemForm

class ItemFormTest(TestCase):

    def test_form_renders_item_text_input(self):
        form = ItemForm()
        self.fail(form.as_p())
```

`form.as_p()` renderiza o formulário como HTML. Esse teste de unidade utiliza um `self.fail` para um código exploratório. Você poderia usar uma sessão `manage.py shell` de forma igualmente fácil, mas seria necessário ficar recarregando o seu código a cada alteração.

Vamos criar um formulário mínimo. Ele herda da classe-base `Form` e tem um único campo chamado `item_text`:

lists/forms.py

```
from django import forms
```

```
class ItemForm(forms.Form):
    item_text = forms.CharField()
```

Vemos agora uma mensagem de falha que nos informa como será a aparência do HTML do formulário, gerado automaticamente:

```
self.fail(form.as_p())
AssertionError: <p><label for="id_item_text">Item text:</label> <input
type="text" name="item_text" required id="id_item_text" /></p>
```

Já é muito parecido com o que há em *base.html*. Não temos o atributo de placeholder nem as classes CSS do Bootstrap. Vamos transformar o nosso teste de unidade em um teste para isso:

lists/tests/test_forms.py

```
class ItemFormTest(TestCase):

    def test_form_item_input_has_placeholder_and_css_classes(self):
        form = ItemForm()
        self.assertIn('placeholder="Enter a to-do item"', form.as_p())
        self.assertIn('class="form-control input-lg"', form.as_p())
```

Isso nos dá uma falha que justifica um pouco de programação de verdade. Como podemos personalizar a entrada para um campo de formulário? Usando um “widget”. Aqui está ele, somente com o placeholder:

lists/forms.py

```
class ItemForm(forms.Form):
    item_text = forms.CharField(
        widget=forms.fields.TextInput(attrs={
            'placeholder': 'Enter a to-do item',
        }),
    )
```

Eis o resultado:

```
AssertionError: 'class="form-control input-lg"' not found in '<p><label
for="id_item_text">Item text:</label> <input type="text" name="item_text"
placeholder="Enter a to-do item" required id="id_item_text" /></p>'
```

E, então, temos:

lists/forms.py

```
widget=forms.fields.TextInput(attrs={
    'placeholder': 'Enter a to-do item',
    'class': 'form-control input-lg',
}),
```



Fazer esse tipo de personalização de widget seria tedioso se tivéssemos um formulário muito maior e mais complexo. Dê uma olhada em [django-crispy-forms](https://django-crispy-forms.readthedocs.org/) (<https://django-crispy-forms.readthedocs.org/>) e em [django-floppyforms](http://bit.ly/1rR5eyD) (<http://bit.ly/1rR5eyD>) para obter ajuda.

Testes orientados a desenvolvimento: usando testes de unidade para uma programação exploratória

Isso soa um pouco como testes orientados a desenvolvimento? De vez em quando, tudo bem.

Quando estiver explorando uma nova API, certamente você tem permissão para trabalhar despreocupadamente com ela por um tempo antes de retornar a um TDD rigoroso. Você pode usar o console interativo ou escrever um pouco de código exploratório (mas deve prometer ao Testing Goat que jogará esse código fora e o reescreverá de forma apropriada depois).

Nesse caso, estamos usando um teste de unidade como forma de fazer experimentos com a API de formulários. Na verdade, é uma maneira muito boa para saber como ela funciona.

Passando para um ModelForm do Django

O que vem a seguir? Queremos que nosso formulário reutilize o código de validação que já definimos em nosso modelo. O Django disponibiliza uma classe especial que pode gerar um formulário automaticamente para um modelo; ela se chama `ModelForm`. Como veremos, ela é configurada por meio de um atributo especial chamado `Meta`:

lists/forms.py

```
from django import forms

from lists.models import Item

class ItemForm(forms.models.ModelForm):

    class Meta:
        model = Item
        fields = ('text',)
```

Em Meta especificamos para qual modelo é o formulário e quais campos queremos que sejam utilizados.

Os ModelForms fazem todo tipo de tarefas inteligentes, como atribuir tipos de entrada sensatos em formulários HTML para diferentes tipos de campos e aplicar uma validação default. Consulte a documentação

(<https://docs.djangoproject.com/en/1.11/topics/forms/modelforms/>)

para obter mais informações.

Agora temos um HTML de formulário com um aspecto um pouco diferente:

```
AssertionError: 'placeholder="Enter a to-do item"' not found in '<p><label
for="id_text">Text:</label> <textarea name="text" cols="40" rows="10" required
id="id_text">\n</textarea></p>'
```

Nosso placeholder e a classe CSS foram perdidos. Contudo, podemos ver também que o formulário está usando name="text" em vez de name="item_text". Provavelmente poderemos conviver com isso. No entanto, uma textarea está sendo usada no lugar de uma entrada usual, é não é essa UI que queremos em nossa aplicação. Felizmente podemos sobrescrever widgets para campos de ModelForm de modo semelhante a como fizemos com o formulário usual:

lists/forms.py

```
class ItemForm(forms.models.ModelForm):
```

```

class Meta:
    model = Item
    fields = ('text',)
    widgets = {
        'text': forms.fields.TextInput(attrs={
            'placeholder': 'Enter a to-do item',
            'class': 'form-control input-lg',
        }),
    }

```

Com isso o teste passa.

Testando e personalizando a validação de formulário

Vamos ver agora se `ModelForm` assumiu as mesmas regras de validação que definimos no modelo. Também veremos como passar dados para o formulário, como se tivessem vindo do usuário:

lists/tests/test_forms.py (ch11I008)

```

def test_form_validation_for_blank_items(self):
    form = ItemForm(data={'text': ''})
    form.save()

```

Eis o resultado:

`ValueError: The Item could not be created because the data didn't validate.`

Muito bom: o formulário não permitirá que você salve os dados caso um texto vazio para o item lhe seja fornecido.

Vamos ver agora se podemos fazer com que ele utilize a mensagem de erro específica que queremos. A API para verificar a validação de formulário *antes* de tentarmos salvar qualquer dado é uma função chamada `is_valid`:

lists/tests/test_forms.py (ch11I009)

```

def test_form_validation_for_blank_items(self):
    form = ItemForm(data={'text': ''})
    self.assertFalse(form.is_valid())
    self.assertEqual(

```

```
        form.errors['text'],
        ["You can't have an empty list item"]
    )
```

Chamar `form.is_valid()` devolve `True` ou `False`, mas também tem o efeito colateral de validar os dados de entrada e preencher o atributo `errors`. É um dicionário que mapeia os nomes dos campos para listas de erros para esses campos (é possível que um campo tenha mais de um erro).

Eis o resultado:

```
AssertionError: ['This field is required.'] != ["You can't have an empty list item"]
```

O Django já tem uma mensagem de erro default que poderíamos apresentar ao usuário – você poderia usá-la se estivesse com pressa para construir a sua aplicação web, mas nós nos preocupamos o suficiente para deixar nossa mensagem especial. Personalizá-la significa alterar `error_messages`, que é outra variável de `Meta`:

lists/forms.py (ch11I010)

```
class Meta:
    model = Item
    fields = ('text',)
    widgets = {
        'text': forms.fields.TextInput(attrs={
            'placeholder': 'Enter a to-do item',
            'class': 'form-control input-lg',
        }),
    }
    error_messages = {
        'text': {'required': "You can't have an empty list item"}
    }
```

OK

Você sabe o que seria melhor do que ficar manipulando todas essas strings de erro? Ter uma constante:

lists/forms.py (ch11I011)

```
EMPTY_ITEM_ERROR = "You can't have an empty list item"
```

[...]

```
error_messages = {  
    'text': {'required': EMPTY_ITEM_ERROR}  
}
```

Execute os testes novamente para ver se eles passam... Tudo certo. Vamos agora alterar o teste:

lists/tests/test_forms.py (ch11I012)

```
from lists.forms import EMPTY_ITEM_ERROR, ItemForm  
[...]
```

```
def test_form_validation_for_blank_items(self):  
    form = ItemForm(data={'text': ''})  
    self.assertFalse(form.is_valid())  
    self.assertEqual(form.errors['text'], [EMPTY_ITEM_ERROR])
```

Os testes continuam passando:

OK

Ótimo. Podemos fazer commit de tudo:

```
$ git status # deve mostrar lists/forms.py e tests/test_forms.py  
$ git add lists  
$ git commit -m "new form for list items"
```

Usando o formulário em nossas views

Pensei originalmente em estender esse formulário para incluir uma validação de unicidade, assim como uma validação de item vazio. No entanto, há uma espécie de corolário na metodologia enxuta de “implantar o mais cedo possível”, que é “combinar o código (fazer merge) o mais cedo possível”. Em outras palavras: enquanto implementamos essas pequenas porções do código de formulários, seria fácil continuar por anos adicionando mais e mais funcionalidades ao formulário – eu devo saber, pois é exatamente isso que fiz durante a versão preliminar deste capítulo, e acabei fazendo todo tipo de trabalho, criando uma classe de formulário capaz de cantar e dançar de todas as formas até perceber que ele

não funcionaria realmente para o nosso caso de uso básico.

Assim, em vez de fazer dessa forma, tente usar sua nova porção de código o mais cedo possível. Isso garante que você jamais terá porções de código não utilizadas por aí, e que você começará a testar o seu código “no mundo real” o mais cedo possível.

Temos uma classe de formulário capaz de renderizar um pouco de HTML e fazer a validação de pelo menos um tipo de erro – vamos começar a usá-la! Poderemos usá-la em nosso template *base.html* e, portanto, em todas as nossas views.

Usando o formulário em uma view com uma requisição GET

Vamos começar em nossos testes de unidade para a view inicial. Adicionaremos um novo método que verificará se estamos usando o tipo correto de formulário:

lists/tests/test_views.py (ch11I013)

```
from lists.forms import ItemForm

class HomePageTest(TestCase):

    def test_uses_home_template(self):
        [...]

    def test_home_page_uses_item_form(self):
        response = self.client.get('/')
        self.assertIsInstance(response.context['form'], ItemForm) ❶
```

❶ `assertIsInstance` verifica se nosso formulário é da classe correta.

Eis o resultado:

```
KeyError: 'form'
```

Desse modo, usamos o formulário na view de nossa página inicial:

lists/views.py (ch11I014)

```
[...]
```



```
from lists.forms import ItemForm
from lists.models import Item, List
```

```
def home_page(request):
    return render(request, 'home.html', {'form': ItemForm()})
```

Tudo bem, vamos agora tentar usá-lo no template – substituímos o antigo `<input ..>` por `{{ form.text }}`:

lists/templates/base.html (ch11l015)

```
<form method="POST" action="{% block form_action %}{% endblock %}">
  {{ form.text }}
  {% csrf_token %}
  {% if error %}
    <div class="form-group has-error">
```

`{{ form.text }}` renderiza somente a entrada HTML para o campo text do formulário.

Uma grande operação de localizar e substituir

Uma atitude que tomamos, porém, foi alterar o nosso formulário – ele não utiliza mais os mesmos atributos id e name. Se executarmos nossos testes funcionais, você verá que eles falham na primeira vez que tentam encontrar a caixa de entrada:

```
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate
element: [id="id_new_item"]
```

Precisaremos corrigir isso, e essa tarefa envolverá uma grande operação de localizar e substituir. Antes disso, vamos fazer um commit para manter as mudanças de nomes separadas da mudança de lógica:

```
$ git diff # revise as mudanças em base.html, views.py e seus testes
$ git commit -am "use new form in home_page, simplify tests. NB breaks
stuff"
```

Vamos corrigir os testes funcionais. Um grep rápido nos mostra que há vários locais em que estamos usando id_new_item:

```
$ grep id_new_item functional_tests/test*
```

```
functional_tests/test_layout_and_styling.py: inputbox =
self.browser.find_element_by_id('id_new_item')
functional_tests/test_layout_and_styling.py: inputbox =
self.browser.find_element_by_id('id_new_item')
functional_tests/test_list_item_validation.py:
self.browser.find_element_by_id('id_new_item').send_keys(Keys.ENTER)
[...]
```

É um bom indicativo para uma refatoração. Vamos criar um novo método auxiliar em *base.py*:

functional_tests/base.py (ch11I018)

```
class FunctionalTest(StaticLiveServerTestCase):
    [...]
    def get_item_input_box(self):
        return self.browser.find_element_by_id('id_text')
```

Então vamos usá-lo em todos os lugares – tive que fazer quatro alterações em *test_simple_list_creation.py*, duas em *test_layout_and_styling.py* e quatro em *test_list_item_validation.py*, por exemplo:

functional_tests/test_simple_list_creation.py

```
# Ela é convidada a inserir um item de tarefa imediatamente
inputbox = self.get_item_input_box()
```

Ou:

functional_tests/test_list_item_validation.py

```
# um item vazio na lista. Ela tecla Enter na caixa de entrada vazia
self.browser.get(self.live_server_url)
self.get_item_input_box().send_keys(Keys.ENTER)
```

Não mostrarei todas as mudanças individualmente; tenho certeza de que você pode cuidar disso sozinho! Você pode refazer o `grep` a fim de verificar se identificou todas as ocorrências.

Concluimos o primeiro passo, mas agora temos que atualizar o restante do código da aplicação para que esteja de acordo com a alteração. Precisamos localizar todas as ocorrências dos antigos `id_new_item` e `name(item_text)` e substituí-las também por `id_text` e `text`,

respectivamente:

```
$ grep -r id_new_item lists/  
lists/static/base.css:#id_new_item {
```

Essa foi uma mudança; de modo semelhante, vamos fazê-la para name:

```
$ grep -lr item_text lists  
[...]  
lists/views.py: item = Item(text=request.POST['item_text'], list=list_)  
lists/views.py: item = Item(text=request.POST['item_text'], list=list_)  
lists/tests/test_views.py: self.client.post('/lists/new',  
data={'item_text': 'A new list item'})  
lists/tests/test_views.py: response = self.client.post('/lists/new',  
data={'item_text': 'A new list item'})  
[...]  
lists/tests/test_views.py: data={'item_text': ''}  
[...]
```

Feito isso, executamos novamente os testes de unidade a fim de verificar se tudo continua funcionando:

```
$ python manage.py test lists  
[...]  
-----  
Ran 17 tests in 0.126s
```

OK

E os testes funcionais também:

```
$ python manage.py test functional_tests  
[...]  
File ".../superlists/functional_tests/test_simple_list_creation.py", line  
37, in test_can_start_a_list_for_one_user  
    return self.browser.find_element_by_id('id_text')  
File ".../superlists/functional_tests/base.py", line 51, in  
get_item_input_box  
    return self.browser.find_element_by_id('id_text')  
[...]  
selenium.common.exceptions.NoSuchElementException: Message: Unable to  
locate  
element: [id="id_text"]
```

```
[...]
FAILED (errors=3)
```

Não exatamente! Vamos ver em que ponto as falhas estão ocorrendo – se verificar o número da linha de uma das falhas, você verá que sempre que submetemos um primeiro item, a caixa de entrada desapareceu da página de listas.

Ao verificar *views.py* e a view *new_list*, podemos ver que o problema ocorre porque, quando detectamos um erro de validação, não estamos passando o formulário para o template *home.html*:

lists/views.py

```
except ValidationError:
    list_.delete()
    error = "You can't have an empty list item"
    return render(request, 'home.html', {"error": error})
```

Queremos usar o formulário nessa view também. Antes de fazer qualquer outra alteração, porém, vamos fazer um commit:

```
$ git status
$ git commit -am "rename all item input ids and names. still broken"
```

Usando o formulário em uma view que aceita requisições POST

Agora queremos adaptar os testes de unidade para a view *new_list*, especialmente o teste que lida com a validação. Vamos analisá-lo agora:

lists/tests/test_views.py

```
class NewListTest(TestCase):
    [...]

    def test_validation_errors_are_sent_back_to_home_page_template(self):
        response = self.client.post('/lists/new', data={'text': ''})
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'home.html')
        expected_error = escape("You can't have an empty list item")
```

```
self.assertContains(response, expected_error)
```

Adaptando os testes de unidade para a view `new_list`

Para começar, esse teste está verificando muitos aspectos de uma só vez, portanto temos aqui uma oportunidade para deixar a situação mais clara. Devemos separar duas asserções diferentes:

- Se houver um erro de validação, devemos renderizar o template da página inicial com um 200.
- Se houver um erro de validação, a resposta deve conter o nosso texto de erro.

Além disso, podemos acrescentar uma nova asserção:

- Se houver um erro de validação, devemos passar o nosso objeto de formulário para o template.

Aproveitando a ocasião, usaremos nossa constante em vez de utilizar a string fixa no código para essa mensagem de erro:

`lists/tests/test_views.py` (ch11I023)

```
from lists.forms import ItemForm, EMPTY_ITEM_ERROR  
[...]
```

```
class NewListTest(TestCase):  
    [...]
```

```
    def test_for_invalid_input_renders_home_template(self):  
        response = self.client.post('/lists/new', data={'text': ''})  
        self.assertEqual(response.status_code, 200)  
        self.assertTemplateUsed(response, 'home.html')
```

```
    def test_validation_errors_are_shown_on_home_page(self):  
        response = self.client.post('/lists/new', data={'text': ''})  
        self.assertContains(response, escape(EMPTY_ITEM_ERROR))
```

```
def test_for_invalid_input_passes_form_to_template(self):
    response = self.client.post('/lists/new', data={'text': ''})
    self.assertIsInstance(response.context['form'], ItemForm)
```

Muito melhor. Cada teste agora está verificando claramente um único aspecto e, com um pouco de sorte, apenas um falhará e nos dirá o que devemos fazer:

```
$ python manage.py test lists
```

```
[...]
```

```
=====
=====
```

```
ERROR: test_for_invalid_input_passes_form_to_template
(lists.tests.test_views.NewListTest)
```

```
-----
```

```
Traceback (most recent call last):
```

```
File ".../superlists/lists/tests/test_views.py", line 49, in
test_for_invalid_input_passes_form_to_template
    self.assertIsInstance(response.context['form'], ItemForm)
```

```
[...]
```

```
KeyError: 'form'
```

```
-----
```

```
Ran 19 tests in 0.041s
```

```
FAILED (errors=1)
```

Usando o formulário na view

Eis o modo como usamos o formulário na view:

lists/views.py

```
def new_list(request):
    form = ItemForm(data=request.POST) ❶
    if form.is_valid(): ❷
        list_ = List.objects.create()
        Item.objects.create(text=request.POST['text'], list=list_)
        return redirect(list_)
    else:
        return render(request, 'home.html', {"form": form}) ❸
```

❶ Passamos os dados de request.POST para o construtor do

formulário.

- ② Usamos `form.is_valid()` para determinar se essa é uma submissão boa ou ruim.
- ③ Se for inválida, passamos o formulário para o template, em vez de passar nossa string de erro fixa no código.

Essa view está com um aspecto bem melhor agora! Além disso, todos os nossos testes passam, exceto um:

```
self.assertContains(response, escape(EMPTY_ITEM_ERROR))
[...]
AssertionError: False is not true : Couldn't find 'You can't have an empty
list item' in response
```

Usando o formulário para exibir erros no template

Estamos com a falha porque ainda não estamos usando o formulário para exibir erros no template:

lists/templates/base.html (ch11l026)

```
<form method="POST" action="{% block form_action %}{% endblock %}">
  {{ form.text }}
  {% csrf_token %}
  {% if form.errors %} ①
    <div class="form-group has-error">
      <div class="help-block">{{ form.text.errors }}</div> ②
    </div>
  {% endif %}
</form>
```

- ① `form.errors` contém uma lista de todos os erros do formulário.
- ② `form.text.errors` é uma lista contendo os erros somente do campo `text`.

O que isso faz com nossos testes?

```
FAIL: test_validation_errors_end_up_on_lists_page
(lists.tests.test_views.ListViewTest)
[...]
```

```
AssertionError: False is not true : Couldn't find 'You can't have an empty list item' in response
```

Uma falha inesperada – na verdade, está nos testes de nossa view final, `view_list`. Como alteramos o modo de os erros serem exibidos em *todos* os templates, não estamos mais mostrando o erro que passamos manualmente para o template.

Isso significa que precisaremos retrabalhar `view_list` também, antes de retornarmos a um estado funcional.

Usando o formulário na outra view

Essa view trata tanto requisições GET quanto requisições POST. Vamos começar verificando se o formulário é usado em requisições GET. Podemos criar um novo teste para isso:

lists/tests/test_views.py

```
class ListViewTest(TestCase):
    [...]

    def test_displays_item_form(self):
        list_ = List.objects.create()
        response = self.client.get(f'/lists/{list_.id}/')
        self.assertIsInstance(response.context['form'], ItemForm)
        self.assertContains(response, 'name="text"')
```

Eis o resultado:

```
KeyError: 'form'
```

A seguir, apresentamos uma implementação mínima:

lists/views.py (ch11|028)

```
def view_list(request, list_id):
    [...]
    form = ItemForm()
    return render(request, 'list.html', {
        'list': list_, "form": form, "error": error
    })
```


Um método auxiliar para vários testes pequenos

A seguir, queremos usar os erros de formulário na segunda view. Separaremos nosso teste atual único para o caso inválido (`test_validation_errors_end_up_on_lists_page`) em vários testes distintos:

`lists/tests/test_views.py` (ch11I030)

```
class ListViewTest(TestCase):
    [...]

    def post_invalid_input(self):
        list_ = List.objects.create()
        return self.client.post(
            f'/lists/{list_.id}/',
            data={'text': ''}
        )

    def test_for_invalid_input_nothing_saved_to_db(self):
        self.post_invalid_input()
        self.assertEqual(Item.objects.count(), 0)

    def test_for_invalid_input_renders_list_template(self):
        response = self.post_invalid_input()
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'list.html')

    def test_for_invalid_input_passes_form_to_template(self):
        response = self.post_invalid_input()
        self.assertIsInstance(response.context['form'], ItemForm)

    def test_for_invalid_input_shows_error_on_page(self):
        response = self.post_invalid_input()
        self.assertContains(response, escape(EMPTY_ITEM_ERROR))
```

Ao criar uma pequena função auxiliar `post_invalid_input`, podemos criar quatro testes separados sem duplicar muitas linhas de código.

A essa altura, já vimos isso várias vezes. Com frequência, parece mais natural escrever testes de view como um único bloco monolítico de asserções – a view deve fazer isso, aquilo e mais

aquilo e então devolver aquilo com isso. Contudo, fazer a separação em vários testes definitivamente vale a pena; como vimos em capítulos anteriores, isso ajuda a isolar o exato problema que você possa ter quando, mais tarde, vier a alterar o seu código e introduzir acidentalmente um bug. Os métodos auxiliares são uma das ferramentas que reduzem a barreira psicológica.

Por exemplo, agora podemos ver que há apenas uma falha, e é uma falha clara:

```
FAIL: test_for_invalid_input_shows_error_on_page
(lists.tests.test_views.ListViewTest)
AssertionError: False is not true : Couldn't find 'You can't have an empty
list item' in response
```

Vamos ver se podemos reescrever adequadamente a view agora de modo que ela utilize o nosso formulário. Eis uma primeira tentativa:

lists/views.py

```
def view_list(request, list_id):
    list_ = List.objects.get(id=list_id)
    form = ItemForm()
    if request.method == 'POST':
        form = ItemForm(data=request.POST)
        if form.is_valid():
            Item.objects.create(text=request.POST['text'], list=list_)
            return redirect(list_)
    return render(request, 'list.html', {'list': list_, "form": form})
```

Com isso, os testes de unidade passam:

```
Ran 23 tests in 0.086s
```

```
OK
```

E quanto aos FTs?

```
ERROR: test_cannot_add_empty_list_items
(functional_tests.test_list_item_validation.ItemValidationTest)
```

```
-----
Traceback (most recent call last):
```

```
File ".../superlists/functional_tests/test_list_item_validation.py", line
15, in test_cannot_add_empty_list_items
```

[...]

```
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
```

```
element: .has-error
```

Não.

Um benefício inesperado: validação gratuita no lado cliente pelo HTML5

O que está acontecendo aqui? Vamos adicionar nosso `time.sleep` usual antes do erro e observar o que está acontecendo (ou iniciar o site manualmente com `manage.py runserver`, se você preferir – veja a Figura 14.1).

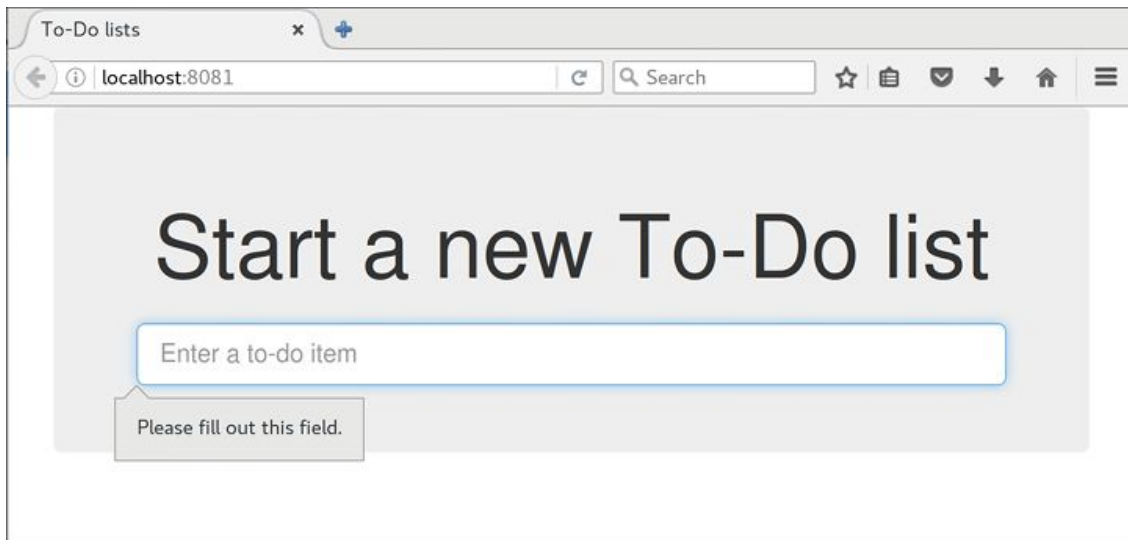


Figura 14.1 – A validação de HTML5 diz não.

Parece que o navegador está impedindo o usuário de sequer submeter a entrada quando essa está vazia.

Isso ocorre porque o Django adicionou o atributo `required` à entrada HTML¹ (observe novamente nossos `printouts as_p()` anteriores, se não acredita em mim). É um novo recurso do HTML5 (<https://developer.mozilla.org/en-US/docs/Web/HTML/Element/Input#attr-required>), e os navegadores atualmente farão alguma validação no lado cliente se virem isso, impedindo que os usuários sequer façam uma submissão de dados

inválidos.

Vamos alterar o nosso FT para que reflita isto:

functional_tests/test_list_item_validation.py (ch11I032)

```
def test_cannot_add_empty_list_items(self):
    # Edith acessa a página inicial e acidentalmente tenta submeter
    # um item vazio na lista. Ela tecla Enter na caixa de entrada vazia
    self.browser.get(self.live_server_url)
    self.get_item_input_box().send_keys(Keys.ENTER)

    # O navegador intercepta a requisição e não carrega a
    # página da lista
    self.wait_for(lambda: self.browser.find_elements_by_css_selector(
        '#id_text:invalid' ❶
    ))

    # Ela começa a digitar um texto para o novo item e o erro desaparece
    self.get_item_input_box().send_keys('Buy milk')
    self.wait_for(lambda: self.browser.find_elements_by_css_selector(
        '#id_text:valid' ❷
    ))

    # E ela pode submeter o item com sucesso
    self.get_item_input_box().send_keys(Keys.ENTER)
    self.wait_for_row_in_list_table('1: Buy milk')

    # De forma perversa, ela agora decide submeter um segundo item
    # em branco na lista
    self.get_item_input_box().send_keys(Keys.ENTER)

    # Novamente, o navegador não concordará
    self.wait_for_row_in_list_table('1: Buy milk')
    self.wait_for(lambda: self.browser.find_elements_by_css_selector(
        '#id_text:invalid'
    ))

    # E ela pode corrigir isso preenchendo o item com um texto
    self.get_item_input_box().send_keys('Make tea')
```

```
self.wait_for(lambda: self.browser.find_elements_by_css_selector(
    '#id_text:valid'
))
self.get_item_input_box().send_keys(Keys.ENTER)
self.wait_for_row_in_list_table('1: Buy milk')
self.wait_for_row_in_list_table('2: Make tea')
```

- ❶ Em vez de verificar nossa mensagem de erro personalizada, fazemos a verificação usando o pseudosseletor de CSS `:invalid`, que o navegador aplica em qualquer entrada HTML5 com dados inválidos.
- ❷ E o oposto é aplicado no caso de entradas válidas.

Viu como nossa função `self.wait_for` está se tornando útil e flexível?

Entretanto, nosso FT parece bem diferente de como era no início, não é mesmo? Tenho certeza de que isso está fazendo com que várias perguntas surjam em sua mente neste momento. Deixe-as de lado por um instante; prometo que falaremos sobre elas. Vamos ver antes se retornamos a um estado em que os testes estejam passando:

```
$ python manage.py test functional_tests
```

```
[...]
```

```
....
```

```
-----  
Ran 4 tests in 12.154s
```

```
OK
```

Um tapinha nas costas

Em primeiro lugar, vamos nos dar um forte tapinha nas costas: acabamos de fazer uma grande mudança em nossa pequena aplicação – aquele campo de entrada, com seu nome e ID, é absolutamente crucial para fazer tudo funcionar. Alteramos sete ou oito arquivos diferentes, fazendo uma refatoração razoavelmente intensa... É o tipo de atividade que, sem testes, me deixaria seriamente preocupado. Com efeito, eu poderia muito bem ter

decidido que não valeria a pena mexer em um código que funcione. Todavia, como temos uma suíte de testes completa, podemos nos aprofundar e sair organizando o código, confiantes por saber que os testes estão aí para identificar qualquer erro que venhamos a cometer. Os testes simplesmente fazem com que seja muito mais provável que continuemos refatorando, reorganizando, tratando, cuidando de nosso código, mantendo tudo arrumado e organizado e limpo e descomplicado e exato e conciso e funcional e bom.

- ~~Remover duplicação na lógica de validação em views~~

Definitivamente é hora de fazer um commit:

```
$ git diff
```

```
$ git commit -am "use form in all views, back to working state"
```

No entanto, desperdiçamos muito tempo?

E quanto à nossa mensagem de erro personalizada? O que dizer de todo aquele esforço para renderizar o formulário em nosso template HTML? Não estamos nem mesmo passando aqueles erros do Django para o usuário se o navegador está interceptando as requisições antes de o usuário sequer fazê-las? E nosso FT não está nem mesmo testando tudo isso!

Bem, você tem toda razão. Contudo, há dois ou três motivos pelos quais todo o nosso tempo não foi desperdiçado. Em primeiro lugar, a validação do lado cliente não é suficiente para garantir que você esteja protegido contra entradas ruins, portanto o lado do servidor sempre será necessário também se você realmente estiver preocupado com a integridade dos dados; usar um formulário é uma boa maneira de encapsular essa lógica.

Além do mais, nem todos os navegadores (*cof, cof – Safari – cof, cof*) implementam totalmente o HTML5, portanto alguns usuários continuarão vendo nossa mensagem de erro personalizada. Se ou quando deixarmos os usuários acessarem nossos dados por meio de uma API (veja o Apêndice F), nossas mensagens de validação

voltarão a ser usadas.

Adicionalmente poderemos reutilizar todo o nosso código de validação e os formulários, além das classes `.has-error` do frontend, no próximo capítulo, quando faremos algumas validações mais sofisticadas, que não poderão ser feitas com a mágica do HTML5.

Porém, você sabe que, mesmo que tudo isso não fosse verdade, você ainda não poderia se autocensurar por ocasionalmente percorrer um beco às escuras enquanto programa. Nenhum de nós é capaz de ver o futuro, e devemos nos concentrar em encontrar a solução correta em vez de olhar para o tempo “desperdiçado” com a solução incorreta.

Usando o método do próprio formulário para salvar dados

Há mais algumas atitudes que podemos tomar para simplificar mais ainda as nossas views. Mencionei que os formulários podem salvar dados no banco de dados para nós. Em nosso caso, isso não funcionará exatamente de imediato porque o item precisa saber em qual lista deverá ser salvo, mas não é difícil corrigir isso.

Como sempre, começaremos com um teste. Somente para ilustrar o problema, vamos ver o que acontecerá se apenas tentarmos chamar `form.save()`:

`lists/tests/test_forms.py (ch11I033)`

```
def test_form_save_handles_saving_to_a_list(self):
    form = ItemForm(data={'text': 'do me'})
    new_item = form.save()
```

O Django não está satisfeito, pois um item deve pertencer a uma lista:

```
django.db.utils.IntegrityError: NOT NULL constraint failed: lists_item.list_id
```

Nossa solução é informar em qual lista o método do formulário deve salvar os dados:

lists/tests/test_forms.py

```
from lists.models import Item, List
[...]

def test_form_save_handles_saving_to_a_list(self):
    list_ = List.objects.create()
    form = ItemForm(data={'text': 'do me'})
    new_item = form.save(for_list=list_)
    self.assertEqual(new_item, Item.objects.first())
    self.assertEqual(new_item.text, 'do me')
    self.assertEqual(new_item.list, list_)
```

Então garantimos que o item seja corretamente salvo no banco de dados, com os atributos corretos:

TypeError: save() got an unexpected keyword argument 'for_list'

Eis o modo como podemos implementar o nosso método personalizado para salvar dados:

lists/forms.py (ch11I035)

```
def save(self, for_list):
    self.instance.list = for_list
    return super().save()
```

O atributo `.instance` em um formulário representa o objeto do banco de dados que está sendo modificado ou criado. Só aprendi isso quando estava escrevendo este capítulo! Há outras maneiras de fazer isso funcionar, incluindo criar manualmente o objeto por conta própria ou usar o argumento `commit=False` para salvar, mas acho que essa é a solução mais limpa. No próximo capítulo, exploraremos uma maneira diferente de fazer um formulário “saber” em qual lista deve atuar:

```
Ran 24 tests in 0.086s
OK
```

Por fim, podemos refatorar nossas views. Começemos com `new_list`:

lists/views.py

```
def new_list(request):
```



```
form = ItemForm(data=request.POST)
if form.is_valid():
    list_ = List.objects.create()
    form.save(for_list=list_)
    return redirect(list_)
else:
    return render(request, 'home.html', {"form": form})
```

Execute novamente os testes para verificar se todos continuam passando:

```
Ran 24 tests in 0.086s
OK
```

É a vez de `view_list`:

lists/views.py

```
def view_list(request, list_id):
    list_ = List.objects.get(id=list_id)
    form = ItemForm()
    if request.method == 'POST':
        form = ItemForm(data=request.POST)
        if form.is_valid():
            form.save(for_list=list_)
            return redirect(list_)
    return render(request, 'list.html', {'list': list_, "form": form})
```

Todos os testes continuam passando:

```
Ran 24 tests in 0.111s
OK
```

e:

```
Ran 4 tests in 14.367s
OK
```

Ótimo! Nossa duas views agora se assemelham muito mais a views “normais” do Django: elas obtêm informações de uma requisição de usuário, combinam esses dados com alguma lógica personalizada ou com informações do URL (`list_id`), passam esses dados para um formulário para validação e possivelmente para salvá-los, e então redirecionam ou renderizam um template.

Formulários e validação são realmente importantes no Django e na programação web em geral, portanto vamos tentar fazer algo um pouco mais complexo no próximo capítulo.

Dicas

Views simples

Se você se vir olhando para views complexas e tendo que escrever muitos testes para elas, é hora de começar a pensar se essa lógica poderia ser transferida para outro lugar: possivelmente para um formulário, como fizemos neste capítulo.

Outro local possível seria para um método personalizado na classe de modelo. E – quando a complexidade da aplicação assim o exigir – fora dos arquivos específicos do Django, em suas próprias classes e funções, que capturem a sua lógica de negócios básica.

Cada teste deve verificar um aspecto

A heurística diz para suspeitar caso haja mais de uma asserção em um teste. Às vezes, duas asserções estão intimamente relacionadas, portanto devem ficar juntas. Com frequência, porém, sua primeira versão preliminar de um teste acaba testando vários comportamentos, e vale a pena reescrevê-la na forma de vários testes. As funções auxiliares podem evitar que os testes fiquem excessivamente extensos.

1 É um novo recurso do Django 1.11.

CAPÍTULO 15

Formulários mais sofisticados

Vamos observar agora o uso de alguns formulários mais sofisticados. Ajudamos nossos usuários a evitar itens em branco na lista, portanto vamos agora ajudá-los a evitar itens duplicados.

Este capítulo dá detalhes mais intrincados da validação de formulários do Django, e você tem minha permissão oficial para ignorá-lo caso já saiba tudo sobre personalização de formulários Django ou se estiver lendo este livro por causa do TDD, e não pelo Django.

Se você ainda está conhecendo o Django, há informações muito boas neste capítulo. Se quiser ignorá-lo e passar adiante, tudo bem também. Não se esqueça de ver rapidamente as informações adicionais sobre a estupidez dos desenvolvedores e a revisão sobre testes de views no final.

Outro FT para itens duplicados

Acrescentamos um segundo método de teste em `ItemValidationTest`:

`functional_tests/test_list_item_validation.py`
(ch13|001)

```
def test_cannot_add_duplicate_items(self):
    # Edith acessa a página inicial e começa uma nova lista
    self.browser.get(self.live_server_url)
    self.get_item_input_box().send_keys('Buy wellies')
    self.get_item_input_box().send_keys(Keys.ENTER)
    self.wait_for_row_in_list_table('1: Buy wellies')

    # Ela tenta acidentalmente inserir um item duplicado
```

```

self.get_item_input_box().send_keys('Buy wellies')
self.get_item_input_box().send_keys(Keys.ENTER)

# Ela vê uma mensagem de erro prestativa
self.wait_for(lambda: self.assertEqual(
    self.browser.find_element_by_css_selector('.has-error').text,
    "You've already got this in your list"
))

```

Por que ter dois métodos de teste em vez de estender um, ou ter um novo arquivo e uma nova classe? É uma questão de avaliar. Esses dois métodos parecem intimamente ligados; ambos são sobre validação no mesmo campo de entrada, portanto parece correto mantê-los no mesmo arquivo. Por outro lado, do ponto de vista lógico, estão suficientemente separados a ponto de ser prático mantê-los em métodos distintos:

```

$ python manage.py test functional_tests.test_list_item_validation
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate
element: .has-error

```

Ran 2 tests in 9.613s

Tudo bem, então sabemos que o primeiro dos dois testes passa no momento. Posso ouvir você perguntando: há uma maneira de executar apenas o teste que falha? Sim, claro que há:

```

$ python manage.py test functional_tests.\
test_list_item_validation.ItemValidationTest.test_cannot_add_duplicate_ite
ms
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate
element: .has-error

```

Evitando duplicações na camada do modelo

A seguir, apresentamos o que realmente queríamos fazer. É um novo teste que verifica se itens duplicados na mesma lista geram um erro:

lists/tests/test_models.py (ch09I028)

```
def test_duplicate_items_are_invalid(self):
    list_ = List.objects.create()
    Item.objects.create(list=list_, text='bla')
    with self.assertRaises(ValidationError):
        item = Item(list=list_, text='bla')
        item.full_clean()
```

Aproveitando a ocasião, adicionamos outro teste para garantir que não vamos exagerar em nossas restrições de integridade:

lists/tests/test_models.py (ch09I029)

```
def test_CAN_save_same_item_to_different_lists(self):
    list1 = List.objects.create()
    list2 = List.objects.create()
    Item.objects.create(list=list1, text='bla')
    item = Item(list=list2, text='bla')
    item.full_clean() # não deve gerar erro
```

Sempre gosto de colocar um pequeno comentário em testes que verifiquem se um caso de uso em particular *não* deve gerar um erro; caso contrário, poderá ser difícil ver o que está sendo testado:

```
AssertionError: ValidationError not raised
```

Se quisermos que o teste deliberadamente dê errado, podemos fazer o seguinte:

lists/models.py (ch09I030)

```
class Item(models.Model):
    text = models.TextField(default="", unique=True)
    list = models.ForeignKey(List, default=None)
```

Isso nos permite verificar se o nosso segundo teste realmente identifica esse problema:

```
Traceback (most recent call last):
  File "../superlists/lists/tests/test_models.py", line 62, in
test_CAN_save_same_item_to_different_lists
    item.full_clean() # não deve gerar erro
  [...]
django.core.exceptions.ValidationError: {'text': ['Item with this Text already
```



```
exists.'])}
```

Informações extras sobre quando testar a estupidez do desenvolvedor

Um dos pontos que deve ser avaliado nos testes é quando devemos escrever testes que soem como “verificar se não fizemos algo estúpido”. Em geral, você deve prestar atenção nisso. Em nosso caso, escrevemos um teste para verificar se não podemos salvar itens duplicados na mesma lista. A maneira mais simples de fazer esse teste passar – o modo como escreveríamos o menor número de linhas de código – seria fazer com que fosse impossível salvar *qualquer* item duplicado. Isso justifica escrever outro teste, apesar de ser algo “estúpido” ou “errado” para implementar.

No entanto, não podemos escrever testes para todas as maneiras possíveis pelas quais poderíamos implementar algo errado. Se você tiver uma função que some dois números, poderá escrever dois testes:

```
assert adder(1, 1) == 2
```

```
assert adder(2, 1) == 3
```

Porém, você tem o direito de supor que a implementação não está sendo deliberadamente enganadora ou perversa:

```
def adder(a, b):
    # código improvável!
    if a == 3:
        return 666
    else:
        return a + b
```

Uma forma de expressar isso é acreditar que você não fará nada *deliberadamente* estúpido, mas não poderá garantir que não fará algo *acidentalmente* estúpido.

Assim como ModelForms, os modelos têm uma class Meta, e é aí que podemos implementar uma restrição que diga que um item deve ser único para uma lista em particular ou, em outras palavras, que text e list em conjunto devem ser únicos:

lists/models.py (ch09I031)

```
class Item(models.Model):
    text = models.TextField(default="")
    list = models.ForeignKey(List, default=None)

    class Meta:
        unique_together = ('list', 'text')
```

Talvez, a essa altura, você queira dar uma espiada rápida na documentação do Django sobre atributos Meta de modelos (<https://docs.djangoproject.com/en/1.11/ref/models/options/>).

Uma pequena digressão sobre ordenação de querysets e representações de string

Quando executamos os testes, eles revelam uma falha inesperada:

```
=====
=====
FAIL: test_saving_and_retrieving_items
(lists.tests.test_models.ListAndItemModelsTest)
-----
Traceback (most recent call last):
  File ".../superlists/lists/tests/test_models.py", line 31, in
```

```
test_saving_and_retrieving_items
    self.assertEqual(first_saved_item.text, 'The first (ever) list item')
AssertionError: 'Item the second' != 'The first (ever) list item'
- Item the second
[...]
```



Conforme a sua plataforma e sua instalação de SQLite, talvez você não veja esse erro. Pode prosseguir, de qualquer modo; o código e os testes, por si só, são interessantes.

Isso é um pouco intrigante. Um pouco de depuração baseado em prints:

lists/tests/test_models.py

```
first_saved_item = saved_items[0]
print(first_saved_item.text)
second_saved_item = saved_items[1]
print(second_saved_item.text)
self.assertEqual(first_saved_item.text, 'The first (ever) list item')
```

nos mostra o seguinte:

```
.....Item the second
The first (ever) list item
F.....
```

Parece que nossa restrição de unicidade causou confusão na ordem default de consultas como `Item.objects.all()`. Apesar de já termos um teste com falha, é melhor acrescentar um novo teste que verifique explicitamente a ordenação:

lists/tests/test_models.py (ch09I032)

```
def test_list_ordering(self):
    list1 = List.objects.create()
    item1 = Item.objects.create(list=list1, text='i1')
    item2 = Item.objects.create(list=list1, text='item 2')
    item3 = Item.objects.create(list=list1, text='3')
    self.assertEqual(
        Item.objects.all(),
        [item1, item2, item3])
```

)

Isso resulta em uma nova falha não muito legível:

```
AssertionError: <QuerySet [<Item: Item object>, <Item: Item object>, <Item: Item object>]> != [<Item: Item object>, <Item: Item object>, <Item: Item object>]
```

Precisamos de uma representação melhor em string para nossos objetos. Vamos acrescentar outro teste de unidade:



Geralmente você deve tomar cuidado ao adicionar mais testes em falha quando já tiver alguns – isso faz com que a leitura da saída dos testes seja muito mais complicada e, de modo geral, deixará você mais apreensivo. Conseguiremos voltar para um estado funcional em algum momento? Nesse caso, todos os testes são bem simples, portanto não estou preocupado.

lists/tests/test_models.py (ch13I008)

```
def test_string_representation(self):
    item = Item(text='some text')
    self.assertEqual(str(item), 'some text')
```

Eis o resultado:

```
AssertionError: 'Item object' != 'some text'
```

Além das duas outras falhas. Vamos começar a corrigi-las agora:

lists/models.py (ch09I034)

```
class Item(models.Model):
    [...]

    def __str__(self):
        return self.text
```



Em versões do Django com Python 2.x, o método de representação de string costumava ser `__unicode__`. Assim como para muitos aspectos da manipulação de strings, isso foi simplificado em Python 3. Consulte a documentação do Django (<https://docs.djangoproject.com/en/1.11/topics/python3/#str-and-unicode-methods>).

Restaram agora duas falhas, e o teste de ordenação tem uma mensagem de falha mais legível:

```
AssertionError: <QuerySet [<Item: i1>, <Item: item 2>, <Item: 3>]> != [<Item: i1>, <Item: item 2>, <Item: 3>]
```

Podemos corrigir isso na class Meta:

lists/models.py (ch09I035)

```
class Meta:
    ordering = ('id',)
    unique_together = ('list', 'text')
```

Funciona?

```
AssertionError: <QuerySet [<Item: i1>, <Item: item 2>, <Item: 3>]> != [<Item: i1>, <Item: item 2>, <Item: 3>]
```

Hã? Funcionou; podemos ver que os itens *estão* na mesma ordem, mas os testes estão confusos. Na verdade, fico sempre deparando com esse problema – as querysets do Django não se comparam bem com listas. Podemos corrigir isso convertendo a queryset em uma lista¹ em nosso teste:

lists/tests/test_models.py (ch09I036)

```
self.assertEqual(
    list(Item.objects.all()),
    [item1, item2, item3]
)
```

Funciona; temos uma suíte de testes passando por completo:

OK

Reescrevendo o teste antigo do modelo

Aquele teste longo do modelo por acaso nos ajudou a encontrar um bug inesperado, mas agora é hora de reescrevê-lo. Eu o escrevi em um estilo bem verboso para introduzir o ORM do Django; porém, na verdade, agora que temos o teste explícito para ordenação, podemos ter a mesma abrangência com dois testes bem menores. Apague `test_saving_and_retrieving_items` e substitua por:

lists/tests/test_models.py (ch13I010)

```
class ListAndItemModelsTest(TestCase):

    def test_default_text(self):
        item = Item()
        self.assertEqual(item.text, "")

    def test_item_is_related_to_list(self):
        list_ = List.objects.create()
        item = Item()
        item.list = list_
        item.save()
        self.assertIn(item, list_.item_set.all())
```

[...]

Isso é de fato mais do que suficiente – uma verificação dos valores default dos atributos em um objeto de modelo recém-inicializado é o bastante para fazer uma verificação de sanidade e saber que, provavelmente, definimos alguns campos em *models.py*. O teste de “item está relacionado à lista” é um verdadeiro teste de “segurança” para garantir que nosso relacionamento de chave estrangeira funciona.

Aproveitando a ocasião, podemos dividir esse arquivo em testes para Item e testes para List (há apenas um desse último, `test_get_absolute_url`):

lists/tests/test_models.py (ch13I011)

```
class ItemModelTest(TestCase):
```

```
    def test_default_text(self):  
        [...]
```

```
class ListModelTest(TestCase):
```

```
    def test_get_absolute_url(self):
```


[...]

Os testes estão mais limpos e organizados:

```
$ python manage.py test lists
```

[...]

```
Ran 29 tests in 0.092s
```

OK

Alguns erros de integridade aparecem quando salvamos os dados

Uma última informação extra antes de prosseguir. Você se lembra de que, no Capítulo 13, mencionei que alguns erros de integridade de dados *são* identificados quando esses são salvos? Tudo depende do fato de a restrição de integridade estar realmente sendo imposta pelo banco de dados.

Experimente executar `makemigrations` e você verá que o Django deseja adicionar a restrição `unique_together` no próprio banco de dados, em vez de tê-la simplesmente como uma restrição na camada de aplicação:

```
$ python manage.py makemigrations
```

```
Migrations for 'lists':
```

```
lists/migrations/0005_auto_20140414_2038.py
```

```
- Change Meta options on item
```

```
- Alter unique_together for item (1 constraint(s))
```

Se alterarmos nosso teste de duplicação para executar um `.save` em vez de um `.full_clean`...

lists/tests/test_models.py

```
def test_duplicate_items_are_invalid(self):
    list_ = List.objects.create()
    Item.objects.create(list=list_, text='bla')
    with self.assertRaises(ValidationError):
        item = Item(list=list_, text='bla')
        # item.full_clean()
        item.save()
```

O resultado será este:

```
ERROR: test_duplicate_items_are_invalid
(lists.tests.test_models.ItemModelTest)
[...]
return Database.Cursor.execute(self, query, params)
sqlite3.IntegrityError: UNIQUE constraint failed: lists_item.list_id,
lists_item.text
[...]
django.db.utils.IntegrityError: UNIQUE constraint failed: lists_item.list_id,
lists_item.text
```

Podemos ver que o erro se propaga a partir do SQLite, e é um erro diferente daquele que queremos; é um `IntegrityError`, e não um `ValidationError`.

Vamos reverter nossas alterações no teste, e veremos que todos os testes passam novamente:

```
$ python manage.py test lists
[...]
Ran 29 tests in 0.092s
OK
```

Agora é hora de fazer commit de nossas mudanças na camada do modelo:

```
$ git status # deve mostrar mudanças em testes + modelos e a nova migração
# vamos dar um nome melhor à nossa nova migração
$ mv lists/migrations/0005_auto*
lists/migrations/0005_list_item_unique_together.py
$ git add lists
$ git diff --staged
$ git commit -am "Implement duplicate item validation at model layer"
```

Fazendo experimentos com validação de itens duplicados na camada de views

Vamos tentar executar o nosso FT, somente para ver em que situação nos encontramos:

```
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate
```

element: .has-error

Caso você não tenha visto à medida que a saída foi apresentada, o site está “quinhentando”.² Um teste de unidade rápido no nível da view deve esclarecer isso:

lists/tests/test_views.py (ch13I014)

```
class ListViewTest(TestCase):
    [...]

    def test_for_invalid_input_shows_error_on_page(self):
        [...]

    def test_duplicate_item_validation_errors_end_up_on_lists_page(self):
        list1 = List.objects.create()
        item1 = Item.objects.create(list=list1, text='textey')
        response = self.client.post(
            f'/lists/{list1.id}/',
            data={'text': 'textey'}
        )

        expected_error = escape("You've already got this in your list")
        self.assertContains(response, expected_error)
        self.assertTemplateUsed(response, 'list.html')
        self.assertEqual(Item.objects.all().count(), 1)
```

Eis o resultado:

```
django.db.utils.IntegrityError: UNIQUE constraint failed: lists_item.list_id,
lists_item.text
```

Queremos evitar erros de integridade! De modo ideal, queremos que a chamada a `is_valid`, de alguma forma, perceba o erro de duplicação antes que tentemos salvar os dados; para isso, porém, nosso formulário precisará saber com antecedência para qual lista ele está sendo usado.

Vamos inserir um `skip` nesse teste por enquanto:

lists/tests/test_views.py (ch13I015)

```
from unittest import skip
```

[...]

```
@skip
def test_duplicate_item_validation_errors_end_up_on_lists_page(self):
```

Um formulário mais complexo para tratar a validação de unicidade

O formulário para criar uma nova lista só precisa conhecer uma informação, isto é, o texto do novo item. Um formulário que valide se os itens da lista são únicos deve saber também qual é a lista. Assim como sobrescrevemos o método para salvar dados em nosso `ItemForm`, desta vez sobrescreveremos o construtor de nossa nova classe de formulário para que ele saiba a qual lista se aplica.

Duplicaremos nossos testes do formulário anterior, adaptando-os levemente:

lists/tests/test_forms.py (ch13|016)

```
from lists.forms import (
    DUPLICATE_ITEM_ERROR, EMPTY_ITEM_ERROR,
    ExistingListItemForm, ItemForm
)
[...]
```

```
class ExistingListItemFormTest(TestCase):

    def test_form_renders_item_text_input(self):
        list_ = List.objects.create()
        form = ExistingListItemForm(for_list=list_)
        self.assertIn('placeholder="Enter a to-do item"', form.as_p())

    def test_form_validation_for_blank_items(self):
        list_ = List.objects.create()
        form = ExistingListItemForm(for_list=list_, data={'text': ''})
        self.assertFalse(form.is_valid())
        self.assertEqual(form.errors['text'], [EMPTY_ITEM_ERROR])
```

```

def test_form_validation_for_duplicate_items(self):
    list_ = List.objects.create()
    Item.objects.create(list=list_, text='no twins!')
    form = ExistingListItemForm(for_list=list_, data={'text': 'no twins!'})
    self.assertFalse(form.is_valid())
    self.assertEqual(form.errors['text'], [DUPLICATE_ITEM_ERROR])

```

Em seguida, iteramos por alguns ciclos de TDD até que tenhamos um formulário com um construtor personalizado que simplesmente ignore o seu argumento `for_list`. (Não mostrarei todos eles, mas estou certo de que você os fará, não é mesmo? Lembre-se: o Goat está vendo tudo.)

lists/forms.py (ch09|071)

```

DUPLICATE_ITEM_ERROR = "You've already got this in your list"
[...]
class ExistingListItemForm(forms.models.ModelForm):
    def __init__(self, for_list, *args, **kwargs):
        super().__init__(*args, **kwargs)

```

Neste ponto, nosso erro deve ser:

```
ValueError: ModelForm has no model class specified.
```

Vamos então ver se fazê-lo herdar de nosso formulário existente ajuda:

lists/forms.py (ch09|072)

```

class ExistingListItemForm(ItemForm):
    def __init__(self, for_list, *args, **kwargs):
        super().__init__(*args, **kwargs)

```

Sim, isso faz com que reste apenas uma falha:

```

FAIL: test_form_validation_for_duplicate_items
(lists.tests.test_forms.ExistingListItemFormTest)
self.assertFalse(form.is_valid())
AssertionError: True is not false

```

O próximo passo exige um pouco de conhecimento do funcionamento interno do Django, mas você pode ler sobre esse assunto em sua documentação sobre validação de modelos

(<https://docs.djangoproject.com/en/1.11/ref/models/instances/#validating-objects>) e validação de formulários (<https://docs.djangoproject.com/en/1.11/ref/forms/validation/>).

O Django utiliza um método chamado `validate_unique`, tanto em formulários quanto em modelos, e podemos usar ambos em conjunto com o atributo `instance`:

lists/forms.py

```
from django.core.exceptions import ValidationError
[...]

class ExistingListItemForm(ItemForm):

    def __init__(self, for_list, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.instance.list = for_list

    def validate_unique(self):
        try:
            self.instance.validate_unique()
        except ValidationError as e:
            e.error_dict = {'text': [DUPLICATE_ITEM_ERROR]}
            self._update_errors(e)
```

Há um pouco do vodu do Django aí, mas, basicamente, tomamos o erro de validação, ajustamos sua mensagem de erro e então o passamos de volta para o formulário.

E conseguimos! Façamos um rápido commit:

```
$ git diff
$ git commit -a
```

Usando o formulário de item de lista existente na view de lista

Vamos ver agora se podemos fazer esse formulário funcionar em nossa view.

Removemos o skip e, aproveitando a ocasião, podemos usar nossa nova constante. Que organizado!

lists/tests/test_views.py (ch13I049)

```
from lists.forms import (
    DUPLICATE_ITEM_ERROR, EMPTY_ITEM_ERROR,
    ExistingListItemForm, ItemForm,
)
[...]

def test_duplicate_item_validation_errors_end_up_on_lists_page(self):
    [...]
    expected_error = escape(DUPLICATE_ITEM_ERROR)
```

Isso traz de volta o nosso erro de integridade:

```
django.db.utils.IntegrityError: UNIQUE constraint failed: lists_item.list_id,
lists_item.text
```

Nossa correção para isso é passar a usar a nova classe de formulário. Antes de implementá-la, vamos localizar os testes em que verificamos a classe do formulário e ajustá-los:

lists/tests/test_views.py (ch13I050)

```
class ListViewTest(TestCase):
    [...]

    def test_displays_item_form(self):
        list_ = List.objects.create()
        response = self.client.get(f'/lists/{list_.id}/')
        self.assertIsInstance(response.context['form'], ExistingListItemForm)
        self.assertContains(response, 'name="text"')

    [...]

    def test_for_invalid_input_passes_form_to_template(self):
        response = self.post_invalid_input()
        self.assertIsInstance(response.context['form'], ExistingListItemForm)
```

Eis o resultado:

```
AssertionError: <ItemForm bound=False, valid=False, fields=(text)> is not an
```

instance of <class 'lists.forms.ExistingListItemForm'>

Assim podemos adaptar a view:

lists/views.py (ch13|051)

```
from lists.forms import ExistingListItemForm, ItemForm
[...]
def view_list(request, list_id):
    list_ = List.objects.get(id=list_id)
    form = ExistingListItemForm(for_list=list_)
    if request.method == 'POST':
        form = ExistingListItemForm(for_list=list_, data=request.POST)
        if form.is_valid():
            form.save()
        [...]
```

Com isso, corrigimos *quase* tudo, exceto uma falha inesperada:

```
TypeError: save() missing 1 required positional argument: 'for_list'
```

Nosso método personalizado para salvar dados do ItemForm pai não é mais necessário. Vamos criar um teste de unidade rápido para isso:

lists/tests/test_forms.py (ch13|053)

```
def test_form_save(self):
    list_ = List.objects.create()
    form = ExistingListItemForm(for_list=list_, data={'text': 'hi'})
    new_item = form.save()
    self.assertEqual(new_item, Item.objects.all()[0])
```

Podemos fazer o nosso formulário chamar o método de salvar dados do avô:

lists/forms.py (ch13|054)

```
def save(self):
    return forms.models.ModelForm.save(self)
```



Eis uma opinião pessoal: eu poderia ter usado `super`, mas prefiro não o utilizar quando ele exige argumentos, digamos, para obter um método do avô. Acho o `super()` sem argumentos de Python 3 incrível para obter o pai

imediatamente. Tudo o mais é muito suscetível a erros e, além disso, acho feio. Você tem direito a uma opinião diferente.

Conseguimos! Todos os testes de unidade estão passando:

```
$ python manage.py test lists
```

```
[...]
```

```
Ran 34 tests in 0.082s
```

OK

O mesmo vale para nosso FT de validação:

```
$ python manage.py test functional_tests.test_list_item_validation
```

```
[...]
```

```
..
```

```
-----  
Ran 2 tests in 12.048s
```

OK

Como verificação final, vamos executar novamente *todos* os FTs:

```
$ python manage.py test functional_tests
```

```
[...]
```

```
.....
```

```
-----  
Ran 5 tests in 19.048s
```

OK

Viva! É hora de um commit final e uma conclusão sobre o que aprendemos sobre testar views nos últimos capítulos.

Conclusão: o que aprendemos sobre testar o Django

Estamos agora em um ponto em que nossa aplicação se parece muito mais com uma aplicação Django “padrão”, e ela implementa as três camadas comuns do Django: modelos, formulários e views. Já não temos mais testes em estilo “bicicleta com rodinhas”, e nosso código se parece muito com um código que ficaríamos satisfeitos de ver em uma aplicação de verdade.

Temos um arquivo de testes de unidade para cada um de nossos arquivos principais de código-fonte. Eis uma revisão do maior (e de mais alto nível), *test_views* (a listagem mostra somente os testes e as asserções principais):

O que testar em views

lists/tests/test_views.py

```
class ListViewTest(TestCase):
    def test_uses_list_template(self):
        response = self.client.get(f'/lists/{list_.id}/') ❶
        self.assertTemplateUsed(response, 'list.html') ❷
    def test_passes_correct_list_to_template(self):
        self.assertEqual(response.context['list'], correct_list) ❸
    def test_displays_item_form(self):
        self.assertIsInstance(response.context['form'], ExistingListItemForm) ❹
        self.assertContains(response, 'name="text"')
    def test_displays_only_items_for_that_list(self):
        self.assertContains(response, 'itemey 1') ❺
        self.assertContains(response, 'itemey 2') ❺
        self.assertNotContains(response, 'other list item 1') ❺
    def test_can_save_a_POST_request_to_an_existing_list(self):
        self.assertEqual(Item.objects.count(), 1) ❻
        self.assertEqual(new_item.text, 'A new item for an existing list') ❻
    def test_POST_redirects_to_list_view(self):
        self.assertRedirects(response, f'/lists/{correct_list.id}/') ❻
    def test_for_invalid_input_nothing_saved_to_db(self):
        self.assertEqual(Item.objects.count(), 0) ❻
    def test_for_invalid_input_renders_list_template(self):
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'list.html') ❻
    def test_for_invalid_input_passes_form_to_template(self):
        self.assertIsInstance(response.context['form'], ExistingListItemForm) ❼
    def test_for_invalid_input_shows_error_on_page(self):
        self.assertContains(response, escape(EMPTY_ITEM_ERROR)) ❼
    def test_duplicate_item_validation_errors_end_up_on_lists_page(self):
        self.assertContains(response, expected_error)
        self.assertTemplateUsed(response, 'list.html')
        self.assertEqual(Item.objects.all().count(), 1)
```

- ① Utilize o Django Test Client.
- ② Verifique o template usado. Então verifique cada item no contexto do template.
- ③ Verifique se os objetos são do tipo correto ou se os querysets contêm os itens corretos.
- ④ Verifique se os formulários são da classe correta.
- ⑤ Pense na lógica de testes dos templates: qualquer for ou if pode merecer um teste mínimo.
- ⑥ Para requisições POST, certifique-se de testar tanto o caso válido quanto o caso inválido.
- ⑦ Opcionalmente faça uma verificação de sanidade para saber se seu formulário é renderizado e se seus erros são exibidos.

Por que esses pontos? Vá direto para o Apêndice B, no qual mostrarei como eles são suficientes para garantir que nossas views continuem corretas caso façamos a sua refatoração a fim de começar a usar views baseadas em classe. A seguir, tentaremos deixar nossa validação de dados mais simpática usando um pouco de código do lado cliente. Sim, você sabe o que isso significa...

-
- 1 Você também poderia dar uma olhada em `assertSequenceEqual` de `unittest` e em `assertQuerysetEqual` das ferramentas de teste do Django, embora confesso que, quando vi `assertQuerysetEqual` pela última vez, fiquei bem perplexo...
 - 2 Está mostrando um código 500 de erro do servidor. Você precisa se acostumar com o jargão!

CAPÍTULO 16

Mergulhando os pés, cautelosamente, no JavaScript

Se o bom Deus quisesse que nos divertíssemos, não teria nos concedido sua preciosa dádiva da incessante miséria.

– John Calvin (conforme retratado em Calvin and the Chipmunks (<http://onemillionpoints.blogspot.co.uk/2008/08/calvin-and-chipmunks.html>))

Nossa nova lógica de validação é boa, mas não seria interessante se as mensagens de erro de itens duplicados desaparecessem depois que o usuário começasse a corrigir o problema? Exatamente como fazem nossos belos erros de validação do HTML5? Para isso, precisaríamos de uma pequena dose de JavaScript.

Definitivamente estamos mimados por programar diariamente em uma linguagem tão agradável quanto Python. JavaScript é a nossa punição. Para um desenvolvedor web, porém, não há como evitá-lo. Portanto, vamos mergulhar os dedos dos pés, com muito cuidado.



Estou supondo que você conhece o básico sobre a sintaxe de JavaScript. Se ainda não leu *JavaScript: The Good Parts*, adquira uma cópia imediatamente! Não é um livro muito longo.

Começando com um FT

Vamos acrescentar um novo teste funcional na classe `ItemValidationTest`:

functional_tests/test_list_item_validation.py (ch14I001)

```
def test_error_messages_are_cleared_on_input(self):
    # Edith inicia uma lista e provoca um erro de validação:
    self.browser.get(self.live_server_url)
    self.get_item_input_box().send_keys('Banter too thick')
    self.get_item_input_box().send_keys(Keys.ENTER)
    self.wait_for_row_in_list_table('1: Banter too thick')
    self.get_item_input_box().send_keys('Banter too thick')
    self.get_item_input_box().send_keys(Keys.ENTER)

    self.wait_for(lambda: self.assertTrue( ❶
        self.browser.find_element_by_css_selector('.has-error').is_displayed() ❷
    ))

    # Ela começa a digitar na caixa de entrada para limpar o erro
    self.get_item_input_box().send_keys('a')

    # Ela fica satisfeita ao ver que a mensagem de erro desaparece
    self.wait_for(lambda: self.assertFalse(
        self.browser.find_element_by_css_selector('.has-error').is_displayed() ❷
    ))
```

- ❶ Usamos outra de nossas chamadas a `wait_for`, dessa vez com `assertTrue`.
- ❷ `is_displayed()` nos informa se um elemento está visível ou não. Não podemos simplesmente depender de verificar se o elemento está presente no DOM, pois agora estamos começando a ocultar elementos.

Esse teste falha de modo apropriado, mas, antes de prosseguir: três acertos e refatorar! Temos diversos lugares onde encontramos o elemento de erro usando CSS. Vamos passar esse código para uma função auxiliar:

functional_tests/test_list_item_validation.py (ch14I002)

```
class ItemValidationTest(FunctionalTest):
```

```
def get_error_element(self):
    return self.browser.find_element_by_css_selector('.has-error')
```

[...]



Gosto de manter as funções auxiliares na classe de FT que as utiliza, e somente as promover para a classe-base quando elas forem realmente necessárias em outro lugar. Isso evita que a classe-base fique muito entulhada. YAGNI (You Ain't Gonna Need It, isto é, Você Não Vai Precisar Disso).

Fazemos, então, três substituições em *test_list_item_validation*, assim:

functional_tests/test_list_item_validation.py (ch14I003)

```
self.wait_for(lambda: self.assertEqual(
    self.get_error_element().text,
    "You've already got this in your list"
))
[...]
```

```
self.wait_for(lambda: self.assertTrue(
    self.get_error_element().is_displayed()
))
[...]
```

```
self.wait_for(lambda: self.assertFalse(
    self.get_error_element().is_displayed()
))
```

Temos uma falha esperada:

```
$ python manage.py test functional_tests.test_list_item_validation
[...]
```

```
self.get_error_element().is_displayed()
AssertionError: True is not false
```

Podemos fazer commit desse código como nossa primeira tentativa de FT.

Configurando um executor de testes básico de JavaScript

Escolher suas ferramentas de teste no mundo Python e Django é razoavelmente simples. O pacote `unittest` da biblioteca-padrão é perfeitamente apropriado, e o executor de testes do Django também é uma boa opção default. Há algumas alternativas por aí – o `nose` (<http://nose.readthedocs.org/>) é popular, o `Green` (<https://github.com/CleanCut/green>) é a novidade do momento no mercado, e, pessoalmente, acho o `pytest` (<http://pytest.org/>) bem impressionante. Porém há uma opção default clara, e é muito boa.¹

Não é bem assim no mundo de JavaScript! Usamos YUI e Jest no trabalho, mas achei melhor dar uma olhada por aí para ver se havia alguma ferramenta nova no mercado. Fiquei impressionado com as opções – `jsUnit`, `QUnit`, `Mocha`, `Chutzpah`, `Karma`, `Jasmine` e muito mais. E não termina por aí: eu havia quase me decidido por uma delas, o `Mocha`², quando descobri que agora tinha que escolher um *framework de asserção* e um *gerador de relatórios* (reporter) e talvez uma *biblioteca de simulação*, e parecia não ter fim!

No final, decidi que devíamos usar a `QUnit` (<http://qunitjs.com/>) porque ela é simples, tem uma aparência semelhante aos testes de unidade de Python e funciona bem com a `jQuery`.

Crie um diretório chamado *tests* em *lists/static* e faça download dos arquivos JavaScript e CSS da `QUnit` nesse local. Também colocaremos um arquivo chamado *tests.html* aí:

```
$ tree lists/static/tests/  
lists/static/tests/  
├── qunit-2.0.1.css  
├── qunit-2.0.1.js  
└── tests.html
```

O boilerplate para um arquivo HTML da `QUnit` tem o seguinte aspecto, incluindo um teste de fumaça (smoke test):

```
<!DOCTYPE html>  
<html>
```



```
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title>Javascript tests</title>
  <link rel="stylesheet" href="qunit-2.0.1.css">
</head>
<body>
  <div id="qunit"></div>
  <div id="qunit-fixture"></div>
  <script src="qunit-2.0.1.js"></script>
  <script>
QUnit.test("smoke test", function (assert) {
  assert.equal(1, 1, "Maths works!");
});
  </script>
</body>
</html>
```

Ao dissecar esse código, os aspectos importantes a serem percebidos são o fato de que trazemos *qunit-2.0.1.js* usando a primeira tag `<script>` e, então, utilizamos a segunda para escrever o corpo principal de testes.

Se o arquivo for aberto com o seu navegador web (não é preciso executar o servidor de desenvolvimento; basta encontrar o arquivo no disco), você deverá ver algo parecido com o que mostra a Figura 16.1.

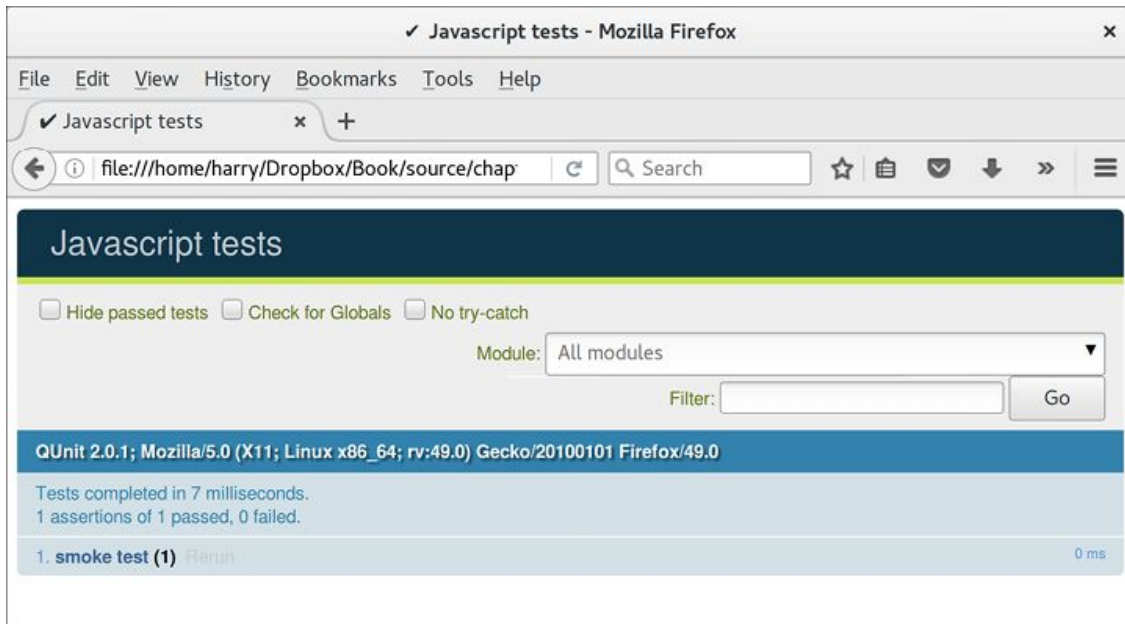


Figura 16.1 – Tela básica da QUnit.

Ao observar o teste propriamente dito, veremos muitas semelhanças com os testes de Python que escrevemos até agora:

```
QUnit.test("smoke test", function (assert) { ❶  
    assert.equal(1, 1, "Maths works!"); ❷  
});
```

- ❶ A função `QUnit.test` define um caso de teste, um pouco como `def test_something(self)` fazia em Python. Seu primeiro argumento é o nome do teste e o segundo é uma função para o corpo do teste.
- ❷ A função `assert.equal` é uma asserção; de modo muito semelhante a `assertEqual`, ela compara dois argumentos. De modo diferente de Python, porém, a mensagem é exibida tanto para falhas quanto para sucessos, portanto ela deve ser redigida como positiva, e não como negativa.

Por que não tentamos alterar esses argumentos para ver uma falha proposital?

Usando a jQuery e a div de fixture

Vamos nos familiarizar melhor com o que nosso framework de testes é capaz de fazer e começar a usar um pouco da jQuery –

uma biblioteca quase indispensável, que oferece uma API compatível com vários navegadores para manipular o DOM.



Caso você ainda não tenha visto a jQuery, tentarei explicá-la à medida que avançarmos, apenas o suficiente para que você não se sinta totalmente perdido; este, porém, não é um tutorial sobre a jQuery. Talvez você ache conveniente investir uma ou duas horas investigando-a em algum ponto deste capítulo.

Faça o download da jQuery mais recente em jquery.com (<https://jquery.com/download/>) e salve-a na pasta *lists/static*.

Em seguida, vamos começar a usá-la em nosso arquivo de testes, além de adicionar dois elementos HTML. Começaremos verificando se podemos exibir e ocultar um elemento, e escrever algumas asserções sobre sua visibilidade:

lists/static/tests/tests.html

```
<div id="qunit-fixture"></div>

<form> ❶
  <input name="text" />
  <div class="has-error">Error text</div>
</form>

<script src="../jquery-3.1.1.min.js"></script> ❷
<script src="qunit-2.0.1.js"></script>
<script>

QUnit.test("smoke test", function (assert) {
  assert.equal($('.has-error').is(':visible'), true); ❸❹
  $('.has-error').hide(); ❺
  assert.equal($('.has-error').is(':visible'), false); ❻
});

</script>
```

- ❶ O `<form>` e seu conteúdo existem para representar o que estará na página real de lista.

- ② É nesse ponto que carregamos a jQuery.
- ③ A mágica da jQuery começa aqui! \$ é o seu canivete suíço. É usado para localizar partes do DOM. Seu primeiro argumento é um seletor CSS; nesse ponto, estamos dizendo-lhe que encontre todos os elementos que tenham a classe “has-error”. Ele devolve um objeto que representa um ou mais elementos do DOM. Esse, por sua vez, tem diversos métodos úteis que nos permitem manipular ou descobrir informações sobre esses elementos.
- ④ Um deles é .is, que pode nos dizer se um elemento corresponde a uma propriedade CSS em particular. Nesse caso, usamos :visible para verificar se o elemento está sendo exibido ou se está oculto.
- ⑤ Então, usamos o método .hide() da jQuery para ocultar a div. Nos bastidores, ela define dinamicamente um style="display: none" no elemento.
- ⑥ Por fim, verificamos se isso funcionou, com um segundo assert.equal.

Se você atualizar o navegador, deverá ver que todos os testes passam:

Resultados esperados da QUnit no navegador

```
2 assertions of 2 passed, 0 failed.  
1. smoke test (2)
```

É hora de ver como as fixtures funcionam. Vamos simplesmente duplicar esse teste:

lists/static/tests/tests.html

```
<script>  
  
QUnit.test("smoke test", function (assert) {  
    assert.equal($('.has-error').is(':visible'), true);  
    $('.has-error').hide();  
    assert.equal($('.has-error').is(':visible'), false);  
});  
QUnit.test("smoke test 2", function (assert) {  
    assert.equal($('.has-error').is(':visible'), true);
```

```

$('.has-error').hide();
assert.equal($('.has-error').is(':visible'), false);
});

</script>

```

De modo um pouco inesperado, descobrimos que um deles falha – veja a Figura 16.2.

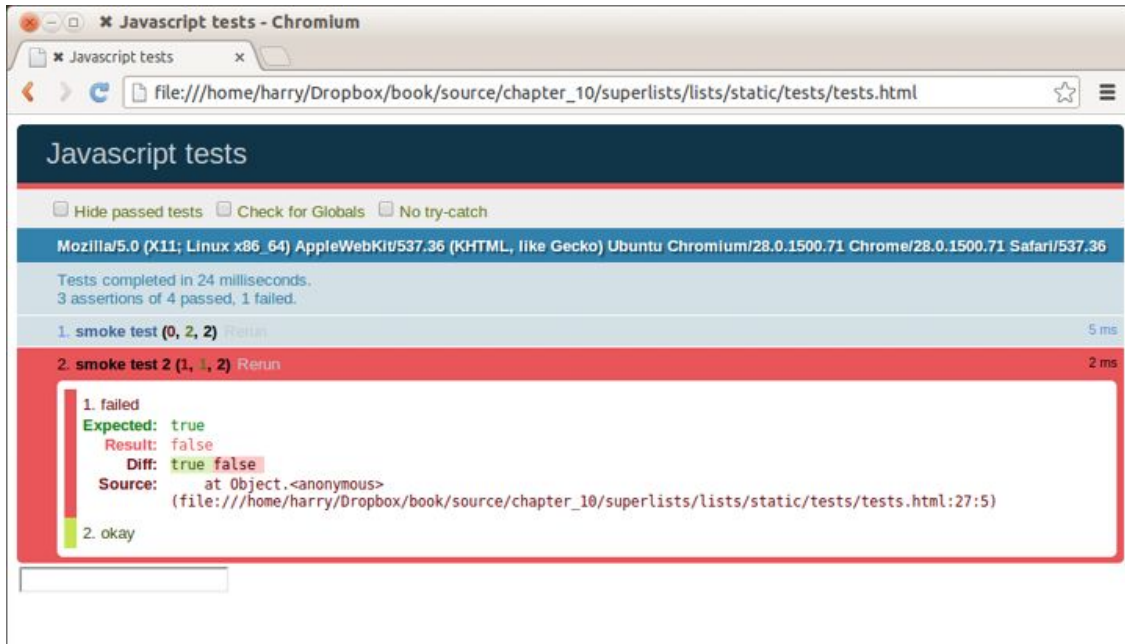


Figura 16.2 – Um dos dois testes está falhando.

O que acontece aqui é que o primeiro teste oculta a div de erro; assim, quando o segundo teste é executado, ela começa invisível.



Os testes da QUnit não executam em uma ordem previsível, portanto você não pode contar com o fato de o primeiro teste executar antes do segundo. Experimente pressionar o botão de atualização algumas vezes e verá que o teste que falha muda...

Precisamos de alguma forma de fazer uma limpeza antes os testes; um pouco como fazem `setUp` e `tearDown`, ou como o executor de testes do Django iniciaria o banco de dados entre cada teste. A div `qunit-fixture` é o que estamos procurando. Transfira o formulário para lá:

lists/static/tests/tests.html

```
<div id="qunit"></div>
<div id="qunit-fixture">
  <form>
    <input name="text" />
    <div class="has-error">Error text</div>
  </form>
</div>
```

```
<script src="../../jquery-3.1.1.min.js"></script>
```

Como você provavelmente deve ter adivinhado, a jQuery reinicia o conteúdo da div de fixtures antes de cada teste, portanto isso nos leva de volta a dois testes passando corretamente:

4 assertions of 4 passed, 0 failed.

1. smoke test (2)
2. smoke test 2 (2)

Construindo um teste de unidade de JavaScript para a funcionalidade desejada

Agora que fomos apresentados às nossas ferramentas de testes de JavaScript, podemos voltar a ter somente um teste e começar a escrever o código verdadeiro:

lists/static/tests/tests.html

```
<script>
```

```
QUnit.test("errors should be hidden on keypress", function (assert) {
  $('input[name="text"]').trigger('keypress'); ❶
  assert.equal($('.has-error').is(':visible'), false);
});
```

```
</script>
```

- ❶ O método `.trigger` da jQuery é usado principalmente para testes. Ele diz para “disparar um evento de DOM JavaScript no(s) elemento(s)”. Nesse caso, estamos usando o evento de

pressionamento de tecla (keypress), disparado internamente pelo navegador sempre que um usuário digita algo em um elemento de entrada em particular.



A jQuery está ocultando uma boa dose de complexidade internamente, nesse caso. Dê uma olhada em Quirksmode.org (<http://www.quirksmode.org/dom/events/index.html>) para uma visão do terrível emaranhado que são as diferenças entre a interpretação de eventos entre navegadores distintos. O motivo de a jQuery ser tão popular é o fato de ela simplesmente fazer tudo isso desaparecer.

Eis o resultado:

```
0 assertions of 1 passed, 1 failed.  
1. errors should be hidden on keypress (1, 0, 1)  
  1. failed  
     Expected: false  
     Result: true
```

Vamos supor que queremos manter o nosso código em um arquivo JavaScript independente do chamado *list.js*.

lists/static/tests/tests.html

```
<script src="../../jquery-3.1.1.min.js"></script>  
<script src="../../list.js"></script>  
<script src="qunit-2.0.1.js"></script>  
<script>  
  [...]
```

Eis o mínimo de código para fazer aquele teste passar:

lists/static/list.js

```
$('.has-error').hide();
```

E funciona...

```
1 assertions of 1 passed, 0 failed.  
1. errors should be hidden on keypress (1)
```

Porém há um problema óbvio. É melhor acrescentar outro teste:

lists/static/tests/tests.html

```

QUnit.test("errors should be hidden on keypress", function (assert) {
    $('input[name="text"]').trigger('keypress');
    assert.equal($('.has-error').is(':visible'), false);
});
QUnit.test("errors aren't hidden if there is no keypress", function (assert) {
    assert.equal($('.has-error').is(':visible'), true);
});

```

Agora temos uma falha esperada:

```

1 assertions of 2 passed, 1 failed.
1. errors should be hidden on keypress (1)
2. errors aren't hidden if there is no keypress (1, 0, 1)
  1. failed
     Expected: true
     Result: false
[...]

```

Podemos fazer uma implementação mais realista:

lists/static/list.js

```

$('input[name="text"]').on('keypress', function () { ❶
    $('.has-error').hide();
});

```

- ❶ Essa linha afirma o seguinte: encontre qualquer elemento de entrada cujo atributo de nome seja “text” e adicione um listener de eventos que reaja a eventos de pressionamento de tecla. O listener de eventos é a função inline, que oculta todos os elementos com a classe `.has-error`.

Isso funciona? Não.

```

1 assertions of 2 passed, 1 failed.
1. errors should be hidden on keypress (1, 0, 1)
  1. failed
     Expected: false
     Result: true
[...]
2. errors aren't hidden if there is no keypress (1)

```

Maldição! Por que isso acontece?

Fixtures, ordem de execução e o estado global: principais desafios nos testes de JavaScript

Uma das dificuldades com JavaScript em geral, e com os testes em particular, está em compreender a ordem de execução de nosso código (isto é, o que acontece quando). Quando o nosso código em *list.js* executa, e quando cada um de nossos testes executa? E como isso interage com o estado global, isto é, com o DOM de nossa página web? E as fixtures que já vimos devem ser limpas após cada teste?

console.log para prints de depuração

Vamos adicionar alguns prints de depuração, ou “console.logs”:

lists/static/tests/tests.html

```
<script>

console.log('qunit tests start');

QUnit.test("errors should be hidden on keypress", function (assert) {
  console.log('in test 1');
  $('input[name="text"]').trigger('keypress');
  assert.equal($('.has-error').is(':visible'), false);
});

QUnit.test("errors aren't hidden if there is no keypress", function (assert) {
  console.log('in test 2');
  assert.equal($('.has-error').is(':visible'), true);
});
</script>
```

Vamos fazer o mesmo em nosso código JavaScript:

lists/static/list.js (ch14|015)

```
$('#input[name="text"]').on('keypress', function () {
  console.log('in keypress handler');
  $('.has-error').hide();
});
```

```
console.log('list.js loaded');
```

Execute os testes novamente, abrindo o console de depuração do navegador (geralmente com Ctrl-Shift-I), e você deverá ver algo como mostra a Figura 16.3.

O que vemos?

- *list.js* é carregado antes. Portanto, nosso listener de eventos deve estar associado ao elemento de entrada.
- Em seguida, nosso arquivo de testes da QUnit é carregado.
- Então cada teste é executado.

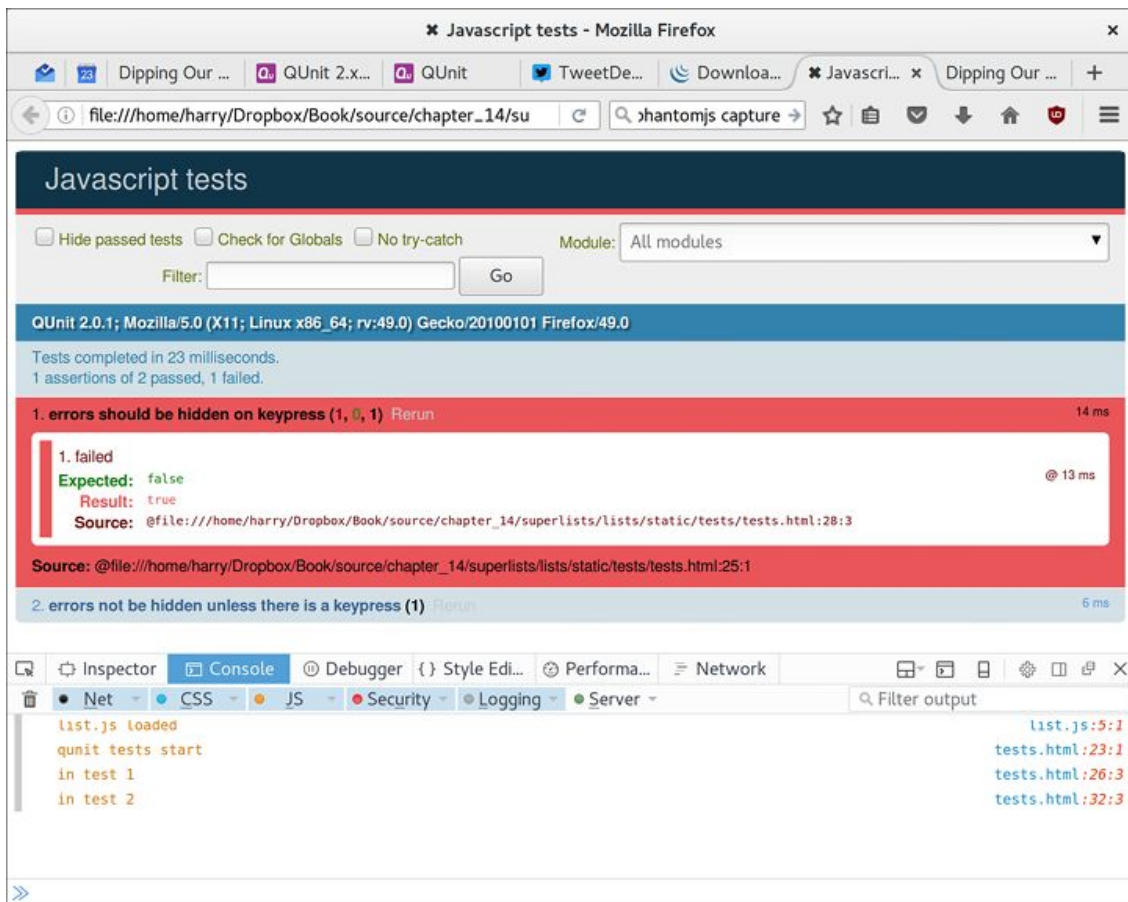


Figura 16.3 – Testes da QUnit com saídas de depuração de *console.log*.

Contudo, ao pensar no processo, cada teste “reiniciará” a div de fixtures, o que significa que o elemento de entrada será destruído e recriado. Desse modo, o elemento de entrada que *list.js* vê e ao qual

associa o listener de eventos terá sido substituído por um novo elemento quando cada teste começar.

Usando uma função de inicialização para ter mais controle sobre o tempo de execução

Precisamos ter mais controle sobre a ordem de execução de nosso JavaScript. Em vez de depender somente do fato de o código de *list.js* executar sempre que for carregado por uma tag `<script>`, podemos usar um padrão comum, que consiste em definir uma função de “inicialização”, e chamá-la sempre que quisermos em nossos testes (e depois, na vida real):

lists/static/list.js

```
var initialize = function () {
  console.log('initialize called');
  $('input[name="text"]').on('keypress', function () {
    console.log('in keypress handler');
    $('.has-error').hide();
  });
};
console.log('list.js loaded');
```

Em nosso arquivo de testes, chamamos `initialize` em cada teste:

lists/static/tests/tests.html (ch14I017)

```
QUnit.test("errors should be hidden on keypress", function (assert) {
  console.log('in test 1');
  initialize();
  $('input[name="text"]').trigger('keypress');
  assert.equal($('.has-error').is(':visible'), false);
});
```

```
QUnit.test("errors aren't hidden if there is no keypress", function (assert) {
  console.log('in test 2');
  initialize();
  assert.equal($('.has-error').is(':visible'), true);
});
```

Agora, veremos que nossos testes passam e nossa saída de depuração deverá fazer mais sentido:

- ```
2 assertions of 2 passed, 0 failed.
1. errors should be hidden on keypress (1)
2. errors aren't hidden if there is no keypress (1)
```

```
list.js loaded
qunit tests start
in test 1
initialize called
in keypress handler
in test 2
initialize called
```

Viva! Vamos remover aqueles console.logs:

## lists/static/list.js

```
var initialize = function () {
 $('input[name="text"]').on('keypress', function () {
 $('.has-error').hide();
 });
};
```

E dos testes também...

## lists/static/tests/tests.html

```
QUnit.test("errors should be hidden on keypress", function (assert) {
 initialize();
 $('input[name="text"]').trigger('keypress');
 assert.equal($('.has-error').is(':visible'), false);
});
```

```
QUnit.test("errors aren't hidden if there is no keypress", function (assert) {
 initialize();
 assert.equal($('.has-error').is(':visible'), true);
});
```

Vamos à hora da verdade: traremos a jQuery, o nosso script e chamaremos a nossa função de inicialização em nossas páginas reais:

## lists/templates/base.html (ch14I020)

```
</div>
<script src="/static/jquery-3.1.1.min.js"></script>
<script src="/static/list.js"></script>

<script>
 initialize();
</script>
</body>
</html>
```



Colocar as cargas de script no final do corpo de seu HTML é uma boa prática, pois significa que o usuário não terá de esperar todo o seu JavaScript ser carregado para ver algo na página. Também ajuda a garantir que a maior parte do DOM foi carregada antes que qualquer script seja executado.

E então executamos o nosso FT:

```
$ python manage.py test functional_tests.test_list_item_validation.\
ItemValidationTest.test_error_messages_are_cleared_on_input
[...]
```

Ran 1 test in 3.023s

OK

Viva! Merece um commit!

```
$ git add lists/static
$ git commit -m"add jquery, qunit tests, list.js with keypress listeners"
```

## Boilerplate para onload e namespacing

*Como diria o detetive Columbo: Ah, e tem mais um detalhe<sup>3</sup>.* O nome de nossa função `initialize` é genérico demais: e se incluíssemos alguma ferramenta JavaScript de terceiros mais tarde, que também definisse uma função chamada `initialize`? Vamos nos dar um “namespace” que seja improvável de ser usado por outras pessoas:

## lists/static/list.js

```
window.Superlists = {}; ❶
window.Superlists.initialize = function () { ❷
 $('input[name="text"]').on('keypress', function () {
 $('.has-error').hide();
 });
};
```

- ❶ Declaramos explicitamente um objeto como uma propriedade do global “window”, dando-lhe um nome que achamos que ninguém mais provavelmente usará.
- ❷ Então fazemos de nossa função initialize um atributo do objeto desse namespace.



Há várias outras maneiras muito mais inteligentes de lidar com namespaces em JavaScript, mas todas são mais complicadas, e não sou expert o suficiente para orientar você a usá-las. Se quiser aprender mais, procure *require.js*, que parecia a solução definitiva, ou no mínimo o era no último fentossegundo de JavaScript.

## lists/static/tests/tests.html

```
<script>
QUnit.test("errors should be hidden on keypress", function (assert) {
 window.Superlists.initialize();
 $('input[name="text"]').trigger('keypress');
 assert.equal($('.has-error').is(':visible'), false);
});

QUnit.test("errors aren't hidden if there is no keypress", function (assert) {
 window.Superlists.initialize();
 assert.equal($('.has-error').is(':visible'), true);
});
</script>
```

Por fim, sempre que você tiver um pouco de JavaScript que interaja com o DOM, é sempre bom encapsulá-lo em um código boilerplate “onload” para garantir que a página tenha sido totalmente carregada antes que ela tente fazer algo. Atualmente isso funciona de qualquer

modo, pois colocamos a tag `<script>` bem no final da página, mas não devemos depender disso.

O boilerplate de `onload` na jQuery é mínimo:

## lists/templates/base.html

```
<script>

$(document).ready(function () {
 window.Superlists.initialize();
});

</script>
```

Leia mais na documentação de `.ready()` da jQuery (<http://api.jquery.com/ready/>).

## Testes de JavaScript no ciclo de TDD

Talvez você esteja se perguntando como esses testes de JavaScript se enquadram em nosso ciclo de TDD com “laço duplo”. A resposta é que eles desempenham exatamente o mesmo papel que nossos testes de unidade de Python.

1. Escreva um FT e veja que ele falha.
2. A seguir, descubra o tipo de código de que você precisa: Python ou JavaScript?
3. Escreva um teste de unidade em uma das linguagens e veja que ele falha.
4. Escreva um pouco de código em uma das linguagens e faça o teste passar.
5. Enxágue e repita.



Você quer ter um pouco mais de prática com JavaScript? Veja se consegue fazer com que nossas mensagens de erro sejam ocultas quando o usuário clicar no elemento de entrada, assim como simplesmente ocorre quando eles digitam nesse local. Você deverá executar um FT para isso também.

Estamos quase prontos para passar para a Parte III. O último passo é implantar o nosso novo código em nossos servidores. Não se esqueça de fazer um último commit incluindo *base.html* antes!

## Alguns tópicos que não puderam ser abordados

Neste capítulo, eu quis discutir o básico sobre testes de JavaScript e como eles se enquadram em nosso fluxo de trabalho com TDD. Eis algumas referências para outras pesquisas:

- No momento, nosso teste apenas verifica se o JavaScript funciona em uma página. É apropriado porque estamos incluindo-o em *base.html*, mas, se o tivéssemos adicionado somente em *home.html*, os testes continuariam passando. É uma questão a ser avaliada, mas você poderia optar por escrever um teste extra nesse caso.
- Ao escrever JavaScript, obtenha o máximo de ajuda que puder de seu editor para evitar “armadilhas” comuns. Dê uma olhada em ferramentas de verificação de sintaxe/erros como “jshint” e “jshint”, também conhecidos como “linters”.
- A QUnit espera principalmente que você “execute” seus testes usando um navegador web real. Isso tem a vantagem de facilitar a criação de algumas fixtures HTML que correspondam ao tipo de HTML que seu site realmente contém, no qual os testes serão executados. Entretanto também é possível executar testes de JavaScript a partir da linha de comando. Veremos um exemplo no Capítulo 24.
- A grande novidade no mundo do desenvolvimento de frontend são os frontends MVC, como *angular.js* e React. A maioria dos tutoriais



para eles utiliza uma biblioteca de asserção do tipo RSpec chamada Jasmine (<https://jasmine.github.io/>). Se você for usar um deles, provavelmente perceberá que a vida será mais fácil se utilizar o Jasmine em vez da QUnit.

Este livro também contém mais diversão com JavaScript! Dê uma olhada no apêndice sobre a API Rest quando estiver pronto.

## Observações sobre testes de JavaScript

- Uma das principais vantagens do Selenium é que ele permite testar se seu JavaScript de fato funciona, exatamente como o seu código Python é testado.
- Há muitas bibliotecas para execução de testes de JavaScript por aí. A QUnit está intimamente relacionada com a jQuery, e esse é o motivo principal pelo qual eu a escolhi.
- Independentemente da biblioteca de testes usada, você sempre precisará encontrar soluções para o desafio principal para os testes de JavaScript, que diz respeito ao *gerenciamento do estado global*. Isso inclui:
  - o DOM / as fixtures de HTML;
  - namespacing;
  - compreender e controlar a ordem de execução.
- Realmente não estou sendo sincero quando digo que JavaScript é terrível. Na verdade, ele pode ser bem divertido. Contudo vou repetir: não se esqueça de ler o livro *JavaScript: The Good Parts*.

---

<sup>1</sup> Devo admitir que, quando começamos a procurar ferramentas BDD para Python, a situação se torna um pouco mais confusa.

<sup>2</sup> Somente porque ela inclui o executor de testes NyanCat (<https://mochajs.org/#nyan>).

<sup>3</sup> N.T.: O detetive Columbo é a personagem principal de uma série policial de TV dos anos 1970, estrelada pelo ator Peter Falk.



# CAPÍTULO 17

## Implantando o nosso novo código

É hora de fazer a implantação de nosso novo código brilhante de validação em nossos servidores live (ao vivo). Será uma chance de ver nossos scripts automatizados de implantação em ação pela segunda vez.



Neste momento, gostaria de expressar um enorme agradecimento a Andrew Godwin e a toda a equipe do Django. Até o Django 1.7, eu costumava ter uma longa seção completa, totalmente dedicada a migrações. Atualmente, as migrações “simplesmente funcionam”, portanto pude descartá-la inteiramente. Obrigado pelo excelente trabalho, pessoal!

### Implantação no servidor de staging

Começaremos pelo servidor de staging:

```
$ git push
$ cd deploy_tools
$ fab deploy:host=elspeth@superlists-staging.ottg.eu
Disconnecting from superlists-staging.ottg.eu... done.
```

Reinicie o Gunicorn:

```
elspeth@server:$ sudo systemctl restart gunicorn-superlists-
staging.ottg.eu
```

Então, execute os testes no staging:

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test
functional_tests
OK
```

## Implantação no servidor live

Supondo que tudo esteja bem, vamos então executar nossa implantação no servidor live:

```
$ fab deploy:host=elspeth@superlists.ottg.eu
elspeth@server:$ sudo service unicorn-superlists.ottg.eu restart
```

## O que fazer se você vir um erro de banco de dados

Como nossas migrações introduzem uma nova restrição de integridade, você poderá descobrir que ela não é aplicada porque algum dado existente viola essa restrição.

Nesse ponto, há duas opções:

- Apagar o banco de dados no servidor e tentar novamente. Afinal de contas, é apenas um projeto lúdico!
- Saber mais sobre migrações de dados. Veja o Apêndice D.

## Conclusão: atribua uma tag no git para a nova versão

A última tarefa a ser feita é atribuir uma tag para a versão em nosso VCS – é importante que possamos sempre manter o controle do que está no servidor live:

```
$ git tag -f LIVE # precisa do -f porque estamos substituindo a tag antiga
$ export TAG=date +DEPLOYED-%F/%H%M
$ git tag $TAG
$ git push -f origin LIVE $TAG
```



Algumas pessoas não gostam de usar push -f e atualizar uma tag existente; em vez disso, elas usarão algum tipo de número de versão para atribuir uma tag às suas versões. Utilize a opção mais adequada para você.

Com essa nota, podemos concluir a Parte II e passar para os tópicos mais empolgantes que compõem a Parte III. Mal posso

esperar!

## **Revisão do procedimento de implantação**

Fizemos duas implantações até agora, portanto é uma boa hora para uma pequena revisão:

- execute um git push do código mais recente;
- faça a implantação no servidor de staging e execute aí os testes funcionais;
- faça a implantação no servidor live;
- atribua uma tag à versão.

Os procedimentos de implantação evoluem e se tornam mais completos à medida que o projeto cresce, e é uma área que pode se tornar difícil de manter, cheia de verificações manuais e procedimentos caso você não tenha cuidado em manter tudo automatizado. Há muito mais a dizer sobre esse assunto, mas está fora do escopo deste livro. Não se esqueça de dar uma olhada no Apêndice C e faça leituras extras sobre o assunto “implantação contínua” (continuous deployment).



## PARTE III

# Tópicos mais avançados sobre testes

“Oh, céus, como assim? Outra seção? Harry, estou exausto, já se foram trezentas páginas, acho que não aguento mais uma nova seção completa do livro. Principalmente se ela se chama ‘Avançados’... talvez eu possa me ver livre dela simplesmente a ignorando?”

Ah, não, não pode! Esta seção pode se chamar avançada, mas está repleta de tópicos realmente importantes para o TDD e o desenvolvimento web. Sem chances de você ignorá-la. Podemos dizer que ela é *até mesmo mais importante* que as duas primeiras seções.

Falaremos sobre como integrar sistemas de terceiros e testá-los. O desenvolvimento web moderno tem tudo a ver com a reutilização de componentes existentes. Abordaremos a simulação (mocking) e o isolamento de testes, uma parte realmente essencial do TDD, além de uma técnica de que você precisará inclusive para as bases de código mais simples. Discutiremos a depuração do lado do servidor e as fixtures de testes, e como configurar um ambiente de Integração Contínua (Continuous Integration). Nada disso é um recurso extra opcional de luxo, do tipo pegar ou largar, para o seu projeto – são todos vitais!

Inevitavelmente a curva de aprendizado se tornará um pouco mais inclinada nesta seção. Você poderá se ver tendo que ler as informações umas duas vezes até elas serem absorvidas, ou poderá perceber que os códigos não funcionarão na primeira tentativa e que

será preciso fazer um pouco de depuração por conta própria. Todavia, seja persistente! Quanto mais difícil, maior será a recompensa. Estarei sempre feliz em ajudar caso você não saiba o que fazer; basta me enviar um email em *obeythetestinggoat@gmail.com*.

Acredite em mim, prometo que o melhor ainda está por vir!





## CAPÍTULO 18

# Autenticação de usuário, spiking e de-spiking

Nosso lindo site de listas está ativo há alguns dias e nossos usuários estão começando a nos dar feedbacks. “Amamos o site” – dizem eles – “mas perdemos nossas listas o tempo todo. Lembrar os URLs manualmente é difícil. Seria ótimo se o site pudesse lembrar com quais listas começamos.”

Lembre-se de Henry Ford e os cavalos mais rápidos. Sempre que ouvir um requisito de usuário, é importante ir um pouco mais além e pensar: qual é o verdadeiro requisito nesse caso? E como posso fazer com que ele envolva uma nova tecnologia interessante que venho tentando experimentar?

Claramente o requisito, nesse caso, é que as pessoas querem ter algum tipo de conta de usuário no site. Assim, sem demora, vamos mergulhar de cabeça na autenticação.

Naturalmente não vamos ficar lidando com lembrar senhas – além de ser *tão* anos 1990, armazenar senhas de usuário de forma protegida é um pesadelo de segurança que preferimos deixar a cargo de outras pessoas. Usaremos algo divertido chamado autenticação sem senha (passwordless auth) como alternativa.

(Se você *insiste* em armazenar as próprias senhas, o módulo de autenticação default do Django está pronto e à sua espera. É simples e interessante, mas deixarei que você o descubra por conta própria.)

## Autenticação sem senha

Qual sistema de autenticação poderíamos usar para evitar que precisemos armazenar as senhas por conta própria? Oauth? Openid? “Login com Facebook”? Argh. Para mim, todos eles têm conotações assustadoras inaceitáveis; por que o Google ou o Facebook deveriam saber em qual site você está fazendo login, e quando?

Na primeira edição, utilizei um projeto experimental chamado “Persona”, preparado por alguns dos maravilhosos idealistas tecno-hippies do Mozilla, mas infelizmente esse projeto foi abandonado.

Como alternativa, encontrei uma abordagem divertida para a autenticação que recebe o nome de “Sem senha” (Passwordless), mas você pode chamá-la de “basta usar o email”.

O sistema foi inventado por alguém irritado com o fato de ter que criar novas senhas para tantos sites, que se viu simplesmente usando senhas aleatórias e descartáveis, nem mesmo tentando lembrar-se delas e usando o recurso de “esqueci minha senha” sempre que precisasse fazer login novamente. Você pode ler tudo sobre esse assunto no Medium (<https://medium.com/@ninjudd/passwords-are-obsolete-9ed56d483eb#.cx8iber30>).

Eis o conceito: basta usar o email para conferir a identidade de alguém. Se você vai ter um recurso do tipo “esqueci minha senha”, então estará confiando no email, de qualquer modo, então por que não extrapolar? Sempre que alguém quiser fazer login, geramos um URL único para as pessoas usarem, enviamos esse URL por email e elas então clicarão nele para acessar o site.

Não é, de forma alguma, um sistema perfeito; com efeito, há muitas sutilezas nas quais devemos pensar antes que seja realmente uma boa solução de login para um site de produção, mas este é apenas um projeto lúdico, portanto vamos lhe dar uma chance.

## Programação exploratória, também conhecida como “spiking”

Antes de escrever este capítulo, tudo que eu sabia sobre a autenticação sem senha era a descrição que eu havia lido no artigo já mencionado. Não tinha visto nenhum código para ela antes, e não sabia realmente por onde começar a sua implementação.

Nos Capítulos 13 e 14, vimos que podemos usar um teste de unidade como forma de explorar uma nova API ou ferramenta, mas, às vezes, queremos simplesmente compor um código sem fazer nenhum teste, apenas para ver se ele funciona, conhecê-lo melhor ou ter uma noção do que ele é. Certamente não há problemas nisso. Ao conhecer uma nova ferramenta ou explorar uma nova solução possível, geralmente é apropriado deixar de lado o rigoroso processo de TDD e construir um pequeno protótipo sem testes, ou, quem sabe, com pouquíssimos testes. O Testing Goat não se importará em desviar o olhar por um instante.

Esse tipo de atividade de prototipagem geralmente é conhecido como “spike” por motivos bem conhecidos (<http://stackoverflow.com/questions/249969/why-are-tdd-spikes-called-spikes>).

Minha primeira atitude foi observar os pacotes já existentes de autenticação de Python e Django, como o `django-allauth` (<http://www.intenct.nl/projects/django-allauth/>) e o `python-social-auth` (<https://github.com/omab/python-social-auth>), porém ambos pareciam demasiadamente complicados para esta etapa (além do mais, será mais divertido fazer a nossa própria implementação!).

Então, mergulhei de cabeça e fiz alguns hackings, e, após alguns becos sem saída e viradas para o lado errado, eu tinha algo que praticamente funcionava. Conduzirei você por um tour e então prosseguiremos fazendo um “de-spiking” da implementação – isto é, substituiremos o protótipo por um código testado, pronto para produção.

Você deve seguir em frente e adicionar esse código em seu próprio site também; então poderá brincar com ele, tentar fazer login com seu próprio endereço de email e se convencer de que ele realmente funciona.

## Iniciando um branch para o spike

Antes de embarcar em um spike, é uma boa ideia iniciar um novo branch, de modo que você continue usando o seu VCS sem se preocupar com o fato de os commits de seu spike acabarem se misturando com seu código de produção:

```
$ git checkout -b passwordless-spike
```

Vamos manter o controle de alguns pontos que esperamos conhecer com o spike:

- Como enviar emails
- Gerar e reconhecer tokens únicos
- Como autenticar alguém no Django
- Quais passos o usuário terá que executar?

## UI de login no frontend

Vamos começar pelo frontend, compondo um formulário para que possamos inserir o seu endereço de email na barra de navegação (navbar) e com um link de logout para os usuários que já tenham se autenticado:

### lists/templates/base.html (ch16l001)

```
<body>
 <div class="container">

 <div class="navbar">
 {% if user.is_authenticated %}
 <p>Logged in as {{ user.email}}</p>
 <p>Log out</p>
 {% else %}
 </div>
 </div>
```

```

 <form method="POST" action="{% url 'send_login_email' %}">
 Enter email to log in: <input name="email" type="text" />
 {% csrf_token %}
 </form>
 {% endif %}
</div>

<div class="row">
[...]
```

## Enviando emails a partir do Django

A teoria de login será algo mais ou menos assim:

- Quando alguém quiser fazer login, geraremos um token secreto único para essa pessoa, armazenaremos esse dado no banco de dados associando-o ao email e enviaremos para ela.
- As pessoas então verificarão o email, que conterà um link com um URL incluindo esse token.
- Quando clicarem nesse link, verificaremos se o token está presente no banco de dados; em caso afirmativo, elas serão logadas como o usuário associado.

Inicialmente prepararemos uma aplicação para as tarefas associadas a contas:

```
$ python manage.py startapp accounts
```

Configuraremos *urls.py* com pelo menos um URL. No *superlists/urls.py* de nível mais alto...

### superlists/urls.py (ch16l003)

```

from django.conf.urls import include, url
from lists import views as list_views
from lists import urls as list_urls
from accounts import urls as accounts_urls
```

```

urlpatterns = [
 url(r'^$', list_views.home_page, name='home'),
 url(r'^lists/', include(list_urls)),
```

```
 url(r'^accounts/', include(accounts_urls)),
]
```

E no *urls.py* do módulo de contas:

## accounts/urls.py (ch16l004)

```
from django.conf.urls import url
from accounts import views

urlpatterns = [
 url(r'^send_email$', views.send_login_email, name='send_login_email'),
]
```

Eis a view responsável por criar um token associado ao endereço de email que o usuário inserir em nosso formulário de login:

## accounts/views.py (ch16l005)

```
import uuid
import sys
from django.shortcuts import render
from django.core.mail import send_mail

from accounts.models import Token

def send_login_email(request):
 email = request.POST['email']
 uid = str(uuid.uuid4())
 Token.objects.create(email=email, uid=uid)
 print('saving uid', uid, 'for email', email, file=sys.stderr)
 url = request.build_absolute_uri(f'/accounts/login?uid={uid}')
 send_mail(
 'Your login link for Superlists',
 f'Use this link to log in:\n\n{url}',
 'noreply@superlists',
 [email],
)
 return render(request, 'login_email_sent.html')
```

Para que esse código funcione, precisaremos de uma mensagem placeholder confirmando que o email foi enviado:

## accounts/templates/login\_email\_sent.html (ch16l006)

```
<html>
<h1>Email sent</h1>

<p>Check your email, you'll find a message with a link that will log you into
the site.</p>

</html>
```

Você pode ver como esse código é improvisado – iríamos integrar esse template ao nosso *base.html* na versão real.

Mais importante ainda, para que a função `send_mail` do Django funcione, precisamos lhe informar o endereço de nosso servidor de emails. Estou simplesmente usando minha conta do Gmail<sup>1</sup> por enquanto. Você pode usar qualquer provedor de emails que quiser, desde que ele aceite SMTP:

## superlists/settings.py (ch16l007)

```
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_HOST_USER = 'obeythetestinggoat@gmail.com'
EMAIL_HOST_PASSWORD = os.environ.get('EMAIL_PASSWORD')
EMAIL_PORT = 587
EMAIL_USE_TLS = True
```



Se você quiser usar o Gmail também, provavelmente terá que acessar a página de configurações de segurança de sua conta do Google. Se você estiver usando uma autenticação de dois fatores, vai querer configurar uma senha específica para a aplicação (<https://myaccount.google.com/apppasswords>). Se não estiver, é provável que ainda vá precisar permitir o acesso para aplicações menos seguras (<https://www.google.com/settings/security/lesssecureapps>). Talvez você queira considerar a criação de uma nova conta do Google para essa finalidade, em vez de usar uma conta que contenha dados confidenciais.

## Usando variáveis de ambiente para evitar segredos no código-fonte

Cedo ou tarde todo projeto precisa descobrir uma maneira de lidar



com “segredos”: informações como senhas de email ou chaves de API, que você não deseja compartilhar com todo o restante do mundo. Se o seu repositório for privado, talvez não haja problemas em salvar esses dados no Git, mas geralmente não é esse o caso. Há também uma intersecção com a necessidade de ter diferentes configurações para os ambientes de desenvolvimento e de produção. (Você se lembra de como lidamos com a configuração de SECRET\_KEY do Django no Capítulo 11?)

Um padrão comum (<https://12factor.net/config>) é usar variáveis de ambiente para esse tipo de definição de configuração, que é o que estou fazendo com os.environ.get.

Para que isso funcione, preciso definir a variável de ambiente no shell que está executando o meu servidor de desenvolvimento:

```
$ export EMAIL_PASSWORD="sekrit"
```

Mais tarde, veremos como adicionar isso no servidor de staging também.

## Armazenando tokens no banco de dados

Como estamos nos saindo?

- ~~Como enviar emails~~
- Gerar e reconhecer tokens únicos
- Como autenticar alguém no Django
- Quais passos o usuário terá que executar?

Precisaremos de um modelo para armazenar nossos tokens no banco de dados – eles associam um endereço de email a um ID único. É bem simples:

### accounts/models.py (ch16l008)

```
from django.db import models
```

```
class Token(models.Model):
 email = models.EmailField()
```

```
uid = models.CharField(max_length=255)
```

## Modelos personalizados para autenticação

Enquanto estamos lidando com modelos, vamos começar a fazer experimentos com a autenticação no Django.

- ~~Como enviar emails~~
- ~~Gerar e reconhecer tokens únicos~~
- Como autenticar alguém no Django
- Quais passos o usuário terá que executar?

O primeiro item de que precisaremos é um modelo de usuário. Quando escrevi este livro pela primeira vez, os modelos de usuário personalizados eram novidade no Django, portanto me aprofundei na documentação do Django sobre autenticação (<https://docs.djangoproject.com/en/1.11/topics/auth/customizing/>) e tentei compor o modelo mais simples possível:

### accounts/models.py (ch16l009)

```
[...]
from django.contrib.auth.models import (
 AbstractBaseUser, BaseUserManager, PermissionsMixin
)
```

```
class ListUser(AbstractBaseUser, PermissionsMixin):
 email = models.EmailField(primary_key=True)
 USERNAME_FIELD = 'email'
 #REQUIRED_FIELDS = ['email', 'height']

 objects = ListUserManager()

 @property
 def is_staff(self):
 return self.email == 'harry.percival@example.com'

 @property
```

```
def is_active(self):
 return True
```

Isso é o que eu chamo de um modelo de usuário mínimo! Um campo, nada desses dados de primeiro nome/sobrenome/nome de usuário sem nexos e, nitidamente, sem senha! Esse problema é de outras pessoas!

Mais uma vez, porém, você pode ver que esse código não está pronto para produção, por causa das linhas comentadas ou o endereço de email de Harry, fixo no código. Arrumaremos tudo isso quando fizermos o de-spiking.

Para que esse código funcione, precisamos de um gerenciador de modelo (model manager) para o usuário:

## accounts/models.py (ch16|010)

```
[...]
class ListUserManager(BaseUserManager):

 def create_user(self, email):
 ListUser.objects.create(email=email)

 def create_superuser(self, email, password):
 self.create_user(email)
```

Não é necessário se preocupar com o que é um gerenciador de modelo nesta etapa; por enquanto, só precisamos dele porque é assim, e ele simplesmente funciona. Quando fizermos o de-spiking, analisaremos cada porção de código que de fato acabará no ambiente de produção e garantiremos que tudo será entendido por completo.

## **Finalizando a autenticação personalizada do Django**

Estamos quase lá – nosso último passo combina reconhecimento do token e então o login propriamente dito do usuário. Feito isso, poderemos, basicamente, riscar todos os itens de nossa folha de

rascunho:

- ~~Como enviar emails~~
- ~~Gerar~~ e reconhecer tokens únicos
- Como autenticar alguém no Django
- Quais passos o usuário terá que executar?

Eis a view que realmente lida com o clique no link que está no email:

## accounts/views.py (ch16|011)

```
import uuid
import sys
from django.contrib.auth import authenticate
from django.contrib.auth import login as auth_login
from django.core.mail import send_mail
from django.shortcuts import redirect, render
[...]
```

```
def login(request):
 print('login view', file=sys.stderr)
 uid = request.GET.get('uid')
 user = authenticate(uid=uid)
 if user is not None:
 auth_login(request, user)
 return redirect('/')
```

A função “authenticate” chama o framework de autenticação do Django, que configuramos usando um “backend personalizado de autenticação”, cuja tarefa é validar o UID e devolver um usuário com o email correto.

Poderíamos ter executado essas tarefas diretamente na view, mas podemos também estruturar tudo do modo como o Django espera que seja feito. O resultado é uma separação razoavelmente clara das responsabilidades:

## accounts/authentication.py (ch16|012)

```
import sys
from accounts.models import ListUser, Token
```

```

class PasswordlessAuthenticationBackend(object):

 def authenticate(self, uid):
 print('uid', uid, file=sys.stderr)
 if not Token.objects.filter(uid=uid).exists():
 print('no token found', file=sys.stderr)
 return None
 token = Token.objects.get(uid=uid)
 print('got token', file=sys.stderr)
 try:
 user = ListUser.objects.get(email=token.email)
 print('got user', file=sys.stderr)
 return user
 except ListUser.DoesNotExist:
 print('new user', file=sys.stderr)
 return ListUser.objects.create(email=token.email)

```

```

def get_user(self, email):
 return ListUser.objects.get(email=email)

```

Novamente, temos muitos prints para depuração nesse código, e um pouco de código duplicado – não é algo que vamos querer ter no ambiente de produção, mas funciona...

Por fim, eis a view de logout:

## accounts/views.py (ch16l013)

```

from django.contrib.auth import login as auth_login, logout as auth_logout
[...]

```

```

def logout(request):
 auth_logout(request)
 return redirect('/')

```

Adicione o login e o logout em nosso *urls.py*...

## accounts/urls.py (ch16l014)

```

from django.conf.urls import url
from accounts import views

```

```
urlpatterns = [
 url(r'^send_email$', views.send_login_email, name='send_login_email'),
 url(r'^login$', views.login, name='login'),
 url(r'^logout$', views.logout, name='logout'),
]
```

Estamos quase lá! Vamos ativar o nosso backend de autenticação e nossa nova aplicação de contas em *settings.py*:

## superlists/settings.py (ch16l015)

```
INSTALLED_APPS = [
 #'django.contrib.admin',
 'django.contrib.auth',
 'django.contrib.contenttypes',
 'django.contrib.sessions',
 'django.contrib.messages',
 'django.contrib.staticfiles',
 'lists',
 'accounts',
]

AUTH_USER_MODEL = 'accounts.ListUser'
AUTHENTICATION_BACKENDS = [
 'accounts.authentication.PasswordlessAuthenticationBackend',
]

MIDDLEWARE = [
 [...]
```

Um `makemigrations` rápido para deixar os modelos de token e de usuário reais:

```
$ python manage.py makemigrations
Migrations for 'accounts':
 accounts/migrations/0001_initial.py
 - Create model ListUser
 - Create model Token
```

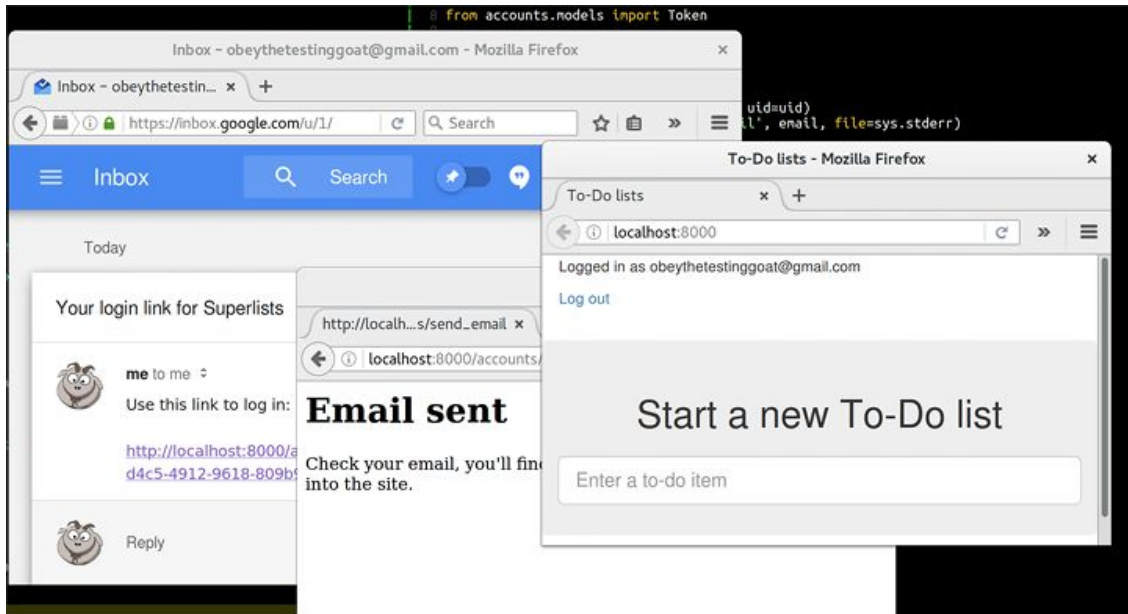
E um `migrate` para construir o banco de dados:

```
$ python manage.py migrate
[...]
```

Running migrations:

Applying accounts.0001\_initial... OK

Acho que acabamos! Por que não disparar um servidor de desenvolvimento com runserver e ver a aparência de tudo (Figura 18.1)?



*Figura 18.1 – Funciona! Funciona! Hahahahaha.*



Se você vir uma mensagem de erro SMTPSenderRefused, não se esqueça de definir a variável de ambiente EMAIL\_PASSWORD no shell que está executando runserver.

É basicamente isso! Durante o processo, tive que lutar arduamente: dei cliques pela UI de segurança da conta do Gmail por um tempo, deparei com vários atributos faltando em meu modelo de usuário personalizado (porque eu não havia lido devidamente a documentação) e, em certo momento, tive até mesmo que passar para a versão de desenvolvimento do Django para resolver um bug, que, felizmente, acabou se mostrando irrelevante.

## **Informação adicional: logging em stderr**

Durante um spiking, é essencial ser capaz de ver as exceções sendo geradas

pelo seu código. De modo irritante, o Django, por padrão, não envia todas as exceções para o terminal, mas você pode fazer com que ele realize isso usando uma variável chamada LOGGING em *settings.py*:

## superlists/settings.py (ch16l017)

```
LOGGING = {
 'version': 1,
 'disable_existing_loggers': False,
 'handlers': {
 'console': {
 'level': 'DEBUG',
 'class': 'logging.StreamHandler',
 },
 },
 'loggers': {
 'django': {
 'handlers': ['console'],
 },
 },
 'root': {'level': 'INFO'},
}
```

O Django utiliza o pacote de logging com aspecto de “enterprise” da biblioteca-padrão de Python que, embora esteja repleto de recursos, exige uma curva de aprendizado bastante inclinada. Ele será discutido com um pouco mais de detalhes no Capítulo 21, e está descrito na documentação do Django (<https://docs.djangoproject.com/en/1.11/topics/logging/>).

No entanto, agora temos uma solução que funciona! Vamos fazer commit em nosso branch de spike:

```
$ git status
$ git add accounts
$ git commit -am "spiked in custom passwordless auth backend"
```

É hora de fazer um de-spiking!

## De-spiking

Fazer um de-spiking significa reescrever o código de seu protótipo usando TDD. Temos agora informações suficientes para “fazer isso



de forma apropriada”. Então, qual é o primeiro passo? Um FT, é claro!

Permaneceremos no branch de spike, por enquanto, para ver o nosso FT passar em nosso código de spiking. Então, retornaremos ao branch master e faremos commit somente do FT.

Eis uma primeira versão simples do FT:

## functional\_tests/test\_login.py

```
from django.core import mail
from selenium.webdriver.common.keys import Keys
import re

from .base import FunctionalTest

TEST_EMAIL = 'edith@example.com'
SUBJECT = 'Your login link for Superlists'

class LoginTest(FunctionalTest):

 def test_can_get_email_link_to_log_in(self):
 # Edith acessa o incrível site de superlistas
 # e, pela primeira vez, percebe que há uma seção de "Log in" na barra
 # de navegação. Essa seção está lhe dizendo para inserir o seu endereço
 # de email, portanto ela faz isso
 self.browser.get(self.live_server_url)
 self.browser.find_element_by_name('email').send_keys(TEST_EMAIL)
 self.browser.find_element_by_name('email').send_keys(Keys.ENTER)

 # Uma mensagem aparece informando-lhe que um email foi enviado
 self.wait_for(lambda: self.assertIn(
 'Check your email',
 self.browser.find_element_by_tag_name('body').text
))

 # Ela verifica seu email e encontra uma mensagem
 email = mail.outbox[0]
 self.assertIn(TEST_EMAIL, email.to)
 self.assertEqual(email.subject, SUBJECT)
```

```

A mensagem contém um link com um url
self.assertIn('Use this link to log in', email.body)
url_search = re.search(r'http://.+?$', email.body)
if not url_search:
 self.fail(f'Could not find url in email body:\n{email.body}')
url = url_search.group(0)
self.assertIn(self.live_server_url, url)

Ela clica no url
self.browser.get(url)

Ela está logada!
self.wait_for(
 lambda: self.browser.find_element_by_link_text('Log out')
)
navbar = self.browser.find_element_by_css_selector('.navbar')
self.assertIn(EMAIL, navbar.text)

```

- ❶ Você estava preocupado em como iríamos lidar com a obtenção de emails em nossos testes? Felizmente podemos trapacear, por enquanto! Ao executar os testes, o Django nos dá acesso a qualquer email que o servidor tente enviar por meio do atributo `mail.outbox`. Deixaremos a verificação de emails “de verdade” para depois (mas faremos isso!).

Se executarmos o FT, ele passará!

```
$ python manage.py test functional_tests.test_login
```

```
[...]
```

```
Not Found: /favicon.ico
```

```
saving uid [...]
```

```
login view
```

```
uid [...]
```

```
got token
```

```
new user
```

```
.
```

```

Ran 1 test in 3.729s
```

```
OK
```

Você pode até mesmo ver parte da saída de depuração que eu deixei nas implementações de view da minha versão spiking. Agora é hora de reverter todas as nossas mudanças temporárias e reintroduzi-las, uma a uma, de modo orientado a testes.

## Revertendo o nosso código de spiking

```
$ git checkout master # volta para o branch master
$ rm -rf accounts # remove qualquer traço de código de spiking
$ git add functional_tests/test_login.py
$ git commit -m "FT for login via email"
```

Vamos executar novamente o FT e deixar que ele oriente o nosso desenvolvimento:

```
$ python manage.py test functional_tests.test_login
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate
element: [name="email"]
[...]
```

A primeira atitude que ele quer que tomemos é adicionar um elemento de entrada para o email. O Bootstrap tem algumas classes embutidas (built-in) para barras de navegação, portanto nós as usaremos, e incluiremos um formulário para o email de login:

### lists/templates/base.html (ch16l020)

```
<div class="container">

 <nav class="navbar navbar-default" role="navigation">
 <div class="container-fluid">
 Superlists
 <form class="navbar-form navbar-right" method="POST" action="#">
 Enter email to log in:
 <input class="form-control" name="email" type="text" />
 {% csrf_token %}
 </form>
 </div>
 </nav>

 <div class="row">
```

[...]

Agora o nosso FT falha porque o formulário de login não faz realmente nada:

```
$ python manage.py test functional_tests.test_login
```

[...]

```
AssertionError: 'Check your email' not found in 'Superlists\nEnter email to log
in:\nStart a new To-Do list'
```



Recomendo reintroduzir a configuração anterior de LOGGING nesse ponto. Não há necessidade de um teste explícito para ela; na eventualidade improvável de ela causar uma falha em algo, nossa suíte de testes atual nos permitirá saber. Como veremos no Capítulo 21, essa configuração será útil para depuração mais tarde.

É hora de começar a escrever um pouco de código Django. Começaremos criando uma aplicação chamada `accounts` para armazenar todos os arquivos relacionados ao login:

```
$ python manage.py startapp accounts
```

Você poderia até mesmo fazer um commit somente para isso, de modo a ser capaz de distinguir os arquivos de placeholder da aplicação de nossas modificações.

Em seguida, vamos reconstruir o nosso modelo mínimo de usuário, desta vez com testes, e ver se ele resulta em um código mais organizado em comparação com o código de spiking.

## Um modelo de usuário personalizado mínimo

O modelo de usuário embutido do Django faz todo tipo de pressuposições sobre as informações que você quer monitorar sobre os usuários, desde registrar explicitamente o primeiro nome e o sobrenome<sup>2</sup> até forçar você a utilizar um nome de usuário. Sou totalmente a favor de não armazenar informações sobre usuários, a menos que seja absolutamente necessário, portanto um modelo de usuários que registre um endereço de email e nada mais parece bom para mim!

A essa altura, estou certo de que você é capaz de lidar com a criação da pasta de testes e seu `__init__.py`, remover `tests.py` e então adicionar um `test_models.py` para que contenha o seguinte:

## accounts/tests/test\_models.py (ch16l024)

```
from django.test import TestCase
from django.contrib.auth import get_user_model
```

```
User = get_user_model()
```

```
class UserModelTest(TestCase):
```

```
 def test_user_is_valid_with_email_only(self):
 user = User(email='a@b.com')
 user.full_clean() # should not raise
```

Isso resulta em uma falha esperada:

```
django.core.exceptions.ValidationError: {'password': ['This field cannot be blank.'], 'username': ['This field cannot be blank.']}
```

Senha? Nome de usuário? Bah! E que tal isto?

## accounts/models.py

```
from django.db import models
```

```
class User(models.Model):
 email = models.EmailField()
```

E fazemos a associação em `settings.py`, adicionando `accounts` em `INSTALLED_APPS`, além de uma variável chamada `AUTH_USER_MODEL`:

## superlists/settings.py (ch16l026)

```
INSTALLED_APPS = [
 #'django.contrib.admin',
 'django.contrib.auth',
 'django.contrib.contenttypes',
 'django.contrib.sessions',
 'django.contrib.messages',
```

```
'django.contrib.staticfiles',
'lists',
'accounts',
]
AUTH_USER_MODEL = 'accounts.User'
```

O próximo erro é um erro de banco de dados:

```
django.db.utils.OperationalError: no such table: accounts_user
```

Isso nos leva, como sempre, a fazer uma migração... Quando tentamos executá-la, o Django reclama que o nosso modelo personalizado de usuário não contém alguns metadados:

```
$ python manage.py makemigrations
```

```
Traceback (most recent call last):
```

```
[...]
```

```
if not isinstance(cls.REQUIRED_FIELDS, (list, tuple)):
```

```
AttributeError: type object 'User' has no attribute 'REQUIRED_FIELDS'
```

Suspiro. Por favor, Django, há somente um campo, portanto você deveria ser capaz de descobrir as respostas a essas perguntas sozinho. Aqui vamos nós:

## accounts/models.py

```
class User(models.Model):
 email = models.EmailField()
 REQUIRED_FIELDS = []
```

Próxima pergunta idiota?<sup>3</sup>

```
$ python manage.py makemigrations
```

```
[...]
```

```
AttributeError: type object 'User' has no attribute 'USERNAME_FIELD'
```

E executaremos mais alguns passos como esse até que tenhamos o seguinte:

## accounts/models.py

```
class User(models.Model):
 email = models.EmailField()
 REQUIRED_FIELDS = []
 USERNAME_FIELD = 'email'
 is_anonymous = False
```

```
is_authenticated = True
```

Agora temos um erro um pouco diferente:

```
$ python manage.py makemigrations
```

```
SystemCheckError: System check identified some issues:
```

```
ERRORS:
```

```
accounts.User: (auth.E003) 'User.email' must be unique because it is named as the 'USERNAME_FIELD'.
```

Bem, a maneira mais simples de corrigir isso seria assim:

## accounts/models.py (ch16|028-1)

```
email = models.EmailField(unique=True)
```

Agora a migração ocorre com sucesso:

```
$ python manage.py makemigrations
```

```
Migrations for 'accounts':
```

```
accounts/migrations/0001_initial.py
```

```
- Create model User
```

E o teste passa:

```
$ python manage.py test accounts
```

```
[...]
```

```
Ran 1 tests in 0.001s
```

```
OK
```

Entretanto, nosso modelo não é assim tão simples quanto poderia. Ele tem o campo de email, além de um campo “ID” gerado automaticamente como sua chave primária. Poderíamos tê-lo simplificado mais ainda!

## Testes como documentação

Vamos fazer todo o processo e transformar o campo de email na chave primária<sup>4</sup> e, desse modo, remover implicitamente a coluna de id gerada automaticamente.

Embora pudéssemos apenas fazer isso e nosso teste continuaria passando, argumentando, de modo concebível, que foi “somente uma refatoração”, seria melhor ter um teste específico:

## accounts/tests/test\_models.py (ch16|028-3)

```
def test_email_is_primary_key(self):
 user = User(email='a@b.com')
 self.assertEqual(user.pk, 'a@b.com')
```

O teste nos ajudará a lembrar caso algum dia retornemos ao código para vê-lo novamente no futuro:

```
self.assertEqual(user.pk, 'a@b.com')
AssertionError: None != 'a@b.com'
```



Seus testes podem ser uma forma de documentação para o código – eles expressam quais são os seus requisitos para uma classe ou função em particular. Às vezes, se você esquecer por que fez algo de determinada maneira, retornar e observar os testes lhe dará a resposta. É por isso que é importante dar nomes de métodos extensos e explícitos aos seus testes.

Eis a implementação (sinta-se à vontade para verificar o que acontece com `unique=True` antes):

## accounts/models.py (ch16|028-4)

```
email = models.EmailField(primary_key=True)
```

Não podemos nos esquecer de ajustar nossas migrações:

```
$ rm accounts/migrations/0001_initial.py
$ python manage.py makemigrations
Migrations for 'accounts':
 accounts/migrations/0001_initial.py
 - Create model User
```

Nossos dois testes passam:

```
$ python manage.py test accounts
[...]
Ran 2 tests in 0.001s
OK
```

## Um modelo de token para associar emails a um ID único

A seguir, vamos construir um modelo de token. Eis um pequeno



teste de unidade que captura a essência – você deverá ser capaz de associar um email a um ID único, e esse ID não deve ser o mesmo duas vezes seguidas:

## accounts/tests/test\_models.py (ch16l030)

```
from accounts.models import Token
[...]
```

```
class TokenModelTest(TestCase):
```

```
 def test_links_user_with_auto_generated_uid(self):
 token1 = Token.objects.create(email='a@b.com')
 token2 = Token.objects.create(email='a@b.com')
 self.assertNotEqual(token1.uid, token2.uid)
```

Orientar os modelos do Django com um TDD básico envolve passar por alguns obstáculos por causa da migração, portanto veremos algumas iterações como essa – alteração mínima de código, fazer migrações, obter um novo erro, apagar migração, recriar novas migrações, outra alteração de código, e assim por diante...

```
$ python manage.py makemigrations
```

```
Migrations for 'accounts':
 accounts/migrations/0002_token.py
 - Create model Token
```

```
$ python manage.py test accounts
```

```
[...]
```

```
TypeError: 'email' is an invalid keyword argument for this function
```

Confio em você para fazer tudo isso escrupulosamente – lembre-se de que posso não ser capaz de vê-lo, mas o Testing Goat pode!

```
$ rm accounts/migrations/0002_token.py
```

```
$ python manage.py makemigrations
```

```
Migrations for 'accounts':
 accounts/migrations/0002_token.py
 - Create model Token
```

```
$ python manage.py test accounts
```

```
AttributeError: 'Token' object has no attribute 'uid'
```

Em algum momento, você deverá obter o código a seguir...

## accounts/models.py (ch16l033)

```
class Token(models.Model):
 email = models.EmailField()
 uid = models.CharField(max_length=40)
```

e este erro:

```
$ python manage.py test accounts
[...]
```

```
self.assertNotEqual(token1.uid, token2.uid)
AssertionError: " == "
```

Nesse ponto, temos que decidir como gerar o nosso campo de ID único e aleatório. Poderíamos usar o módulo `random`, mas Python, na verdade, oferece outro módulo especificamente projetado para gerar IDs únicos; ele se chama “`uuid`” (universally unique id, ou id universalmente único).

Podemos usá-lo assim:

## accounts/models.py (ch16l035)

```
import uuid
[...]
```

```
class Token(models.Model):
 email = models.EmailField()
 uid = models.CharField(default=uuid.uuid4, max_length=40)
```

Trabalhando um pouco mais com migrações, devemos fazer com que os testes passem:

```
$ python manage.py test accounts
[...]
```

Ran 3 tests in 0.015s

OK

Bem, com isso, estamos no caminho! Pelo menos a camada de modelos está pronta. No próximo capítulo, veremos a simulação (mocking): uma técnica essencial para testar dependências externas, como o email.

# **Programação exploratória, spiking e de-spiking**

## *Spiking*

Programação exploratória para conhecer uma nova API ou explorar a viabilidade de uma nova solução. O spiking pode ser realizado sem testes. Fazer seu spike em um novo branch é uma boa ideia, retornando para o branch master quando fizer o de-spiking.

## *De-spiking*

Consiste em tomar o trabalho de um spike e torná-lo parte da base de código de produção. A ideia é jogar fora totalmente o código do spike antigo e começar de novo do zero, usando mais uma vez o TDD. O código de de-spiking muitas vezes pode acabar com um aspecto bem diferente do spike original, e geralmente será muito mais elegante.

## *Escrevendo o seu FT para um código de spiking*

Se essa é uma boa ideia ou não dependerá de suas circunstâncias. O motivo pelo qual isso pode ser conveniente é a possibilidade de ajudar você a escrever o FT de modo correto – descobrir como testar o seu spike pode ser simplesmente tão desafiador quanto o próprio spike. Por outro lado, poderá restringir você a implementar de novo uma solução bem semelhante àquela que você usou em seu spike – é algo no qual você deve prestar atenção.

- 
- 1 Eu não acabei de apresentar toda uma introdução insistindo nas implicações de privacidade ao usar o Google para login, somente para depois usar o Gmail, não é? Sim, é uma contradição (honestamente, vou deixar de usar o Gmail algum dia!). Nesse caso, porém, estou usando-o apenas para testes, e o importante é que não forço meus usuários a utilizar o Google.
  - 2 Uma decisão da qual você verá mantenedores de destaque do Django dizendo que atualmente se arrependem. Nem todos têm um primeiro nome e um sobrenome.
  - 3 Você poderia perguntar o seguinte: se eu acho que o Django é tão tolo, por que não submeto um pull request para corrigi-lo? Deveria ser uma correção simples. Bem, prometo que farei isso assim que acabar de escrever o livro. Por enquanto, comentários sarcásticos deverão ser suficientes.
  - 4 Emails talvez não sejam a chave primária perfeita na vida real. Um leitor, claramente com cicatrizes emocionais profundas, enviou-me um email lamentando o quanto sofreu por mais de uma década tentando lidar com os efeitos de emails como chaves primárias, por tornarem impossível o gerenciamento de contas de vários usuários. Então, como sempre, a sua experiência pode ser diferente.



## CAPÍTULO 19

# Usando mocks para testar dependências externas ou reduzir a duplicação

Neste capítulo, começaremos a testar as partes de nosso código que enviam emails. No FT, vimos que o Django nos dá uma forma de obter qualquer email enviado por ele, por meio do atributo `mail.outbox`. Neste capítulo, porém, gostaria de demonstrar uma técnica muito importante de teste chamada *simulação* (mocking); desse modo, para esses testes de unidade, fingiremos que esse belo atalho do Django não existe.



Eu estou dizendo a você que não use o `mail.outbox` do Django? Não; utilize-o, pois é um bom atalho. Contudo, quero ensinar você a usar mocks (simulações) porque eles são uma ferramenta conveniente de propósito geral para testes de unidade de dependências externas. Pode ser que você nem sempre vá usar o Django! Mesmo que use, talvez não envie emails – qualquer interação com uma API de terceiros é um bom candidato para testes com mocks.

## Antes de começar: definindo a infraestrutura básica

Vamos criar uma view básica e definir o URL antes. Podemos fazer isso com um teste simples, em que o nosso novo URL para enviar o email de login deve, em algum momento, fazer um redirecionamento para a página inicial:



## accounts/tests/test\_views.py

```
from django.test import TestCase
```

```
class SendLoginEmailViewTest(TestCase):
```

```
 def test_redirects_to_home_page(self):
 response = self.client.post('/accounts/send_login_email', data={
 'email': 'edith@example.com'
 })
 self.assertRedirects(response, '/')
```

Configure o include em *superlists/urls.py*, além do url em *accounts/urls.py*, e faça o teste passar com algo um pouco parecido com isto:

## accounts/views.py

```
from django.core.mail import send_mail
from django.shortcuts import redirect
```

```
def send_login_email(request):
 return redirect('/')
```

Adicionei a importação da função `send_mail` como um placeholder, por enquanto:

```
$ python manage.py test accounts
[...]
Ran 4 tests in 0.015s
OK
```

Tudo bem, agora temos um ponto de partida, portanto vamos começar a simular!

## Fazendo uma simulação manual, também conhecida como monkeypatching

Quando chamamos `send_mail` na vida real, esperamos que o Django faça uma conexão com o nosso provedor de emails e envie realmente um email pela internet pública. Não é algo que queremos

que aconteça em nossos testes. Você terá um problema semelhante sempre que tiver um código com efeitos colaterais externos – chamar uma API, enviar um tweet ou um SMS, ou o que quer que seja. Em nossos testes de unidade, não queremos enviar nossos verdadeiros tweets nem as chamadas de API pela internet. Contudo, ainda queremos ter um modo de testar se o nosso código está correto. Os mocks<sup>1</sup> são a resposta.

Na verdade, um dos ótimos aspectos positivos de Python é o fato de a sua natureza dinâmica facilitar bastante a execução de tarefas como a simulação, ou o que às vezes chamamos de monkeypatching ([https://en.wikipedia.org/wiki/Monkey\\_patch](https://en.wikipedia.org/wiki/Monkey_patch)). Vamos supor que, como primeiro passo, queiramos criar um código que chame `send_mail` com a linha correta para o assunto, o endereço do remetente e do destinatário. O código seria semelhante a este:

## accounts/views.py

```
def send_login_email(request):
 email = request.POST['email']
 # send_mail(
 # 'Your login link for Superlists',
 # 'body text tbc',
 # 'noreply@superlists',
 # [email],
 #)
 return redirect('/')
```

Como podemos testar isso sem chamar a *verdadeira* função `send_mail`? A resposta é a seguinte: nosso teste pode pedir a Python que substitua a função `send_mail` por uma versão falsa, em tempo de execução, antes que a view `send_login_email` seja chamada. Dê uma olhada neste código:

## accounts/tests/test\_views.py (ch17I005)

```
from django.test import TestCase
import accounts.views ❷

class SendLoginEmailViewTest(TestCase):
```

[...]

```
def test_sends_mail_to_address_from_post(self):
 self.send_mail_called = False
```

```
def fake_send_mail(subject, body, from_email, to_list): ❶
 self.send_mail_called = True
 self.subject = subject
 self.body = body
 self.from_email = from_email
 self.to_list = to_list
```

```
accounts.views.send_mail = fake_send_mail ❷
```

```
self.client.post('/accounts/send_login_email', data={
 'email': 'edith@example.com'
})
```

```
self.assertTrue(self.send_mail_called)
self.assertEqual(self.subject, 'Your login link for Superlists')
self.assertEqual(self.from_email, 'noreply@superlists')
self.assertEqual(self.to_list, ['edith@example.com'])
```

- ❶ Definimos uma função `fake_send_mail`, que parece a função `send_mail` real, mas tudo que ela faz é salvar algumas informações sobre como foi chamada, usando algumas variáveis em `self`.
- ❷ Então, antes que o código em teste seja executado chamando `self.client.post`, trocamos a `accounts.views.send_mail` verdadeira pela nossa versão falsa – é bem simples e basta fazer uma atribuição.

É importante perceber que não há nada realmente mágico acontecendo nesse caso; só estamos tirando proveito da natureza dinâmica de Python e das regras de escopo.

Até realmente chamarmos uma função, podemos modificar as variáveis às quais ela tem acesso, desde que acessemos o namespace correto (é por isso que importamos o módulo `accounts` de nível mais alto, a fim de acessarmos o módulo `accounts.views`, que é o escopo no qual a função `accounts.views.send_login_email` executará).

Isso não é algo que funcione somente em testes de unidade. Esse tipo de “monkeypatching” pode ser usado em qualquer tipo de código Python!

Talvez demore um pouco para que você absorva isso. Veja se você se convence de que esse código não é totalmente maluco, antes de ler um pouco mais de detalhes.

- Por que usamos `self` como forma de passar informações por aí? É apenas uma variável conveniente, disponível tanto no escopo da função `fake_send_mail` quanto fora dela. Poderíamos usar qualquer objeto mutável, como uma lista ou um dicionário, desde que façamos mudanças in-place em uma variável existente fora de nossa função falsa. (Sinta-se à vontade para brincar com diferentes maneiras de fazer isso se estiver curioso, e veja o que funciona e o que não funciona.)
- O “antes” é crucial! Nem posso dizer quantas vezes fiquei lá sentado, pensando por que um mock não estava funcionando, somente para me dar conta de que eu não havia feito um mock *antes* de chamar o código em teste.

Vamos ver se o nosso objeto mock implementado manualmente nos deixará gerar um pouco de código orientado a testes:

```
$ python manage.py test accounts
[...]
self.assertTrue(self.send_mail_called)
AssertionError: False is not true
```

Vamos então chamar `send_mail` ingenuamente:

## accounts/views.py

```
def send_login_email(request):
 send_mail()
 return redirect('/')
```

Eis o resultado:

```
TypeError: fake_send_mail() missing 4 required positional arguments: 'subject',
'body', 'from_email', and 'to_list'
```

Parece que o nosso monkeypatch está funcionando! Chamamos `send_mail` e nossa função `fake_send_mail` foi usada, a qual deseja ter mais argumentos. Vamos tentar isto:

## accounts/views.py

```
def send_login_email(request):
 send_mail('subject', 'body', 'from_email', ['to email'])
 return redirect('/')
```

Eis o resultado:

```
self.assertEqual(self.subject, 'Your login link for Superlists')
AssertionError: 'subject' != 'Your login link for Superlists'
```

Está funcionando muito bem. Agora podemos fazer todo o processo para obter um código semelhante a este:

## accounts/views.py

```
def send_login_email(request):
 email = request.POST['email']
 send_mail(
 'Your login link for Superlists',
 'body text tbc',
 'noreply@superlists',
 [email]
)
 return redirect('/')
```

e testes que passam!

```
$ python manage.py test accounts
```

```
Ran 5 tests in 0.016s
```

```
OK
```

Brilhante! Conseguimos escrever testes para um pouco de código que normalmente<sup>2</sup> faria um acesso externo e tentaria enviar emails de verdade pela internet; ao “simular” a função `send_email`, pudemos escrever os testes e o código do mesmo modo.

## Biblioteca Mock de Python

O pacote popular *mock* foi adicionado à biblioteca-padrão como parte do Python 3.3.<sup>3</sup> Ele disponibiliza um objeto mágico chamado Mock; teste-o em um shell Python:

```
>>> from unittest.mock import Mock
>>> m = Mock()
>>> m.any_attribute
<Mock name='mock.any_attribute' id='140716305179152'>
>>> type(m.any_attribute)
<class 'unittest.mock.Mock'>
>>> m.any_method()
<Mock name='mock.any_method()' id='140716331211856'>
>>> m.foo()
<Mock name='mock.foo()' id='140716331251600'>
>>> m.called
False
>>> m.foo.called
True
>>> m.bar.return_value = 1
>>> m.bar(42, var='thing')
1
>>> m.bar.call_args
call(42, var='thing')
```

Um objeto mágico que responda a qualquer requisição de um atributo ou de chamada de método com outros mocks, que pode ser configurado para devolver valores específicos para suas chamadas e que permite inspecionar com quais dados ele foi chamado? Parece ser conveniente usar algo assim em nossos testes de unidade!

## Usando unittest.patch

Como se isso não bastasse, o módulo *mock* também tem uma função auxiliar chamada *patch*, que pode ser usada para fazer o monkeypatching que fizemos manualmente antes.

Explicarei como tudo isso funciona em breve, mas vamos vê-lo em ação primeiro:

## accounts/tests/test\_views.py (ch17I007)

```
from django.test import TestCase
from unittest.mock import patch
[...]
```

```
@patch('accounts.views.send_mail')
def test_sends_mail_to_address_from_post(self, mock_send_mail):
 self.client.post('/accounts/send_login_email', data={
 'email': 'edith@example.com'
 })

 self.assertEqual(mock_send_mail.called, True)
 (subject, body, from_email, to_list), kwargs = mock_send_mail.call_args
 self.assertEqual(subject, 'Your login link for Superlists')
 self.assertEqual(from_email, 'noreply@superlists')
 self.assertEqual(to_list, ['edith@example.com'])
```

Se os testes forem executados novamente, veremos que continuam passando. Como devemos sempre suspeitar de qualquer teste que continue passando depois de uma mudança significativa, vamos provocar deliberadamente uma falha, apenas para conferir:

## accounts/tests/test\_views.py (ch17I008)

```
self.assertEqual(to_list, ['schmedith@example.com'])
```

Além disso, vamos adicionar um pequeno print de depuração em nossa view:

## accounts/views.py (ch17I009)

```
def send_login_email(request):
 email = request.POST['email']
 print(type(send_mail))
 send_mail(
 [...]
```

Vamos executar os testes novamente:

```
$ python manage.py test accounts
[...]
<class 'function'>
<class 'unittest.mock.MagicMock'>
```

```
[...]
AssertionError: Lists differ: ['edith@example.com'] !=
['schmedith@example.com']
[...]
```

Ran 5 tests in 0.024s

FAILED (failures=1)

Sem dúvida, os testes falham. Podemos ver que, imediatamente antes da mensagem de falha, quando exibimos o `type` da função `send_mail`, no primeiro teste de unidade, é uma função normal, mas, no segundo teste, vemos um objeto `mock`.

Vamos remover o erro proposital e detalhar exatamente o que está acontecendo:

## accounts/tests/test\_views.py (ch17I011)

```
@patch('accounts.views.send_mail') ❶
def test_sends_mail_to_address_from_post(self, mock_send_mail): ❷
 self.client.post('/accounts/send_login_email', data={
 'email': 'edith@example.com' ❸
 })

 self.assertEqual(mock_send_mail.called, True) ❹
 (subject, body, from_email, to_list), kwargs = mock_send_mail.call_args ❺
 self.assertEqual(subject, 'Your login link for Superlists')
 self.assertEqual(from_email, 'noreply@superlists')
 self.assertEqual(to_list, ['edith@example.com'])
```

❶ O decorador `patch` recebe o nome de um objeto com notação de ponto para fazer um monkeypatch. É equivalente a substituir manualmente `send_mail` em `accounts.views`. A vantagem do decorador é que, em primeiro lugar, ele substitui automaticamente o alvo por um `mock`. Em segundo lugar, o decorador coloca automaticamente o objeto original de volta no final! (Do contrário, o objeto continuaria sujeito a um monkeypatching pelo restante da execução do teste, o que poderia causar problemas em outros testes.)

❷ `patch` então injeta o objeto simulado no teste como argumento do



método de teste. Podemos escolher o nome que quisermos para ele, mas geralmente eu utilizo uma convenção com `mock_` mais o nome original do objeto.

- ③ Chamamos nossa função em teste como fazemos normalmente, mas tudo que estiver nesse método de teste terá o nosso mock aplicado a ele, portanto a view não chamará o objeto `send_mail` real; ela verá `mock_send_mail` em seu lugar.
- ④ Agora podemos fazer asserções sobre o que aconteceu com esse objeto mock durante o teste. Podemos ver que ele foi chamado...
- ⑤ ...e podemos também desempacotar seus vários argumentos posicionais e nomeados, além de analisar com quais dados ele foi chamado. (Mais tarde discutiremos `call_args` com um pouco mais de detalhes.)

Tudo claro como água? Não? Não se preocupe; realizaremos mais alguns testes com mocks para ver se eles começam a fazer mais sentido à medida que os usarmos com mais frequência.

## Avançando um pouco mais no FT

Inicialmente vamos retomar o nosso FT e ver em que ponto ele está falhando:

```
$ python manage.py test functional_tests.test_login
```

```
[...]
```

```
AssertionError: 'Check your email' not found in 'Superlists\nEnter email to log
in:\nStart a new To-Do list'
```

Submeter o endereço de email no momento não tem efeito, pois o formulário não está enviando os dados para nenhum lugar. Vamos configurá-lo em *base.html*:<sup>4</sup>

### lists/templates/base.html (ch17I012)

```
<form class="navbar-form navbar-right"
 method="POST"
 action="{% url 'send_login_email' %}">
```

Isso ajuda? Não, temos o mesmo erro. Por quê? Porque não estamos realmente exibindo uma mensagem de sucesso depois que

enviamos um email ao usuário. Vamos acrescentar um teste para isso.

## Testando o framework de mensagens do Django

Usaremos o “framework de mensagens” do Django, que geralmente é utilizado para exibir mensagens efêmeras de “sucesso” ou “avisos”, para mostrar os resultados de uma ação. Consulte a documentação de mensagens do Django (<https://docs.djangoproject.com/en/1.11/ref/contrib/messages/>) caso ainda não o tenha visto antes.

Testar mensagens do Django é um caminho um pouco tortuoso – temos que passar `follow=True` para o cliente de testes para lhe dizer que obtenha a página após o redirecionamento 302 e analise o seu contexto em busca de uma lista de mensagens (que precisamos listar antes para que ela funcione apropriadamente). Eis a aparência disso:

### accounts/tests/test\_views.py (ch17|013)

```
def test_adds_success_message(self):
 response = self.client.post('/accounts/send_login_email', data={
 'email': 'edith@example.com'
 }, follow=True)

 message = list(response.context['messages'])[0]
 self.assertEqual(
 message.message,
 "Check your email, we've sent you a link you can use to log in."
)
 self.assertEqual(message.tags, "success")
```

O resultado será este:

```
$ python manage.py test accounts
[...]
message = list(response.context['messages'])[0]
IndexError: list index out of range
```

Podemos fazer o teste passar com:

## accounts/views.py (ch17l014)

```
from django.contrib import messages
[...]

def send_login_email(request):
 [...]
 messages.success(
 request,
 "Check your email, we've sent you a link you can use to log in."
)
 return redirect('/')
```

## Mocks podem deixar você altamente acoplado com a implementação



Esta caixa de texto é uma dica de testes de nível intermediário. Se você não compreender na primeira vez que o ler, retorne e leia novamente quando tiver terminado este capítulo e o Capítulo 23.

Eu disse que as mensagens de testes têm um caminho um pouco tortuoso; tive que fazer várias tentativas para que estivessem corretas. Com efeito, no trabalho, nós desistimos de testá-las assim e decidimos simplesmente usar mocks. Vamos ver como seria a aparência do teste nesse caso:

## accounts/tests/test\_views.py (ch17l014-2)

```
from unittest.mock import patch, call
[...]

@patch('accounts.views.messages')
def test_adds_success_message_with_mocks(self, mock_messages):
 response = self.client.post('/accounts/send_login_email', data={
 'email': 'edith@example.com'
 })

 expected = "Check your email, we've sent you a link you can use to log in."
 self.assertEqual(
 mock_messages.success.call_args,
 call(response.wsgi_request, expected),
```

)

Simulamos o módulo `messages` e verificamos se `messages.success` foi chamada com os argumentos corretos: a requisição original e a mensagem que queremos.

Você poderia fazer o teste passar usando exatamente o mesmo código de antes. Contudo, eis o problema: o framework `messages` oferece mais de uma maneira de obter o mesmo resultado. Eu poderia ter implementado o código assim:

## `accounts/views.py` (ch17I014-3)

```
messages.add_message(
 request,
 messages.SUCCESS,
 "Check your email, we've sent you a link you can use to log in."
)
```

O teste original, sem mock, continuaria passando. Entretanto, o nosso teste com mock falhará, pois não estamos mais chamando `messages.success`; estamos chamando `messages.add_message`. Apesar de o resultado final ser o mesmo e o nosso código estar “correto”, o teste apresenta falha. É isso o que as pessoas querem dizer quando falam que usar mocks pode deixar você “altamente acoplado com a implementação”. Em geral, dizemos que é melhor testar comportamentos, e não os detalhes de implementação; teste o que acontece, e não como você o faz. Os mocks, com frequência, acabam pecando muito pelo lado do “como”, e não do “o quê”.

Há discussões mais detalhadas sobre os prós e contras dos mocks em capítulos mais adiante.

## Adicionando mensagens em nosso HTML

O que acontece em seguida no teste funcional? Ah. Nada ainda. Precisamos adicionar as mensagens na página. Algo como:

## `lists/templates/base.html` (ch17I015)

```
[...]
</nav>

{% if messages %}
```

```

<div class="row">
 <div class="col-md-8">
 {% for message in messages %}
 {% if message.level_tag == 'success' %}
 <div class="alert alert-success">{{ message }}</div>
 {% else %}
 <div class="alert alert-warning">{{ message }}</div>
 {% endif %}
 {% endfor %}
 </div>
</div>
{% endif %}

```

Agora avançamos um pouco? Sim!

```
$ python manage.py test accounts
```

```
[...]
```

```
Ran 6 tests in 0.023s
```

OK

```
$ python manage.py test functional_tests.test_login
```

```
[...]
```

```
AssertionError: 'Use this link to log in' not found in 'body text tbc'
```

Precisamos preencher o texto de corpo do email com um link que o usuário possa usar para fazer login.

No entanto, vamos apenas trapacear por enquanto, alterando o valor na view:

### accounts/views.py

```

send_mail(
 'Your login link for Superlists',
 'Use this link to log in',
 'noreply@superlists',
 [email]
)

```

Isso faz com que o FT avance um pouco mais:

```
$ python manage.py test functional_tests.test_login
```

```
[...]
```

```
AssertionError: Could not find url in email body:
```

Use this link to log in

## Começando pelo URL de login

Vamos ter que construir algum tipo de URL! Construiremos um que, mais uma vez, simplesmente trapaceie:

`accounts/tests/test_views.py (ch17I017)`

```
class LoginViewTest(TestCase):
```

```
 def test_redirects_to_home_page(self):
 response = self.client.get('/accounts/login?token=abcd123')
 self.assertRedirects(response, '/')
```

Estamos supondo que passaremos o token em um parâmetro de GET, após o ?. Ele não precisa fazer nada por enquanto.

Tenho certeza de que você conseguirá obter o boilerplate para um URL básico e uma view, por meio de erros como estes:

- Sem URL:

AssertionError: 404 != 302 : Response didn't redirect as expected: Response code was 404 (expected 302)

- Sem view:

AttributeError: module 'accounts.views' has no attribute 'login'

- View com falha:



ValueError: The view accounts.views.login didn't return an HttpResponseRedirect object.  
It returned None instead.

- Tudo bem!

```
$ python manage.py test accounts
[...]
```

```
Ran 7 tests in 0.029s
```

```
OK
```

Agora podemos dar aos usuários um link para ser utilizado. Contudo, ele ainda não fará muito, pois continuamos sem um token para lhe oferecer.

## Verificando se enviamos um link com um token para o usuário

De volta à nossa view `send_login_email`, testamos os campos de assunto, remetente e destinatário do email. O corpo é a parte que deverá incluir um token ou um URL que possa ser usado para login. Vamos gerar a especificação de dois testes para isso:

### accounts/tests/test\_views.py (ch171021)

```
from accounts.models import Token
[...]
```

```
def test_creates_token_associated_with_email(self):
 self.client.post('/accounts/send_login_email', data={
 'email': 'edith@example.com'
 })
 token = Token.objects.first()
 self.assertEqual(token.email, 'edith@example.com')
```

```
@patch('accounts.views.send_mail')
def test_sends_link_to_login_using_token_uid(self, mock_send_mail):
 self.client.post('/accounts/send_login_email', data={
 'email': 'edith@example.com'
 })

 token = Token.objects.first()
 expected_url = f'http://testserver/accounts/login?token={token.uid}'
```

```
(subject, body, from_email, to_list), kwargs = mock_send_mail.call_args
self.assertIn(expected_url, body)
```

O primeiro teste é bem simples; ele verifica se o token que criamos no banco de dados está associado ao endereço de email da requisição post.

O segundo teste é o segundo que usa mocks. Simulamos a função `send_mail` novamente usando o decorador `patch`, mas, dessa vez, estamos interessados no argumento `body` dos argumentos da chamada.

A execução desses testes agora resultará em falha porque não estamos criando nenhum tipo de token:

```
$ python manage.py test accounts
[...]
AttributeError: 'NoneType' object has no attribute 'email'
[...]
AttributeError: 'NoneType' object has no attribute 'uid'
```

Podemos fazer o primeiro teste passar criando um token:

## accounts/views.py (ch17I022)

```
from accounts.models import Token
[...]

def send_login_email(request):
 email = request.POST['email']
 token = Token.objects.create(email=email)
 send_mail(
 [...]
```

Agora o segundo teste exige que usemos realmente o token no corpo de nosso email:

```
[...]
AssertionError:
'http://testserver/accounts/login?token=[...]'
not found in 'Use this link to log in'
```

```
FAILED (failures=1)
```

Desse modo, podemos inserir o token em nosso email, assim:

## accounts/views.py (ch17|023)

```
from django.core.urlresolvers import reverse
[...]
```

```
def send_login_email(request):
 email = request.POST['email']
 token = Token.objects.create(email=email)
 url = request.build_absolute_uri(❶
 reverse('login') + '?token=' + str(token.uid)
)
 message_body = f'Use this link to log in:\n\n{url}'
 send_mail(
 'Your login link for Superlists',
 message_body,
 'noreply@superlists',
 [email]
)
[...]
```

- ❶ `request.build_absolute_uri` merece ser mencionado – é uma forma de construir um URL “completo”, incluindo o nome do domínio e a parte do `http(s)`, no Django. Há outras maneiras, mas geralmente elas envolvem entrar no framework de “sites”, e isso se tornará rapidamente complicado demais. Você poderá ver muitas outras discussões sobre esse assunto caso esteja curioso, pesquisando um pouco no Google.

Mais duas peças do quebra-cabeça. Precisamos de um backend de autenticação, cuja tarefa será analisar os tokens para saber se são válidos e, então, devolver os usuários correspondentes; em seguida, precisaremos fazer com que a nossa view de login realmente faça o login dos usuários, se puderem ser autenticados.

## De-spiking de nosso backend personalizado de autenticação

Nosso backend personalizado de autenticação vem a seguir. Eis a sua aparência no spike:

```

class PasswordlessAuthenticationBackend(object):

 def authenticate(self, uid):
 print('uid', uid, file=sys.stderr)
 if not Token.objects.filter(uid=uid).exists():
 print('no token found', file=sys.stderr)
 return None
 token = Token.objects.get(uid=uid)
 print('got token', file=sys.stderr)
 try:
 user = ListUser.objects.get(email=token.email)
 print('got user', file=sys.stderr)
 return user
 except ListUser.DoesNotExist:
 print('new user', file=sys.stderr)
 return ListUser.objects.create(email=token.email)

 def get_user(self, email):
 return ListUser.objects.get(email=email)

```

Decodificando esse código:

- Tomamos um UID e verificamos se ele está presente no banco de dados.
- Devolvemos None se não estiver.
- Se existir, extraímos um endereço de email e encontramos um usuário com esse endereço ou criamos um novo usuário.

## Um if = mais um teste

Eis uma regra geral para esses tipos de teste: qualquer if implica um teste extra, e qualquer try/except significa um teste extra, portanto devemos ter cerca de três testes. Que tal algo como o teste a seguir?

`accounts/tests/test_authentication.py`

```

from django.test import TestCase
from django.contrib.auth import get_user_model

```

```
from accounts.authentication import PasswordlessAuthenticationBackend
from accounts.models import Token
User = get_user_model()
```

```
class AuthenticateTest(TestCase):
```

```
 def test_returns_None_if_no_such_token(self):
 result = PasswordlessAuthenticationBackend().authenticate(
 'no-such-token'
)
 self.assertIsNone(result)
```

```
 def test_returns_new_user_with_correct_email_if_token_exists(self):
 email = 'edith@example.com'
 token = Token.objects.create(email=email)
 user = PasswordlessAuthenticationBackend().authenticate(token.uid)
 new_user = User.objects.get(email=email)
 self.assertEqual(user, new_user)
```

```
 def test_returns_existing_user_with_correct_email_if_token_exists(self):
 email = 'edith@example.com'
 existing_user = User.objects.create(email=email)
 token = Token.objects.create(email=email)
 user = PasswordlessAuthenticationBackend().authenticate(token.uid)
 self.assertEqual(user, existing_user)
```

Em *authenticate.py*, teremos somente um pequeno placeholder:

## accounts/authentication.py

```
class PasswordlessAuthenticationBackend(object):
```

```
 def authenticate(self, uid):
 pass
```

Como estamos nos saindo?

```
$ python manage.py test accounts
```

```
.FE.....
```

```
=====
=====
```

```
ERROR: test_returns_new_user_with_correct_email_if_token_exists
(accounts.tests.test_authentication.AuthenticateTest)
```

```

Traceback (most recent call last):
```

```
File "../superlists/accounts/tests/test_authentication.py", line 21, in
test_returns_new_user_with_correct_email_if_token_exists
 new_user = User.objects.get(email=email)
```

```
[...]
```

```
accounts.models.DoesNotExist: User matching query does not exist.
```

```
=====
=====
```

```
FAIL: test_returns_existing_user_with_correct_email_if_token_exists
(accounts.tests.test_authentication.AuthenticateTest)
```

```

Traceback (most recent call last):
```

```
File "../superlists/accounts/tests/test_authentication.py", line 30, in
test_returns_existing_user_with_correct_email_if_token_exists
 self.assertEqual(user, existing_user)
```

```
AssertionError: None != <User: User object>
```

```

Ran 12 tests in 0.038s
```

```
FAILED (failures=1, errors=1)
```

Eis uma primeira tentativa:

## accounts/authentication.py (ch17I026)

```
from accounts.models import User, Token
```

```
class PasswordlessAuthenticationBackend(object):
```

```
 def authenticate(self, uid):
 token = Token.objects.get(uid=uid)
 return User.objects.get(email=token.email)
```

Com isso, um teste passa, porém o outro falha:

```
$ python manage.py test accounts
```

```
ERROR: test_returns_None_if_no_such_token
(accounts.tests.test_authentication.AuthenticateTest)
```

accounts.models.DoesNotExist: Token matching query does not exist.

```
ERROR: test_returns_new_user_with_correct_email_if_token_exists
(accounts.tests.test_authentication.AuthenticateTest)
```

[...]

accounts.models.DoesNotExist: User matching query does not exist.

Vamos corrigir cada um desses pontos individualmente:

## accounts/authentication.py (ch17I027)

```
def authenticate(self, uid):
 try:
 token = Token.objects.get(uid=uid)
 return User.objects.get(email=token.email)
 except Token.DoesNotExist:
 return None
```

Agora nos resta uma só falha:

```
ERROR: test_returns_new_user_with_correct_email_if_token_exists
(accounts.tests.test_authentication.AuthenticateTest)
```

[...]

accounts.models.DoesNotExist: User matching query does not exist.

FAILED (errors=1)

Podemos tratar o último caso assim:

## accounts/authentication.py (ch17I028)

```
def authenticate(self, uid):
 try:
 token = Token.objects.get(uid=uid)
 return User.objects.get(email=token.email)
 except User.DoesNotExist:
 return User.objects.create(email=token.email)
 except Token.DoesNotExist:
 return None
```

O código está mais elegante que o de nosso spike!



## Método `get_user`

Cuidamos da função `authenticate`, que o Django usará para fazer login de novos usuários. A segunda parte do protocolo que temos que implementar é o método `get_user`, cuja tarefa é obter um usuário de acordo com o seu identificador único (o endereço de email), ou devolver `None` caso não seja possível encontrar um (observe novamente o código do spike se precisar recordar).

Eis um par de testes para esses dois requisitos:

### `accounts/tests/test_authentication.py` (ch17I030)

```
class GetUserTest(TestCase):

 def test_gets_user_by_email(self):
 User.objects.create(email='another@example.com')
 desired_user = User.objects.create(email='edith@example.com')
 found_user = PasswordlessAuthenticationBackend().get_user(
 'edith@example.com'
)
 self.assertEqual(found_user, desired_user)

 def test_returns_None_if_no_user_with_that_email(self):
 self.assertIsNone(
 PasswordlessAuthenticationBackend().get_user('edith@example.com')
)
```

Esta é a nossa primeira falha:

```
AttributeError: 'PasswordlessAuthenticationBackend' object has no attribute
'get_user'
```

Vamos criar, então, outro placeholder:

### `accounts/authentication.py` (ch17I031)

```
class PasswordlessAuthenticationBackend(object):

 def authenticate(self, uid):
 [...]
```

```
def get_user(self, email):
 pass
```

Agora temos:

```
self.assertEqual(found_user, desired_user)
AssertionError: None != <User: User object>
```

E (passo a passo, somente para ver se o nosso teste falha do modo como achamos que o fará):

## accounts/authentication.py (ch17I033)

```
def get_user(self, email):
 return User.objects.first()
```

Com isso, passamos pela primeira asserção e chegamos a:

```
self.assertEqual(found_user, desired_user)
AssertionError: <User: User object> != <User: User object>
```

Então chamamos get com o email como argumento:

## accounts/authentication.py (ch17I034)

```
def get_user(self, email):
 return User.objects.get(email=email)
```

Agora o nosso teste para o caso None falha:

```
ERROR: test_returns_None_if_no_user_with_that_email
[...]
accounts.models.DoesNotExist: User matching query does not exist.
```

Isso nos leva a finalizar o método assim:

## accounts/authentication.py (ch17I035)

```
def get_user(self, email):
 try:
 return User.objects.get(email=email)
 except User.DoesNotExist:
 return None ❶
```

- ❶ Você poderia simplesmente usar `pass` aqui, e a função devolveria `None` por padrão. No entanto, como precisamos especificamente que a função devolva `None`, a regra do “explícito é melhor do que implícito” se aplica nesse caso.

Com isso, os testes passam:

OK

Temos um backend de autenticação funcionando!

## Usando o nosso backend de autenticação na view de login

O último passo é usar o backend em nossa view de login. Inicialmente vamos adicioná-lo em *settings.py*:

### superlists/settings.py (ch17I036)

```
AUTH_USER_MODEL = 'accounts.User'
AUTHENTICATION_BACKENDS = [
 'accounts.authentication.PasswordlessAuthenticationBackend',
]
```

[...]

Em seguida, vamos escrever alguns testes para o que deve acontecer em nossa view. Vamos observar novamente o spike:

### accounts/views.py

```
def login(request):
 print('login view', file=sys.stderr)
 uid = request.GET.get('uid')
 user = auth.authenticate(uid=uid)
 if user is not None:
 auth.login(request, user)
 return redirect('/')
```

Precisamos que a view chame `django.contrib.auth.authenticate` e, então, caso um usuário seja devolvido, chamamos `django.contrib.auth.login`.



Essa é uma boa hora para consultar a documentação do Django sobre autenticação (<https://docs.djangoproject.com/en/1.11/topics/auth/default/#how-to-log-a-user-in>) para ter um pouco mais de contexto.

## Uma razão alternativa para usar mocks: reduzir a duplicação

Até agora, usamos mocks para testar dependências externas, por exemplo, a função de envio de emails do Django. O principal motivo para usar um mock foi nos isolarmos dos efeitos colaterais externos – nesse caso, para evitar o envio de emails de verdade durante os nossos testes.

Nesta seção, veremos um tipo de uso diferente para os mocks. Nesse cenário, não temos nenhum efeito colateral com o qual estejamos preocupados, mas há ainda alguns motivos pelos quais vamos querer usar um mock.

O modo de testar essa view de login sem mocks seria ver se ela realmente faz o login do usuário, verificando se ele recebe um cookie de sessão autenticada nas circunstâncias corretas.

Contudo, o nosso backend de autenticação tem alguns caminhos diferentes no código: devolve None para tokens inválidos, usuários existentes caso esses já existam e cria novos usuários para tokens válidos se ainda não existirem. Desse modo, para testar essa view por inteiro, eu teria que escrever testes para esses três casos.



Uma boa justificativa para o uso de mocks é quando eles reduzem a duplicação entre os testes. É uma forma de evitar uma *explosão combinatória*.

Além do mais, o fato de estarmos usando a função `auth.authenticate` do Django em vez de chamar o nosso próprio código diretamente é relevante: isso nos permite ter a opção de adicionar outros backends no futuro.

Assim, nesse caso (em comparação com o exemplo na caixa de texto “Mocks podem deixar você altamente acoplado com a implementação”), a implementação não importa, e usar um mock evitará que tenhamos duplicação em nossos testes. Vamos ver como será a aparência desse teste:

## accounts/tests/test\_views.py (ch17I037)

```
from unittest.mock import patch, call
[...]
```

```
@patch('accounts.views.auth') ❶
def test_calls_authenticate_with_uid_from_get_request(self, mock_auth): ❷
 self.client.get('/accounts/login?token=abcd123')
 self.assertEqual(
 mock_auth.authenticate.call_args, ❸
 call(uid='abcd123') ❹
)
```

- ❶ Esperamos usar o módulo `django.contrib.auth` em `views.py` e fazemos a sua simulação aqui. Observe que, dessa vez, não estamos simulando uma função, mas, sim, um módulo completo e, desse modo, simulando implicitamente todas as funções (e qualquer outro objeto) que o módulo contém.
- ❷ Como sempre, o objeto simulado é injetado em nosso método de teste.
- ❸ Dessa vez, simulamos um módulo, e não uma função. Portanto, analisamos o `call_args` da função `mock_auth.authenticate`, e não do módulo `mock_auth`. Como todos os atributos de um mock são mais mocks, esse é um mock também. Você pode começar a ver por que objetos `Mock` são tão convenientes, se compararmos com a tentativa de construir os nossos próprios mocks.
- ❹ Agora, em vez de “desempacotar” os argumentos da chamada, usamos a função `call` para ter uma maneira mais elegante de dizer com quais dados ela deveria ter sido chamada – isto é, o token da requisição GET. (Veja a caixa de texto a seguir.)

### Sobre o `call_args` de mocks

A propriedade `call_args` em um mock representa os argumentos posicionais e nomeados com os quais o mock foi chamado. É um tipo especial de objeto de “chamada”, que é essencialmente uma tupla de (`positional_args`, `keyword_args`). `positional_args`, por si só, é uma tupla, constituída do conjunto de argumentos posicionais. `keyword_args` é um dicionário.

```
>>> from unittest.mock import Mock, call
>>> m = Mock()
>>> m(42, 43, 'positional arg 3', key='val', thing=666)
<Mock name='mock()' id='139909729163528'>

>>> m.call_args
call(42, 43, 'positional arg 3', key='val', thing=666)

>>> m.call_args == ((42, 43, 'positional arg 3'), {'key': 'val', 'thing': 666})
True
>>> m.call_args == call(42, 43, 'positional arg 3', key='val', thing=666)
True
```

Desse modo, em nosso teste, como alternativa, poderíamos ter feito isto:

## accounts/tests/test\_views.py

```
self.assertEqual(
 mock_auth.authenticate.call_args,
 ((,), {'uid': 'abcd123'})
)
ou isto
args, kwargs = mock_auth.authenticate.call_args
self.assertEqual(args, (,))
self.assertEqual(kwargs, {'uid': 'abcd123'})
```

No entanto, você pode ver como usar a função auxiliar `call` é melhor.

O que acontecerá se executarmos o teste? Eis o primeiro erro:

```
$ python manage.py test accounts
```

```
[...]
```

```
AttributeError: <module 'accounts.views' from
'../../superlists/accounts/views.py'> does not have the attribute 'auth'
```



module `foo` does not have the attribute `bar` (o módulo `foo` não tem o atributo `bar`) é uma primeira falha comum em um teste que utilize mocks. Ela informa que você está tentando simular algo que ainda não existe (ou que ainda não foi importado) no módulo alvo.

Após a importação de `django.contrib.auth`, o erro muda:

## accounts/views.py (ch17I038)

```
from django.contrib import auth, messages
[...]
```

Agora temos:

```
AssertionError: None != call(uid='abcd123')
```

A falha agora está nos informando que a view não chama a função `auth.authenticate` de modo algum. Vamos corrigir isso, mas façamos com que haja uma falha proposital, somente para conferir:

## accounts/views.py (ch17I039)

```
def login(request):
 auth.authenticate('bang!')
 return redirect('/')
```

Tudo certo!

```
$ python manage.py test accounts
[...]
```

```
AssertionError: call('bang!') != call(uid='abcd123')
```

```
[...]
```

```
FAILED (failures=1)
```

Vamos dar a `authenticate` os argumentos que ela espera:

## accounts/views.py (ch17I040)

```
def login(request):
 auth.authenticate(uid=request.GET.get('token'))
 return redirect('/')
```

Com isso, temos testes que passam:

```
$ python manage.py test accounts
[...]
```

```
Ran 15 tests in 0.041s
```

```
OK
```

## Usando `mock.return_value`

A seguir queremos verificar se, no caso de a função de autenticação devolver um usuário, esse será passado para `auth.login`. Vamos ver como é a aparência desse teste:

## accounts/tests/test\_views.py (ch17I041)

```
@patch('accounts.views.auth') ❶
def test_calls_auth_login_with_user_if_there_is_one(self, mock_auth):
 response = self.client.get('/accounts/login?token=abcd123')
 self.assertEqual(
 mock_auth.login.call_args, ❷
 call(response.wsgi_request, mock_auth.authenticate.return_value) ❸
)
```

- ❶ Simulamos o módulo `contrib.auth` novamente.
- ❷ Dessa vez, analisamos os argumentos de chamada da função `auth.login`.
- ❸ Verificamos se ela é chamada com o objeto de requisição que a view vê, e o objeto de “usuário” que a função `authenticate` devolve. Como `authenticate` também está sendo simulada, podemos usar o seu atributo especial “`return_value`”.

Quando chamamos um mock, temos outro mock. Contudo, você também pode obter uma cópia desse mock devolvido a partir do mock original que você chamou. Cara, com certeza é difícil explicar isso sem mencionar um bocado de “mock”! Outra pequena demonstração no console pode ajudar nesse caso:

```
>>> m = Mock()
>>> thing = m()
>>> thing
<Mock name='mock()' id='140652722034952'>
>>> m.return_value
<Mock name='mock()' id='140652722034952'>
>>> thing == m.return_value
True
```

De qualquer modo, o que temos como resultado da execução do teste?

```
$ python manage.py test accounts
```

```
[...]
```

```
 call(response.wsgi_request, mock_auth.authenticate.return_value)
AssertionError: None != call(<WSGIRequest: GET '/accounts/login?t[...]
```

Com certeza, ele está nos dizendo que, definitivamente, não



estamos chamando `auth.login` ainda. Vamos tentar fazer isso. Antes, como sempre, propositalmente incorreto!

## accounts/views.py (ch17I042)

```
def login(request):
 auth.authenticate(uid=request.GET.get('token'))
 auth.login('ack!')
 return redirect('/')
```

Tudo certo!

```
TypeError: login() missing 1 required positional argument: 'user'
[...]
AssertionError: call('ack!') != call(<WSGIRequest: GET
'/accounts/login?token=[...]')
```

Vamos corrigir isso:

## accounts/views.py (ch17I043)

```
def login(request):
 user = auth.authenticate(uid=request.GET.get('token'))
 auth.login(request, user)
 return redirect('/')
```

Agora temos esta reclamação inesperada:

```
ERROR: test_redirects_to_home_page
(accounts.tests.test_views.LoginViewTest)
[...]
AttributeError: 'AnonymousUser' object has no attribute '_meta'
```

Ela ocorre porque continuamos chamando `auth.login` indiscriminadamente, em qualquer tipo de usuário, e isso está causando problemas em nosso teste original para o redirecionamento, que, no momento, *não está* simulando `auth.login`. Precisamos adicionar um `if` (e, desse modo, criar outro teste); aproveitando a ocasião, veremos o que é o patching no nível de classe.

## Patching no nível de classe

Queremos adicionar outro teste, com outro

@patch('accounts.views.auth'), e isso está começando a ficar repetitivo. Usando a regra dos “três acertos”, podemos passar o decorador patch para o nível de classe. Isso terá como efeito simular accounts.views.auth em todos os métodos de teste dessa classe. Também significa que nosso teste original de redirecionamento agora terá também a variável mock\_auth injetada:

## accounts/tests/test\_views.py (ch171044)

```
@patch('accounts.views.auth') ❶
class LoginViewTest(TestCase):

 def test_redirects_to_home_page(self, mock_auth): ❷
 [...]

 def test_calls_authenticate_with_uid_from_get_request(self, mock_auth): ❸
 [...]

 def test_calls_auth_login_with_user_if_there_is_one(self, mock_auth): ❹
 [...]

 def test_does_not_login_if_user_is_not_authenticated(self, mock_auth):
 mock_auth.authenticate.return_value = None ❺
 self.client.get('/accounts/login?token=abcd123')
 self.assertEqual(mock_auth.login.called, False) ❻
```

- ❶ Passamos o patch para o nível da classe...
- ❷ o que significa que temos um argumento extrainjetado em nosso primeiro método de teste...
- ❸ Podemos remover os decoradores de todos os demais testes.
- ❹ Em nosso novo teste, definimos explicitamente o return\_value em mock\_auth.authenticate, *antes* de chamar self.client.get.
- ❺ Fazemos uma asserção para verificar se, caso authenticate devolva None, não devemos chamar auth.login de forma alguma.

Com isso, limpamos a falha espúria, e temos uma falha específica e esperada na qual trabalharemos:

```
self.assertEqual(mock_auth.login.called, False)
AssertionError: True != False
```

E fazemos com que o teste passe, assim:

## accounts/views.py (ch17I045)

```
def login(request):
 user = auth.authenticate(uid=request.GET.get('token'))
 if user:
 auth.login(request, user)
 return redirect('/')
```

Nós já chegamos lá?

## Hora da verdade: o FT passará?

Acho que estamos praticamente prontos para experimentar o nosso teste funcional!

Vamos somente garantir que o nosso template-base mostre uma barra de navegação diferente para usuários que fizeram login e para os que não o fizeram (nosso FT conta com isso):

## lists/templates/base.html (ch17I046)

```
<nav class="navbar navbar-default" role="navigation">
 <div class="container-fluid">
 Superlists
 {% if user.email %}
 <ul class="nav navbar-nav navbar-right">
 <li class="navbar-text">Logged in as {{ user.email }}
 Log out

 {% else %}
 <form class="navbar-form navbar-right"
 method="POST"
 action="{% url 'send_login_email' %}">
 Enter email to log in:
 <input class="form-control" name="email" type="text" />
 {% csrf_token %}
 </form>
 {% endif %}
```

```
</div>
</nav>
```

E verifique se...

```
$ python manage.py test functional_tests.test_login
Internal Server Error: /accounts/login
[...]
File ".../superlists/accounts/views.py", line 31, in login
 auth.login(request, user)
[...]
ValueError: The following fields do not exist in this model or are m2m fields:
last_login
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate
element: Log out
```

Oh, não! Algo não está certo. Porém, supondo que a configuração de LOGGING em *settings.py* tenha sido mantida, você deverá ver o traceback explicativo, conforme acabou de ser mostrado. Ele informa algo sobre um campo `last_login`.

Em minha opinião (<https://code.djangoproject.com/ticket/26823>), esse é um bug de Django, mas, essencialmente, o framework de autenticação espera que o modelo de usuário tenha um campo `last_login`. Não temos um. Não temos, porém! Há uma forma de lidar com essa falha.

Vamos escrever antes um teste de unidade que reproduza o bug. Como ele está relacionado ao nosso modelo personalizado de usuário, um bom lugar para tê-lo seria em *test\_models.py*:

## accounts/tests/test\_models.py (ch17I047)

```
from django.test import TestCase
from django.contrib import auth
from accounts.models import Token
User = auth.get_user_model()
```

```
class UserModelTest(TestCase):
```

```

def test_user_is_valid_with_email_only(self):
 [...]
def test_email_is_primary_key(self):
 [...]

def test_no_problem_with_auth_login(self):
 user = User.objects.create(email='edith@example.com')
 user.backend = ""
 request = self.client.request().wsgi_request
 auth.login(request, user) # não deve gerar erro

```

Criamos um objeto de requisição e um usuário e, em seguida, passamos esses dados para a função `auth.login`.

Isso fará o nosso erro ser gerado:

```

 auth.login(request, user) # não deve gerar erro
[...]
```

ValueError: The following fields do not exist in this model or are m2m fields:  
last\_login

O motivo específico para a existência desse bug não é de fato importante para este livro, mas, se estiver curioso sobre o que está ocorrendo exatamente nesse caso, consulte as linhas de código-fonte do Django listadas no traceback e leia a sua documentação sobre sinais (<https://docs.djangoproject.com/en/1.11/topics/signals/>).

A boa notícia é que podemos corrigir esse problema assim:

## accounts/models.py (ch17I048)

```

import uuid
from django.contrib import auth
from django.db import models

auth.signals.user_logged_in.disconnect(auth.models.update_last_login)

class User(models.Model):
 [...]

```

Como está o nosso FT agora?

```
$ python manage.py test functional_tests.test_login
```

[...]

.

-----  
Ran 1 test in 3.282s

OK

## Teoricamente o FT funciona! Ele funciona na prática?

Uau! Você acredita nisso? Mal posso crer! É hora de conferir manualmente com runserver:

```
$ python manage.py runserver
```

[...]

Internal Server Error: /accounts/send\_login\_email

Traceback (most recent call last):

File ".../superlists/accounts/views.py", line 20, in send\_login\_email

ConnectionRefusedError: [Errno 111] Connection refused

Provavelmente você verá um erro, como ocorreu comigo, quando tentar executar tudo manualmente. Há dois problemas possíveis:

- Em primeiro lugar, precisamos adicionar de novo a configuração de email em *settings.py*.
- Em segundo, provavelmente precisaremos fazer um export na senha de email em nosso shell.

### superlists/settings.py (ch17I049)

```
EMAIL_HOST = 'smtp.gmail.com'
```

```
EMAIL_HOST_USER = 'obeythetestinggoat@gmail.com'
```

```
EMAIL_HOST_PASSWORD = os.environ.get('EMAIL_PASSWORD')
```

```
EMAIL_PORT = 587
```

```
EMAIL_USE_TLS = True
```

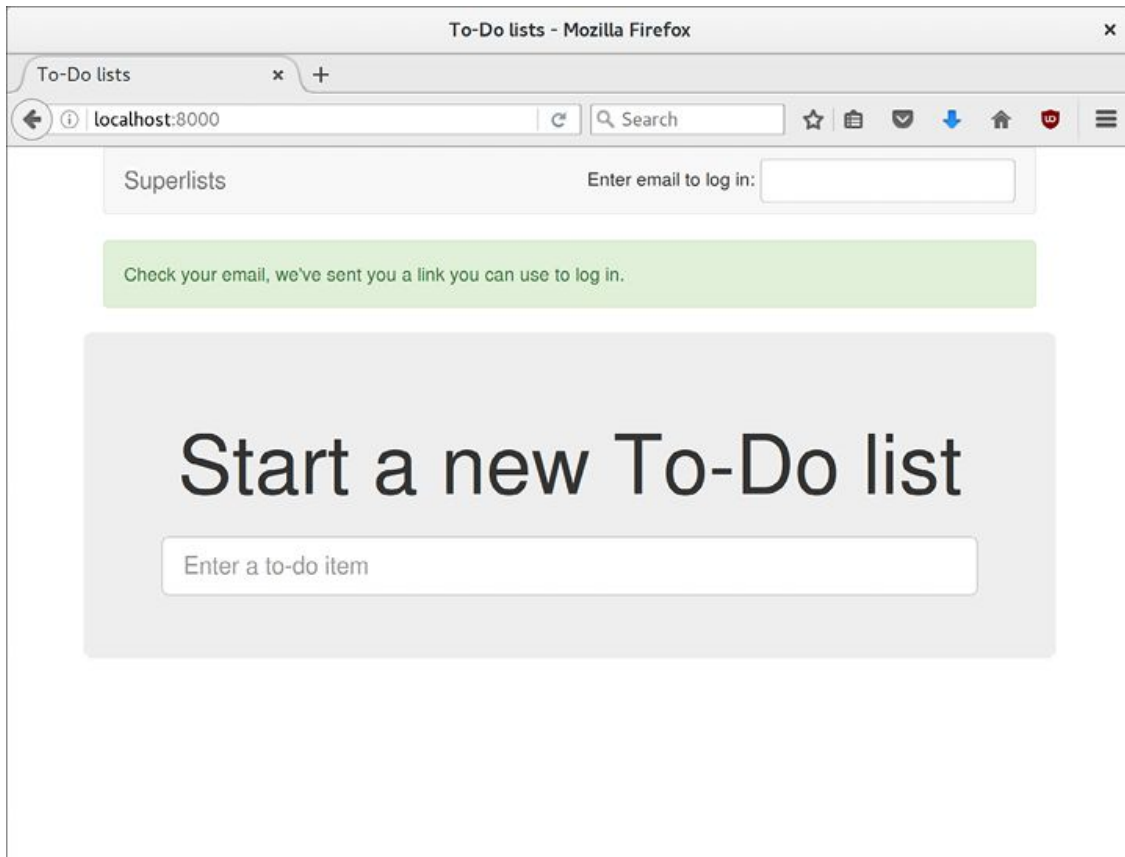
e

```
$ export EMAIL_PASSWORD="sekrit"
```

```
$ python manage.py runserver
```

Então você deverá ver algo semelhante ao que está sendo

mostrado na Figura 19.1.



*Figura 19.1 – Verifique o seu email...*

Uh-huu!

Fiquei esperando até agora para fazer um commit, somente para me certificar de que tudo está funcionando. Nesse ponto, você poderia fazer uma série de commits separados – um para a view de login, outro para o backend de autenticação, outro para o modelo de usuário e mais um para a associação do template. Ou poderia muito bem decidir que, como tudo está inter-relacionado e nenhuma mudança funcionará sem as demais, faria apenas um grande commit:

```
$ git status
$ git add .
$ git diff --staged
$ git commit -m "Custom passwordless auth backend + custom user model"
```

## Terminando o nosso FT, testando o logout

A última tarefa que precisamos fazer antes de encerrarmos o expediente é testar o link de logout. Vamos estender o FT com mais alguns passos:

### functional\_tests/test\_login.py (ch17I050)

```
[...]
Ela está logada!
self.wait_for(
 lambda: self.browser.find_element_by_link_text('Log out')
)
navbar = self.browser.find_element_by_css_selector('.navbar')
self.assertIn(TEST_EMAIL, navbar.text)

Agora ela faz logout
self.browser.find_element_by_link_text('Log out').click()

Ela não está mais logada!
self.wait_for(
 lambda: self.browser.find_element_by_name('email')
)
navbar = self.browser.find_element_by_css_selector('.navbar')
self.assertNotIn(TEST_EMAIL, navbar.text)
```

Com isso, podemos ver que o teste está falhando porque o botão de logout não funciona:

```
$ python manage.py test functional_tests.test_login
```

```
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate
element: [name="email"]
```

Implementar um botão de logout, na verdade, é bem simples: podemos usar a view de logout embutida do Django (<http://bit.ly/SuI0hA>), que limpa a sessão do usuário e o redireciona para uma página de nossa preferência:

### accounts/urls.py (ch17I051)

```
from django.contrib.auth.views import logout
```



[...]

```
urlpatterns = [
 url(r'^send_login_email$', views.send_login_email, name='send_login_email'),
 url(r'^login$', views.login, name='login'),
 url(r'^logout$', logout, {'next_page': '/'}, name='logout'),
]
```

Em *base.html*, simplesmente transformamos o logout em um verdadeiro link de URL:

## lists/templates/base.html (ch17I052)

```
Log out
```

Com isso, o FT passa por completo – na verdade, temos uma suíte de testes passando por inteiro:

```
$ python manage.py test functional_tests.test_login
```

```
[...]
```

```
OK
```

```
$ python manage.py test
```

```
[...]
```

```
Ran 59 tests in 78.124s
```

```
OK
```



Não estamos nem próximos de um sistema de login verdadeiramente seguro ou aceitável, nesse caso. Como essa é apenas uma aplicação de exemplo para um livro, deixaremos isso como está, mas, na “vida real”, você deve explorar muitos outros problemas de segurança e de usabilidade antes de dizer que o trabalho está pronto. Estamos perigosamente próximos de “implementar a nossa própria criptografia” nesse caso, e contar com um sistema de login mais consagrado seria muito mais seguro.

No próximo capítulo, começaremos a fazer bom uso de nosso sistema de login. Nesse ínterim, faça um commit e aprecie a revisão a seguir:

## Sobre simulação (mocking) em Python

## *Simulação e dependências externas*

Usamos a simulação em testes de unidade quando temos uma dependência externa que não queremos realmente utilizar em nossos testes. Um mock é usado para simular uma API de terceiros. Embora seja possível “desenvolver seus próprios” mocks em Python, um framework de simulação, como o módulo mock, oferece muitos atalhos convenientes, que facilitarão escrever (e, acima de tudo, ler) seus testes.

## *Monkeypatching*

Consiste em substituir um objeto em um namespace em tempo de execução. É usado em nossos testes de unidade para substituir uma função real que tenha efeitos colaterais indesejados por um objeto mock, usando o decorador patch.

## *A biblioteca Mock*

Michael Foord (que costumava trabalhar na empresa que deu origem à PythonAnywhere, pouco antes de eu ter me juntado a ela) escreveu a excelente biblioteca “Mock”, que atualmente está integrada na biblioteca-padrão do Python 3. Ela contém praticamente tudo que você poderá precisar para simulação em Python.

## *O decorador patch*

`unittest.mock` disponibiliza uma função chamada `patch`, que pode ser usada para “simular” qualquer objeto do módulo que você estiver testando. É comumente utilizado como um decorador em um método de teste, ou até no nível de classe, quando é aplicado a todos os métodos de teste dessa classe.

## *Mocks podem deixar você altamente acoplado com a implementação*

Como vimos na caixa de texto “Mocks podem deixar você altamente acoplado com a implementação”, os mocks podem deixar você altamente acoplado com a sua implementação. Por esse motivo, você não deve usá-los, a menos que tenha um bom motivo.

## *Mocks podem evitar duplicação em seus testes*

Por outro lado, não faz sentido duplicar todos os seus testes de uma função em uma porção de código de nível mais alto que utilize essa função. Nesse caso, usar um mock reduzirá a duplicação.

Há várias outras discussões sobre os prós e contras dos mocks posteriormente. Continue lendo!

- 
- 1 Estou usando o termo genérico “mock”, mas os entusiastas de testes gostam de fazer a distinção com outros tipos de uma classe genérica de ferramentas de teste chamadas “Test Doubles” (Dublês de teste), que incluem spies, fakes e stubs. As diferenças não são realmente importantes para este livro, mas, se quiser conhecer os detalhes, consulte esta incrível wiki de Justin Searls (<https://github.com/testdouble/contributing-tests/wiki/Test-Double>). Aviso: o site está absolutamente repleto de ótimo conteúdo sobre testes.
  - 2 Sim, eu sei que o Django já simula emails para nós por meio de mail.outbox, mas, novamente, vamos fingir que ela não existe. E se você estivesse usando Flask? Ou se essa fosse uma chamada de API, e não um email?
  - 3 No Python 2, você pode instalá-lo com `pip install mock`.
  - 4 Separei a tag de formulário em três linhas para que caibam de forma elegante no livro. Caso ainda não tenha visto isso antes, esse HTML poderá parecer um pouco estranho para você, mas é válido. Contudo, não é necessário usá-lo se você não gostar. :)





## CAPÍTULO 20

# Fixtures de teste e um decorador para esperas explícitas

Agora que temos um sistema de autenticação funcional, queremos usá-lo para identificar usuários e poder lhes mostrar todas as listas que criaram.

Para isso, precisaremos escrever FTs que tenham um usuário logado. Em vez de fazer com que cada teste passe pela dança do email de login (que consome tempo), queremos que seja possível pular essa parte.

Isso diz respeito à separação de responsabilidades. Os testes funcionais não são como os testes de unidade, pois em geral eles não têm uma única asserção. Conceitualmente, porém, eles deveriam testar um único ponto. Não há a necessidade de cada FT individual testar os mecanismos de login/logout. Se descobrirmos uma forma de “trapacear” e pular essa parte, gastaremos menos tempo esperando pela execução de caminhos de teste duplicados.



Não exagere na remoção de duplicação nos FTs. Uma das vantagens de um FT é a sua capacidade de identificar interações estranhas e imprevisíveis entre diferentes partes de sua aplicação.



Este capítulo acabou de ser reescrito para a nova edição, portanto me avise, usando [obeythetestinggoat@gmail.com](mailto:obeythetestinggoat@gmail.com), se você vir algum problema

ou se tiver alguma sugestão para melhorias!

## Pulando o processo de login e criando previamente uma sessão

É bem comum para um usuário retornar a um site e ainda ter um cookie, o que significa que ele estará “previamente autenticado”, portanto essa não é uma trapaça totalmente fora da realidade. Eis o modo como você pode configurar isso:

### functional\_tests/test\_my\_lists.py

```
from django.conf import settings
from django.contrib.auth import BACKEND_SESSION_KEY, SESSION_KEY,
get_user_model
from django.contrib.sessions.backends.db import SessionStore
from .base import FunctionalTest
User = get_user_model()
```

```
class MyListsTest(FunctionalTest):
```

```
 def create_pre_authenticated_session(self, email):
 user = User.objects.create(email=email)
 session = SessionStore()
 session[SESSION_KEY] = user.pk ❶
 session[BACKEND_SESSION_KEY] =
settings.AUTHENTICATION_BACKENDS[0]
 session.save()
 ## para definir um cookie, precisamos antes acessar o domínio.
 ## as páginas 404 são as que carregam mais rapidamente!
 self.browser.get(self.live_server_url + "/404_no_such_url/")
 self.browser.add_cookie(dict(
 name=settings.SESSION_COOKIE_NAME,
 value=session.session_key, ❷
 path='/',
))
```

- ❶ Criamos um objeto de sessão no banco de dados. A chave da sessão é a chave primária do objeto usuário (que, na verdade, é o endereço de email do usuário).

- 2 Então adicionamos um cookie no navegador, que corresponda à sessão no servidor – em nossa próxima visita ao site, o servidor deverá nos reconhecer como um usuário logado.

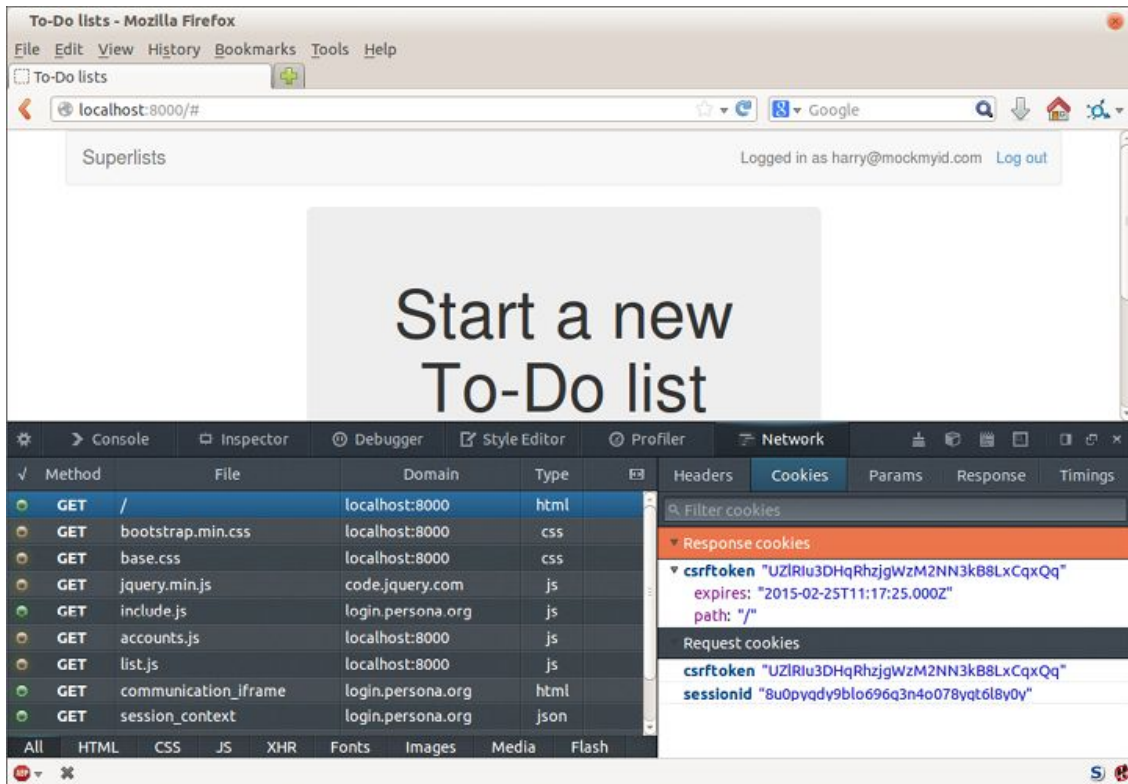
Observe que, do modo como está, esse código só funcionará porque estamos usando LiveServerTestCase, portanto os objetos User e Session que criamos acabarão no mesmo banco de dados que o servidor de testes. Mais tarde precisaremos modificar esse código para que funcione também no banco de dados do servidor de staging.

## **Sessões do Django: como os cookies de um usuário informam ao servidor que ele está autenticado**

*Essa é uma tentativa de explicar sessões, cookies e autenticação no Django.*

Como o HTTP não tem estado (é stateless), os servidores precisam ter uma maneira de reconhecer diferentes clientes a *cada requisição*. Endereços IP podem ser compartilhados, portanto a solução costumeira é dar a cada cliente um ID de sessão único, que será armazenado em um cookie e submetido a cada requisição. O servidor armazenará esse ID em algum lugar (por padrão, no banco de dados) e, então, poderá reconhecer cada requisição que chegar como sendo de um cliente em particular.

Se você fizer login no site usando o servidor de desenvolvimento, poderá ver seu ID de sessão manualmente, se quiser. Por padrão, ele está armazenado com a chave sessionid. Veja a Figura 20.1.



*Figura 20.1 – Analisando o cookie de sessão na barra de ferramentas de Debug.*

Esses cookies de sessão são definidos para todos os visitantes de um site do Django, estejam eles logados ou não.

Quando queremos reconhecer um cliente como um usuário logado e autenticado, novamente, em vez de pedir ao cliente que envie seu nome de usuário e a senha a cada requisição, o servidor pode simplesmente marcar a sessão desse cliente como uma sessão autenticada e associá-la a um ID de usuário em seu banco de dados.

Uma sessão é uma estrutura de dados semelhante a um dicionário, e o ID do usuário é armazenado com a chave dada por `django.contrib.auth.SESSION_KEY`. Você pode conferir isso em um `./manage.py shell`, se quiser:

```
$ python manage.py shell
```

```
[...]
```

```
In [1]: from django.contrib.sessions.models import Session
```

```
substitua o seu id de sessão do cookie de seu navegador aqui
```

```
In [2]: session = Session.objects.get(
 session_key="8u0pygdy9blo696g3n4o078ygt6l8y0y"
```

)

```
In [3]: print(session.get_decoded())
{'_auth_user_id': 'obeythetestinggoat@gmail.com', '_auth_user_backend':
'accounts.authentication.PasswordlessAuthenticationBackend'}
```

Você também pode armazenar qualquer outra informação que quiser na sessão de um usuário, como forma de manter o controle temporariamente sobre algum estado. Isso funciona para usuários que não estejam logados também. Basta usar `request.session` em qualquer view, e ela funcionará como um dicionário. Há mais informações na documentação do Django sobre sessões (<http://bit.ly/2tGVbQE>).

## Verificando se o código funciona

Para verificar se o código funciona, seria bom usar parte do código de nosso teste anterior. Vamos criar duas funções de nomes `wait_to_be_logged_in` e `wait_to_be_logged_out`. Para acessá-las a partir de um teste diferente, devemos inseri-las em `FunctionalTest`. Também as adaptaremos levemente para que aceitem um endereço de email arbitrário como parâmetro:

### functional\_tests/base.py (ch18I002)

```
class FunctionalTest(StaticLiveServerTestCase):
 [...]

 def wait_to_be_logged_in(self, email):
 self.wait_for(
 lambda: self.browser.find_element_by_link_text('Log out')
)
 navbar = self.browser.find_element_by_css_selector('.navbar')
 self.assertIn(email, navbar.text)

 def wait_to_be_logged_out(self, email):
 self.wait_for(
 lambda: self.browser.find_element_by_name('email')
)
 navbar = self.browser.find_element_by_css_selector('.navbar')
```

```
self.assertNotIn(email, navbar.text)
```

Hum, nada mal, mas não estou muito satisfeito com o volume de duplicações de código associado a `wait_for` nesse caso. Vamos fazer uma anotação para lembrar de retomar isso e fazer esses auxiliares funcionarem.

- **Limpar o código associado a `wait_for` em `base.py`**

Inicialmente vamos usá-los em `test_login.py`:

## functional\_tests/test\_login.py (ch18l003)

```
def test_can_get_email_link_to_log_in(self):
 [...]
 # Ela está logada!
 self.wait_to_be_logged_in(email=TEST_EMAIL)

 # Agora ela faz logout
 self.browser.find_element_by_link_text('Log out').click()

 # Ela não está mais logada!
 self.wait_to_be_logged_out(email=TEST_EMAIL)
```

Somente para garantir que não causamos nenhum outro problema, vamos executar o teste de login de novo:

```
$ python manage.py test functional_tests.test_login
[...]
OK
```

Agora podemos escrever um placeholder para o teste de “My Lists” (Minhas Listas) e ver se o nosso criador de sessão autenticada realmente funciona:

## functional\_tests/test\_my\_lists.py (ch18l004)

```
def test_logged_in_users_lists_are_saved_as_my_lists(self):
 email = 'edith@example.com'
 self.browser.get(self.live_server_url)
 self.wait_to_be_logged_out(email)

 # Edith é uma usuária logada
```

```
self.create_pre_authenticated_session(email)
self.browser.get(self.live_server_url)
self.wait_to_be_logged_in(email)
```

Eis o resultado:

```
$ python manage.py test functional_tests.test_my_lists
[...]
OK
```

É uma boa hora para um commit:

```
$ git add functional_tests
$ git commit -m "test_my_lists: precreate sessions, move login checks into
base"
```

## Fixtures de teste de JSON são consideradas prejudiciais

Quando preenchemos previamente o banco de dados com dados de teste, como fizemos nesse caso com o objeto User e seu objeto Session associado, o que estamos fazendo é configurando uma “fixture de teste”.

O Django vem com suporte embutido para salvar objetos no banco de dados como JSON (usando `manage.py dumpdata`), e carregá-los automaticamente em suas execuções de teste usando o atributo de classe `fixtures` em `TestCase`.

Cada vez mais pessoas estão começando a dizer: não use fixtures JSON (<http://bit.ly/1kSTyrb>). Elas são um pesadelo de manutenção quando seus modelos mudam. Além disso, é difícil para o leitor dizer quais dos diversos valores de atributo especificados no JSON são cruciais para o comportamento em teste, e quais servem apenas para preenchimento. Por fim, mesmo que os testes comecem compartilhando fixtures, cedo ou tarde um teste vai querer versões um pouco diferentes dos dados, e você acabará copiando tudo para mantê-los isolados; mais uma vez, será difícil dizer o que é relevante para o teste e o que apenas está lá por acaso.

Em geral, será muito mais simples carregar os dados diretamente usando o ORM do Django.



Depois que você tiver mais que um punhado de campos em um modelo,

e/ou vários modelos relacionados, até mesmo o uso de ORM poderá ser inconveniente. Nesse caso há uma ferramenta chamada `factory_boy` (<https://factoryboy.readthedocs.org/>), que muitas pessoas juram ser boa.

## Nossa função auxiliar final de espera explícita: um decorador `wait`

Até agora usamos decoradores algumas vezes em nosso código, mas é hora de saber como eles realmente funcionam, criando o nosso próprio decorador.

Inicialmente vamos pensar em como queremos que o nosso decorador funcione. Seria bom se pudéssemos substituir toda a lógica personalizada de espera/retentativa/timeout em `wait_for_row_in_list_table` e o inline `self.wait_for` em `wait_to_be_logged_in/out`. Algo como o código a seguir seria adorável:

### `functional_tests/base.py` (ch18l005)

```
@wait
def wait_for_row_in_list_table(self, row_text):
 table = self.browser.find_element_by_id('id_list_table')
 rows = table.find_elements_by_tag_name('tr')
 self.assertIn(row_text, [row.text for row in rows])

@wait
def wait_to_be_logged_in(self, email):
 self.browser.find_element_by_link_text('Log out')
 navbar = self.browser.find_element_by_css_selector('.navbar')
 self.assertIn(email, navbar.text)

@wait
def wait_to_be_logged_out(self, email):
 self.browser.find_element_by_name('email')
 navbar = self.browser.find_element_by_css_selector('.navbar')
 self.assertNotIn(email, navbar.text)
```

Você está pronto para um mergulho de cabeça? Embora os



decoradores sejam bem difíceis de compreender (sei que precisei de um bom tempo até me sentir à vontade com eles, e ainda preciso pensar com muito cuidado sempre que crio um), o interessante é que já mergulhamos os dedos dos pés na programação funcional, com a nossa função auxiliar `self.wait_for`. Essa é uma função que aceita outra função como argumento, e o mesmo vale para um decorador. A diferença é que o decorador não executa realmente nenhum código por si só – ele devolve uma versão modificada da função que recebe.

Nosso decorador devolve uma nova função que ficará chamando a função recebida, capturando nossas exceções usuais, até que um timeout ocorra. Eis uma primeira tentativa:

## functional\_tests/base.py (ch18I006)

```
def wait(fn): ❶
 def modified_fn(): ❸
 start_time = time.time()
 while True: ❹
 try:
 return fn() ❺
 except (AssertionError, WebDriverException) as e: ❹
 if time.time() - start_time > MAX_WAIT:
 raise e
 time.sleep(0.5)
 return modified_fn ❷
```

- ❶ Um decorador é uma forma de modificar uma função; ele aceita uma função como argumento...
- ❷ e devolve outra função como a versão modificada (ou “decorada”).
- ❸ Eis o local em que criamos a nossa função modificada.
- ❹ E eis o nosso laço familiar, que continuará executando, capturando as exceções usuais, até o nosso timeout expirar.
- ❺ Como sempre, chamamos a nossa função e retornamos imediatamente se não houver nenhuma exceção.

Está *quase* tudo certo, mas nem tanto; vamos tentar executar?

```
$ python manage.py test functional_tests.test_my_lists
```

```
[...]
```

```
self.wait_to_be_logged_out(email)
```

```
TypeError: modified_fn() takes 0 positional arguments but 2 were given
```

De modo diferente de `self.wait_for`, o decorador está sendo aplicado em funções com argumentos:

## functional\_tests/base.py

```
@wait
```

```
def wait_to_be_logged_in(self, email):
```

```
 self.browser.find_element_by_link_text('Log out')
```

`wait_to_be_logged_in` aceita `self` e `email` como argumentos posicionais. Contudo, quando está decorada, ela é substituída por `modified_fn`, que não aceita nenhum argumento. Como fazer, como num passe de mágica, para que a nossa `modified_fn` trate os mesmos argumentos que qualquer `fn` recebido pelo decorador possa ter?

A resposta está em usar um pouco da mágica de Python, `*args` e `**kwargs`, mais formalmente conhecidos como “argumentos variádicos”

(<https://docs.python.org/3/tutorial/controlflow.html#keywordarguments>), aparentemente (acabei de aprender):

## functional\_tests/base.py (ch18I007)

```
def wait(fn):
```

```
 def modified_fn(*args, **kwargs): ❶
```

```
 start_time = time.time()
```

```
 while True:
```

```
 try:
```

```
 return fn(*args, **kwargs) ❷
```

```
 except (AssertionError, WebDriverException) as e:
```

```
 if time.time() - start_time > MAX_WAIT:
```

```
 raise e
```

```
 time.sleep(0.5)
```

```
 return modified_fn
```

❶ Ao usar `*args` e `**kwargs`, especificamos que `modified_fn` pode aceitar

quaisquer argumentos posicionais e nomeados arbitrários.

- ② Como os capturamos na definição da função, garantimos que passaremos esses mesmos argumentos para `fn` quando a chamamos.

Um dos cenários interessantes em que isso pode ser usado é quando criamos um decorador que altere os argumentos de uma função. Todavia não entraremos nesse assunto agora. O ponto principal é que o nosso decorador agora funciona:

```
$ python manage.py test functional_tests.test_my_lists
[...]
OK
```

Você sabe o que é verdadeiramente satisfatório? É o fato de podermos usar o nosso decorador `wait` em nossa função auxiliar `self.wait_for` também! Assim:

**functional\_tests/base.py (ch18|008)**

```
@wait
def wait_for(self, fn):
 return fn()
```

Maravilhoso! Agora toda a nossa lógica de espera/retentativa está encapsulada em um só lugar, e temos uma maneira bela e rápida de aplicar essas esperas, seja inline em nossos FTs usando `self.wait_for`, ou em qualquer função auxiliar que utilize o decorador `@wait`.

No próximo capítulo, tentaremos fazer a implantação de nosso código no servidor de staging e usaremos as fixtures de sessão previamente autenticada no servidor. Como veremos, isso nos ajudará a identificar um ou dois pequenos bugs!

## Lições aprendidas

## *Decoradores são bons*

Os decoradores podem ser uma ótima maneira de abstrair diferentes níveis de responsabilidade. Eles nos permitem escrever nossas asserções de teste sem ter que pensar nas esperas ao mesmo tempo.

### *Remova a duplicação em seus FTs, com cuidado*

Cada FT individual não precisa testar todas as partes de sua aplicação. Em nosso caso, queríamos evitar passar por todo o processo de login em cada FT que precisasse de um usuário autenticado, portanto utilizamos uma fixture de teste para “trapacear” e pular essa parte. Talvez você ache outras tarefas que queira pular em seus FTs. Eis uma advertência, porém: os testes funcionais existem para capturar interações imprevisíveis entre diferentes partes de sua aplicação, portanto tome cuidado para não forçar a remoção de duplicações ao extremo.

## *Fixtures de teste*

Fixtures de testes referem-se a dados de teste que devem ser configurados como uma pré-condição antes de um teste ser executado – com frequência, isso significa preencher o banco de dados com algumas informações, mas, conforme vimos (nos cookies de navegador), pode envolver outros tipos de pré-condições.

## *Evite fixtures de JSON*

O Django facilita salvar e restaurar dados do banco de dados em formato JSON (e em outros) usando os comandos de gerenciamento `dumpdata` e `loaddata`. A maioria das pessoas não recomenda usá-los para fixtures de testes, pois são difíceis de administrar quando o esquema de seu banco de dados muda. Utilize o ORM ou uma ferramenta como o `factory_boy` (<https://factoryboy.readthedocs.org/>).





# CAPÍTULO 21

## Depuração no lado do servidor

Vamos retirar algumas camadas da pilha de atividades em que estamos trabalhando: temos boas funções auxiliares para espera; para que as estávamos usando? Ah, sim, estávamos esperando para fazer login. E por que fizemos isso? Ah, sim, tínhamos acabado de implementar uma maneira de pré-autenticar um usuário.

### Prova definitiva: usando o servidor de staging para capturar os últimos bugs

Todo o código está bom para executar os FTs localmente, mas como ele se sairá no servidor de staging? Vamos tentar fazer a implantação de nosso site. Nesse processo, capturaremos um bug inesperado (é para isso que serve o staging!) e, então, teremos que descobrir uma forma de administrar o banco de dados no servidor de testes:

```
$ cd deploy_tools
$ fab deploy --host=elspeth@superlists-staging.ottg.eu
[...]
```

Reinicie o Gunicorn...

```
elspeth@server:$ sudo systemctl daemon-reload
elspeth@server:$ sudo systemctl restart gunicorn-superlists-
staging.ottg.eu
```

Eis o que acontece quando executamos os testes funcionais:

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test
functional_tests
```

=====

=====

ERROR: test\_can\_get\_email\_link\_to\_log\_in  
(functional\_tests.test\_login.LoginTest)

-----

Traceback (most recent call last):

File "../functional\_tests/test\_login.py", line 22, in  
test\_can\_get\_email\_link\_to\_log\_in  
self.assertIn('Check your email', body.text)

AssertionError: 'Check your email' not found in 'Server Error (500)'

=====

=====

ERROR: test\_logged\_in\_users\_lists\_are\_saved\_as\_my\_lists  
(functional\_tests.test\_my\_lists.MyListsTest)

-----

Traceback (most recent call last):

File "/home/harry/./book-example/functional\_tests/test\_my\_lists.py",  
line 34, in test\_logged\_in\_users\_lists\_are\_saved\_as\_my\_lists  
self.wait\_to\_be\_logged\_in(email)

File "/worskpace/functional\_tests/base.py", line 42, in wait\_to\_be\_logged\_in  
self.browser.find\_element\_by\_link\_text('Log out')

[...]

selenium.common.exceptions.NoSuchElementException: Message: Unable to locate

element: {"method": "link text", "selector": "Log out"}

Stacktrace:

[...]

-----

Ran 8 tests in 27.933s

FAILED (errors=2)

Não conseguimos fazer login – seja com o sistema de emails real, seja com a nossa sessão previamente autenticada. Parece que o nosso belo sistema novo de autenticação está causando falhas no servidor.

Vamos colocar em prática um pouco de depuração no lado do servidor!

## Configurando o logging

Para rastrear esse problema, temos que configurar o Gunicorn para que faça um pouco de logging. Ajuste a configuração do Gunicorn no servidor usando vi ou nano:

```
server: /etc/systemd/system/gunicorn-superlists-
staging.ottg.eu.service
```

```
ExecStart=/home/elspeth/sites/superlists-staging.ottg.eu/virtualenv/bin/gunicorn \
--bind unix:/tmp/superlists-staging.ottg.eu.socket \
--capture-output \
--access-logfile ../access.log \
--error-logfile ../error.log \
superlists.wsgi:application
```

Isso fará com que um log de acesso e um log de erro sejam colocados na pasta `~/sites/$SITENAME`.

Certifique-se também de que o seu `settings.py` ainda contenha as configurações de LOGGING, que farão com que os dados sejam realmente enviados para o console:

### superlists/settings.py

```
LOGGING = {
 'version': 1,
 'disable_existing_loggers': False,
 'handlers': {
 'console': {
 'level': 'DEBUG',
 'class': 'logging.StreamHandler',
 },
 },
 'loggers': {
 'django': {
 'handlers': ['console'],
 },
 },
 'root': {'level': 'INFO'},
}
```

Reiniciamos mais uma vez o Gunicorn e, em seguida, executamos novamente o FT, ou tentamos fazer login manualmente. Enquanto isso ocorre, podemos observar os logs no servidor usando:

```
elspeth@server:$ sudo systemctl daemon-reload
elspeth@server:$ sudo systemctl restart gunicorn-superlists-
staging.ottg.eu
elspeth@server:$ tail -f error.log # supõe que estamos na pasta
~/sites/$SITENAME
```

Você deverá ver um erro como este:

```
Internal Server Error: /accounts/send_login_email
Traceback (most recent call last):
 File "/home/elspeth/sites/superlists-staging.ottg.eu/virtualenv/lib/python3.6/[...]
 response = wrapped_callback(request, *callback_args, **callback_kwargs)
 File
"/home/elspeth/sites/superlists-staging.ottg.eu/source/accounts/views.py", line
20, in send_login_email
 [email]
[...]
 self.connection.sendmail(from_email, recipients,
message.as_bytes(linesep='\r\n'))
 File "/usr/lib/python3.6/smtplib.py", line 862, in sendmail
 raise SMTPSenderRefused(code, resp, from_addr)
smtplib.SMTPSenderRefused: (530, b'5.5.1 Authentication Required. Learn
more
at\n5.5.1 https://support.google.com/mail/?p=WantAuthError [...]
- smtp', 'noreply@superlists')
```

Hum, o Gmail está se recusando a enviar os nossos emails, não é mesmo? Mas por que isso acontece? Ah, sim, não informamos qual é a nossa senha ao servidor!

## Definindo variáveis de ambiente secretas no servidor

No Capítulo 11, vimos uma maneira de definir valores secretos no servidor; usamos isso para preencher a configuração de SECRET\_KEY do Django – criamos arquivos Python uma só vez no sistema de arquivos do servidor e os importamos.

Nesses capítulos, usamos variáveis de ambiente em nossos shells para armazenar a senha de nosso email, portanto vamos fazer o mesmo no servidor. Podemos definir a variável de ambiente no arquivo de configuração de Systemd:

```
server: /etc/systemd/system/gunicorn-superlists-
staging.ottg.eu.service
```

```
[Service]
User=elspeth
Environment=EMAIL_PASSWORD=yoursekritpasswordhere
WorkingDirectory=/home/elspeth/sites/superlists-staging.ottg.eu/source
[...]
```



Uma vantagem de segurança a favor da qual podemos argumentar quando usamos esse arquivo de configuração é o fato de podermos restringir suas permissões para que ele seja legível somente pelo root – algo que não podemos fazer para os arquivos-fontes Python de nossa aplicação.

Ao salvar esse arquivo e fazer a dança usual com `daemon-reload` e `restart gunicorn`, podemos executar os FTs novamente, e...

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test
functional_tests
```

```
[...]
```

```
Traceback (most recent call last):
```

```
File ".../superlists/functional_tests/test_login.py", line 25, in
test_can_get_email_link_to_log_in
 email = mail.outbox[0]
```

```
IndexError: list index out of range
```

```
[...]
```

```
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate
```

```
element: {"method": "link text", "selector": "Log out"}
```

A falha em `my_lists` continua a mesma, porém temos mais informações em nosso teste de login: o FT avança e o site agora parece estar enviando emails corretamente (e o log do servidor não

mostra nenhum erro), mas não podemos verificar o email no mail.outbox...

## Adaptando o nosso FT para que possamos testar emails de verdade via POP3

Ah. Isso explica tudo. Agora que estamos executando em um servidor de verdade, e não no LiveServerTestCase, não podemos mais inspecionar o django.mail.outbox local para ver os emails enviados.

Em primeiro lugar, precisaremos saber se estamos executando no servidor de staging ou em nossos FTs. Vamos salvar a variável `staging_server` em `self` no arquivo `base.py`:

### functional\_tests/base.py (ch18I009)

```
def setUp(self):
 self.browser = webdriver.Firefox()
 self.staging_server = os.environ.get('STAGING_SERVER')
 if self.staging_server:
 self.live_server_url = 'http://' + self.staging_server
```

Então implementamos uma função auxiliar capaz de obter um email real de um verdadeiro servidor de email POP3, usando o cliente POP3 da biblioteca-padrão de Python, terrivelmente sinuoso:

### functional\_tests/test\_login.py (ch18I010)

```
import os
import poplib
import re
import time
[...]

def wait_for_email(self, test_email, subject):
 if not self.staging_server:
 email = mail.outbox[0]
 self.assertIn(test_email, email.to)
 self.assertEqual(email.subject, subject)
 return email.body
```

```

email_id = None
start = time.time()
inbox = poplib.POP3_SSL('pop.mail.yahoo.com')
try:
 inbox.user(test_email)
 inbox.pass_(os.environ['YAHOO_PASSWORD'])
 while time.time() - start < 60:
 # obtém as 10 mensagens mais recentes
 count, _ = inbox.stat()
 for i in reversed(range(max(1, count - 10), count + 1)):
 print('getting msg', i)
 _, lines, __ = inbox.retr(i)
 lines = [l.decode('utf8') for l in lines]
 print(lines)
 if f'Subject: {subject}' in lines:
 email_id = i
 body = '\n'.join(lines)
 return body
 time.sleep(5)
finally:
 if email_id:
 inbox.dele(email_id)
 inbox.quit()

```



Estou usando uma conta do Yahoo para testes, mas você pode utilizar qualquer serviço de email que quiser, desde que ele ofereça acesso via POP3. Será necessário definir a variável de ambiente YAHOO\_PASSWORD no console que estiver executando o FT.

Então, como resultado, fazemos o restante das alterações necessárias no FT. Inicialmente preenchemos uma variável test\_email, que será diferente para testes locais e testes no staging:

## functional\_tests/test\_login.py (ch18|011-1)

```
@@ -7,7 +7,7 @@ from selenium.webdriver.common.keys import Keys
```

```
from .base import FunctionalTest
```

```
-TEST_EMAIL = 'edith@example.com'
```

```
+
```

```
SUBJECT = 'Your login link for Superlists'
```

```
@@ -33,7 +33,6 @@ class LoginTest(FunctionalTest):
 print('getting msg', i)
 __, lines, __ = inbox.retr(i)
 lines = [l.decode('utf8') for l in lines]
- print(lines)
 if f'Subject: {subject}' in lines:
 email_id = i
 body = '\n'.join(lines)
@@ -49,6 +48,11 @@ class LoginTest(FunctionalTest):
 # Edith acessa o incrível site de superlistas e, pela primeira vez,
 # percebe que há uma seção de "Log in" na barra de navegação
 # Essa seção está lhe dizendo para inserir o seu endereço de email,
 # portanto ela faz isso
+ if self.staging_server:
+ test_email = 'edith.testuser@yahoo.com'
+ else:
+ test_email = 'edith@example.com'
+
 self.browser.get(self.live_server_url)
```

E então fazemos modificações que envolvem o uso dessa variável e a chamada de nossa nova função auxiliar:

## functional\_tests/test\_login.py (ch18|011-2)

```
@@ -54,7 +54,7 @@ class LoginTest(FunctionalTest):
 test_email = 'edith@example.com'

 self.browser.get(self.live_server_url)
- self.browser.find_element_by_name('email').send_keys(TEST_EMAIL)
+ self.browser.find_element_by_name('email').send_keys(test_email)
 self.browser.find_element_by_name('email').send_keys(Keys.ENTER)

 # Uma mensagem aparece informando-lhe que um email foi enviado
@@ -64,15 +64,13 @@ class LoginTest(FunctionalTest):
))

 # Ela verifica seu email e encontra uma mensagem
- email = mail.outbox[0]
```



```

- self.assertEqual(email.subject, SUBJECT)
+ body = self.wait_for_email(test_email, SUBJECT)

 # A mensagem contém um link com um url
- self.assertIn('Use this link to log in', email.body)
- url_search = re.search(r'http://.+/$', email.body)
+ self.assertIn('Use this link to log in', body)
+ url_search = re.search(r'http://.+/$', body)
 if not url_search:
- self.fail(f'Could not find url in email body:\n{email.body}')
+ self.fail(f'Could not find url in email body:\n{body}')
 url = url_search.group(0)
 self.assertEqual(self.live_server_url, url)

@@ -80,11 +78,11 @@ class LoginTest(FunctionalTest):
 self.browser.get(url)

 # Ela está logada!
- self.wait_to_be_logged_in(email=TEST_EMAIL)
+ self.wait_to_be_logged_in(email=test_email)

 # Agora ela faz logout
 self.browser.find_element_by_link_text('Log out').click()

 # Ela não está mais logada!
- self.wait_to_be_logged_out(email=TEST_EMAIL)
+ self.wait_to_be_logged_out(email=test_email)

```

Acredite ou não, isso realmente funcionará e nos dará um FT de fato capaz de verificar se os logins funcionam, envolvendo emails de verdade!



Acabei de compor esse código de verificação de emails e, no momento, ele está bem deselegante e frágil (um problema comum é obter o email incorreto, proveniente de uma execução anterior do teste). Com um pouco de limpeza e mais alguns laços de retentativas, o código poderá evoluir para algo mais confiável. De modo alternativo, serviços como *mailinator.com* lhe darão endereços de email descartáveis e uma API para conferi-los, por um preço baixo.

## Administrando o banco de dados de testes no ambiente de staging

Agora podemos executar novamente os nossos FTs e obter a próxima falha: nossa tentativa de criar sessões previamente autenticadas não funciona, de modo que o teste de “My Lists” (Minhas Listas) falha:

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test functional_tests
```

```
ERROR: test_logged_in_users_lists_are_saved_as_my_lists (functional_tests.test_my_lists.MyListsTest)
```

```
[...]
```

```
selenium.common.exceptions.TimeoutException: Message: Could not find element
```

```
with id id_logout. Page text was:
```

```
Superlists
```

```
Sign in
```

```
Start a new To-Do list
```

```
Ran 8 tests in 72.742s
```

```
FAILED (errors=1)
```

Isso ocorre porque nossa função utilitária de teste `create_pre_authenticated_session` atua somente no banco de dados local. Vamos descobrir como os nossos testes poderão administrar o banco de dados no servidor.

## Um comando de gerenciamento do Django para criar sessões

Para executar algo no servidor, precisaremos implementar um script autocontido que possa ser executado a partir da linha de comando no servidor, provavelmente usando o Fabric.

Ao tentar implementar um script independente, que funcione com o Django (isto é, que seja capaz de conversar com o banco de dados, e assim por diante), há alguns problemas inoportunos que devem ser resolvidos de forma apropriada, por exemplo, definir a variável

de ambiente `DJANGO_SETTINGS_MODULE`, e configurar `sys.path` corretamente.

Em vez de ficar lidando com tudo isso, o Django permite que você crie seus próprios “comandos de gerenciamento” (comandos que você pode executar com `python manage.py`), os quais cuidarão de toda essa manipulação de paths para você. Eles ficam em uma pasta chamada *management/commands* em suas aplicações:

```
$ mkdir -p functional_tests/management/commands
$ touch functional_tests/management/__init__.py
$ touch functional_tests/management/commands/__init__.py
```

O boilerplate em um comando de gerenciamento é uma classe que herda de `django.core.management.BaseCommand` e que define um método chamado `handle`:

## functional\_tests/management/commands/create\_session.py

```
from django.conf import settings
from django.contrib.auth import BACKEND_SESSION_KEY, SESSION_KEY,
get_user_model
User = get_user_model()
from django.contrib.sessions.backends.db import SessionStore
from django.core.management.base import BaseCommand
```

```
class Command(BaseCommand):
```

```
 def add_arguments(self, parser):
 parser.add_argument('email')
```

```
 def handle(self, *args, **options):
 session_key = create_pre_authenticated_session(options['email'])
 self.stdout.write(session_key)
```

```
def create_pre_authenticated_session(email):
 user = User.objects.create(email=email)
 session = SessionStore()
```

```
session[SESSION_KEY] = user.pk
session[BACKEND_SESSION_KEY] =
settings.AUTHENTICATION_BACKENDS[0]
session.save()
return session.session_key
```

Obtivemos o código de `create_pre_authenticated_session` de `test_my_lists.py`. `handle` obterá um endereço de email do parser e então devolverá a chave de sessão que queremos adicionar nos cookies de nosso navegador, e o comando de gerenciamento o exibirá na linha de comando. Experimente executar o seguinte:

```
$ python manage.py create_session a@b.com
Unknown command: 'create_session'
```

Mais um passo: precisamos adicionar `functional_tests` em nosso `settings.py` para que ele o reconheça como uma verdadeira aplicação, que tenha comandos de gerenciamento, além de testes:

## superlists/settings.py

```
+++ b/superlists/settings.py
@@ -42,6 +42,7 @@ INSTALLED_APPS = [
 'lists',
 'accounts',
+ 'functional_tests',
]
```

Agora funciona:

```
$ python manage.py create_session a@b.com
qnslckvp2aga7tm6xuiivyb0ob1akzzwl
```



Se você vir um erro informando que a tabela `auth_user` está faltando, talvez precise executar `manage.py migrate`. Caso isso não funcione, apague o arquivo `db.sqlite3` e execute `migrate` novamente a fim de obter dados limpos.

## Fazendo o FT executar o comando de gerenciamento no servidor

A seguir precisamos ajustar `test_my_lists` para que execute a função

local quando estivermos no servidor local, e faça-o executar o comando de gerenciamento no servidor de staging se estivermos aí:

## functional\_tests/test\_my\_lists.py (ch18l016)

```
from django.conf import settings
from .base import FunctionalTest
from .server_tools import create_session_on_server
from .management.commands.create_session import
create_pre_authenticated_session

class MyListsTest(FunctionalTest):

 def create_pre_authenticated_session(self, email):
 if self.staging_server:
 session_key = create_session_on_server(self.staging_server, email)
 else:
 session_key = create_pre_authenticated_session(email)
 ## para definir um cookie, precisamos antes acessar o domínio.
 ## as páginas 404 são as que carregam mais rapidamente!
 self.browser.get(self.live_server_url + "/404_no_such_url/")
 self.browser.add_cookie(dict(
 name=settings.SESSION_COOKIE_NAME,
 value=session_key,
 path='/',
))

 [...]
```

Vamos ajustar *base.py* também para obter um pouco mais de informações quando preencheremos `self.against_staging`:

## functional\_tests/base.py (ch18l017)

```
from .server_tools import reset_database ❶
[...]
```

```
class FunctionalTest(StaticLiveServerTestCase):

 def setUp(self):
 self.browser = webdriver.Firefox()
 self.staging_server = os.environ.get('STAGING_SERVER')
```

```
if self.staging_server:
 self.live_server_url = 'http://' + self.staging_server
 reset_database(self.staging_server) ❶
```

- ❶ Essa será a nossa função para reiniciar o banco de dados do servidor entre cada teste. Explicarei a lógica do código de criação de sessão, que deverá explicar também como isso funciona.

## Usando o Fabric diretamente de Python

Em vez de usar o comando `fab`, o Fabric disponibiliza uma API que permite executar comandos de servidor do Fabric diretamente inline em seu código Python. Basta deixar que ele saiba qual é a “string de host” com a qual você está se conectando:

### functional\_tests/server\_tools.py

```
from fabric.api import run
from fabric.context_managers import settings

def _get_manage_dot_py(host):
 return f'~/sites/{host}/virtualenv/bin/python ~/sites/{host}/source/manage.py'

def reset_database(host):
 manage_dot_py = _get_manage_dot_py(host)
 with settings(host_string=f'elspeth@{host}'): ❶
 run(f'{manage_dot_py} flush --noinput') ❷

def create_session_on_server(host, email):
 manage_dot_py = _get_manage_dot_py(host)
 with settings(host_string=f'elspeth@{host}'): ❶
 session_key = run(f'{manage_dot_py} create_session {email}') ❷
 return session_key.strip()
```

- ❶ Eis o gerenciador de contexto que define a string de host no formato `user@server-address` (deixei o nome de usuário em meu servidor, `elspeth`, fixo no código, portanto faça ajustes conforme necessário).

- ② Então, depois que estivermos no gerenciador de contexto, podemos simplesmente chamar comandos do Fabric como se estivéssemos em um fabfile.

## Revisão: criando sessões localmente *versus* no ambiente de staging

Tudo isso faz sentido? Talvez um diagrama com um pouco de arte com ASCII ajude:

Localmente:

```
+-----+ +-----+
| MyListsTest | --> | .management.commands.create_session |
| .create_pre_authenticated_session | | .create_pre_authenticated_session |
| (localmente) | | (localmente) |
+-----+ +-----+
```

No ambiente de staging:

```
+-----+ +-----+
| MyListsTest | | .management.commands.create_session |
| .create_pre_authenticated_session | | .create_pre_authenticated_session |
| (localmente) | | (no servidor) |
+-----+ +-----+
 | ^
 v |
+-----+ +-----+ +-----+
| server_tools | --> | fabric | --> | ./manage.py create_session |
| .create_session_on_server | | "run" | | (no servidor) |
| (localmente) | +-----+ +-----+
+-----+
```

Em qualquer um dos casos, vamos ver se isso funciona. Em primeiro lugar, localmente, para ver se não provocamos falhas em nada:

```
$ python manage.py test functional_tests.test_my_lists
[...]
OK
```

Em seguida, no servidor. Vamos fazer um push de nosso código antes:

```
$ git push # será necessário fazer commit das alterações antes.
$ cd deploy_tools
$ fab deploy --host=superlists-staging.ottg.eu
```

Agora vamos executar o teste:

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test \
functional_tests.test_my_lists
[...]
[superlists-staging.ottg.eu] Executing task 'reset_database'
~/sites/superlists-staging.ottg.eu/source/manage.py flush --noinput
[superlists-staging.ottg.eu] out: Syncing...
[superlists-staging.ottg.eu] out: Creating tables ...
[...]
.

Ran 1 test in 25.701s
```

OK

Está parecendo bom! Podemos executar novamente todos os testes por garantia...

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test \
functional_tests
[...]
[superlists-staging.ottg.eu] Executing task 'reset_database'
[...]
Ran 8 tests in 89.494s
```

OK

Viva!



Apresentei uma maneira de administrar o banco de dados de teste, mas você poderia testar outras – por exemplo, se estiver usando MySQL ou Postgres, poderia abrir um túnel SSH para o servidor e utilizar encaminhamento de porta (port forwarding) para conversar diretamente com o banco de dados. Em seguida, poderia corrigir settings.DATABASES durante os FTs para conversar com a porta sujeita ao tunelamento.

**Aviso: tome cuidado para não executar o código**



## de teste no servidor live

Estamos em um território perigoso, agora que temos um código capaz de afetar diretamente um banco de dados no servidor. Então, tome muito, muito cuidado para não destruir acidentalmente o seu banco de dados de produção ao executar os FTs no host incorreto.

Você pode considerar a instalação de algumas medidas de proteção nesse ponto. Por exemplo, é possível colocar o ambiente de staging e de produção em servidores diferentes, e fazer isso de modo que eles utilizem pares de chave distintos para autenticação, com frases-senha (passphrases) diferentes. De modo semelhante, esse é um território perigoso para executar testes em clones de dados de produção. Tenho uma historinha sobre enviar acidentalmente milhares de faturas duplicadas para clientes no Apêndice D. Aprenda com o meu erro.

## Incluindo o nosso código de logging

Antes de terminar, vamos inserir a nossa configuração de logging. Seria conveniente manter o nosso novo código de logging presente, sujeito ao sistema de controle de versões, para que possamos depurar qualquer problema de login no futuro. Afinal de contas, esse código pode indicar que há alguém com segundas intenções...

Vamos começar salvando a configuração do Gunicorn em nosso arquivo de template em *deploy\_tools*:

### deploy\_tools/gunicorn-systemd.template.service (ch18l020)

```
[...]
Environment=EMAIL_PASSWORD=SEKRIT
ExecStart=/home/elspeth/sites/SITENAME/virtualenv/bin/gunicorn \
 --bind unix:/tmp/SITENAME.socket \
 --access-logfile ../access.log \
 --error-logfile ../error.log \
 superlists.wsgi:application
[...]
```

Vamos acrescentar um pequeno lembrete em nossas anotações

sobre provisionamento acerca da necessidade de definir a variável de ambiente com a senha de email usando o arquivo de configuração do Gunicorn:

## deploy\_tools/provisioning\_notes.md (ch18l021)

```
Serviço Systemd
```

```
* veja gunicorn-systemd.template.service
* substitua SITENAME, por exemplo, por staging.my-domain.com
* substitua SEKRIT pela senha de email
[...]
```

## Conclusão

Na verdade, deixar o seu novo código ativo e executando em um servidor sempre tende a eliminar alguns bugs e problemas inesperados de última hora. Tivemos um pouco de trabalho para resolvê-los, mas acabamos com muitas implementações úteis, como resultado.

Temos um adorável decorador genérico `wait`, que, a partir de agora, será uma boa função auxiliar pythônica para os nossos FTs. Temos fixtures de teste que funcionam tanto localmente quanto no servidor, incluindo a capacidade de testar uma integração “real” com emails. Além disso, temos algumas configurações mais robustas para logging.

Entretanto, antes de fazermos a implantação de nosso site live, será melhor dar aos usuários o que eles queriam – o próximo capítulo descreve como lhes proporcionar a capacidade de salvar suas listas em uma página “My Lists” (Minhas Listas).

## Lições aprendidas ao capturar bugs no ambiente de staging

### *As fixtures também precisam funcionar remotamente*

LiveServerTestCase facilita interagir com o banco de dados de teste usando o ORM do Django para testes que executem localmente. Interagir com o banco de dados no servidor de staging não é tão simples assim. Uma solução são os comandos de gerenciamento do Fabric e do Django, conforme mostramos, mas você deve explorar o que lhe for adequado – túneis SSH, por exemplo.

### *Tome muito cuidado ao reiniciar dados em seus servidores*

Um comando capaz de limpar remotamente todo o banco de dados em um de seus servidores é uma arma perigosa, e você deve ter certeza absoluta de que ele jamais atingirá acidentalmente os seus dados de produção.

### *Logging é crucial para depuração no servidor*

No mínimo, você vai querer ver qualquer mensagem de erro que estiver sendo gerada pelo servidor. Para bugs mais complexos, vai também querer fazer o ocasional “print de depuração” e vê-lo em um arquivo, em algum lugar.



## CAPÍTULO 22

# Finalizando “My Lists”: TDD Outside-In

Neste capítulo, gostaria de discutir uma técnica chamada TDD Outside-In (TDD de fora para dentro). É basicamente o que viemos fazendo o tempo todo. Nosso processo de TDD com “laço duplo”, segundo o qual escrevemos o teste funcional antes, e depois os testes de unidade, já é uma manifestação da abordagem outside-in – fazemos o design do sistema a partir de fora e construímos o nosso código em camadas. Agora vou deixar isso explícito e discutir alguns dos problemas comuns envolvidos aí.

### **A alternativa: “Inside-Out”**

A alternativa ao “outside-in” (de fora para dentro) é trabalhar “inside-out” (de dentro para fora), que é o modo como a maioria das pessoas trabalha intuitivamente antes de conhecer o TDD. Depois de definir um design, a tendência natural, às vezes, é implementá-lo começando pelos componentes mais internos, de nível mais baixo, antes.

Por exemplo, quando deparamos com o nosso problema atual – oferecer aos usuários uma página “My Lists” (Minhas Listas) com as listas salvas –, a tentação é começar adicionando um atributo de “proprietário” ao objeto de modelo List, raciocinando que um atributo como esse “obviamente” será necessário. Feito isso, modificaríamos as camadas de código mais periféricas, como as views e os templates, tirando proveito do novo atributo, e então, por fim, adicionaríamos o roteamento de URL para que apontasse para a

nova view.

Nós nos sentimos à vontade porque isso significa que jamais trabalharíamos com uma porção de código que dependesse de algo ainda não implementado. Cada parte do trabalho na camada interna seria uma base sólida sobre a qual construiríamos a próxima camada mais externa.

Contudo, trabalhar de dentro para fora dessa maneira também apresenta alguns pontos fracos.

## **Por que dar preferência para a abordagem “Outside-In”?**

O problema mais evidente com a abordagem inside-out é que ela exige que nos afastemos de um fluxo de trabalho TDD. A primeira falha de nosso teste funcional pode se dar por causa do roteamento de URL ausente, porém decidimos ignorar isso e saímos adicionando atributos em nossos objetos de modelo no banco de dados.

Talvez tenhamos ideias em nossa mente sobre o novo comportamento desejado para as nossas camadas internas, como modelos de banco de dados; com frequência, essas ideias serão muito boas, mas, na verdade, são apenas especulações sobre o que será realmente necessário, pois ainda não construímos as camadas mais externas que as usarão.

Um problema que pode resultar disso é a construção de componentes internos que sejam mais genéricos ou mais capacitados do que realmente precisaremos, o que será um desperdício de tempo, além de uma fonte adicional de complexidade em seu projeto. Outro problema comum é que você criará componentes internos com uma API conveniente para o próprio design interno deles, mas que, posteriormente, acabará se mostrando inapropriado para as chamadas que suas camadas mais externas gostariam de fazer... Pior ainda, você poderá acabar com

componentes internos que – você perceberá mais tarde – não resolvem realmente o problema que suas camadas mais externas precisam resolver.

Em oposição, trabalhar com a abordagem outside-in nos permite usar cada camada para pensar na API desejada que seja a mais conveniente possível para a camada abaixo dela. Vamos ver isso em ação.

## FT para “My Lists”

À medida que trabalharmos com o teste funcional a seguir, começaremos com o lado mais externo (a camada de apresentação), passaremos para as funções de view (ou “controladores”) e, por fim, para as camadas mais internas, que, nesse caso, serão o código dos modelos.

Sabemos que o código de nosso `create_pre_authenticated_session` agora funciona, portanto podemos simplesmente escrever o nosso FT para que procure uma página “My Lists”:

### functional\_tests/test\_my\_lists.py (ch19|001-1)

```
def test_logged_in_users_lists_are_saved_as_my_lists(self):
 # Edith é uma usuária logada
 self.create_pre_authenticated_session('edith@example.com')

 # Edith acessa a página inicial e começa uma lista
 self.browser.get(self.live_server_url)
 self.add_list_item('Reticulate splines')
 self.add_list_item('Immanentize eschaton')
 first_list_url = self.browser.current_url

 # Ela percebe o link para "My lists" pela primeira vez.
 self.browser.find_element_by_link_text('My lists').click()

 # Ela vê que sua lista está lá, nomeada de acordo com

 # o primeiro item da lista
```

```

self.wait_for(
 lambda: self.browser.find_element_by_link_text('Reticulate splines')
)
self.browser.find_element_by_link_text('Reticulate splines').click()
self.wait_for(
 lambda: self.assertEqual(self.browser.current_url, first_list_url)
)

```

Criamos uma lista com dois itens e então verificamos se essa lista aparece em uma nova página “My Lists”, e se está “nomeada” de acordo com o primeiro item da lista.

Vamos validar se isso realmente funciona criando uma segunda lista e vendo se ela aparece também na página My Lists. O FT continua e, aproveitando a ocasião, verificamos se são apenas os usuários logados que podem ver a página “My Lists”:

## functional\_tests/test\_my\_lists.py (ch19|001-2)

```

[...]
self.wait_for(
 lambda: self.assertEqual(self.browser.current_url, first_list_url)
)

Ela decide iniciar outra lista, somente para conferir
self.browser.get(self.live_server_url)
self.add_list_item('Click cows')
second_list_url = self.browser.current_url

Em "my lists", sua nova lista aparece
self.browser.find_element_by_link_text('My lists').click()
self.wait_for(
 lambda: self.browser.find_element_by_link_text('Click cows')
)
self.browser.find_element_by_link_text('Click cows').click()
self.wait_for(
 lambda: self.assertEqual(self.browser.current_url, second_list_url)
)

Ela faz logout A opção "My lists" desaparece
self.browser.find_element_by_link_text('Log out').click()
self.wait_for(lambda: self.assertEqual(

```



```
 self.browser.find_elements_by_link_text('My lists'),
 []
))
```

Nosso FT utiliza um novo método auxiliar `add_list_item`, que abstrai a entrada de texto na caixa de entrada correta. Vamos defini-la em `base.py`:

## functional\_tests/base.py (ch19|001-3)

```
from selenium.webdriver.common.keys import Keys
[...]

def add_list_item(self, item_text):
 num_rows = len(self.browser.find_elements_by_css_selector('#id_list_table
tr'))
 self.get_item_input_box().send_keys(item_text)
 self.get_item_input_box().send_keys(Keys.ENTER)
 item_number = num_rows + 1
 self.wait_for_row_in_list_table(f'{item_number}: {item_text}')
```

Além disso, aproveitando a oportunidade, podemos usá-la em alguns dos outros FTs, assim:

## functional\_tests/test\_list\_item\_validation.py

```
self.add_list_item('Buy wellies')
```

Acho que os FTs se tornarão muito mais legíveis. Fiz um total de seis alterações – veja se você concorda comigo.

Vamos fazer uma execução rápida de todos os FTs, um commit e então voltaremos para o FT no qual estamos trabalhando. Eis a aparência de como deve ser o primeiro erro:

```
$ python3 manage.py test functional_tests.test_my_lists
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate
element: My lists
```

## Camada mais externa: apresentação e templates

O teste está falhando no momento, informando que não é capaz de

encontrar um link contendo “My Lists”. Podemos cuidar disso na camada de apresentação, em *base.html*, em nossa barra de navegação. Eis a alteração mínima de código:

## lists/templates/base.html (ch19I002-1)

```
{% if user.email %}
 <ul class="nav navbar-nav navbar-left">
 My lists

 <ul class="nav navbar-nav navbar-right">
 <li class="navbar-text">Logged in as {{ user.email }}
 Log out

```

É claro que esse link não vai realmente para lugar nenhum, mas ele nos conduz à próxima falha:

```
$ python3 manage.py test functional_tests.test_my_lists
[...]
lambda: self.browser.find_element_by_link_text('Reticulate splines')
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate
element: Reticulate splines
```

Essa falha nos informa que teremos que construir uma página que apresente todas as listas de um usuário pelo título. Vamos começar pelo básico – um URL e um template placeholder para ela.

Novamente, podemos seguir a abordagem outside-in, começando pela camada de apresentação, somente com o URL e nada mais:

## lists/templates/base.html (ch19I002-2)

```
<ul class="nav navbar-nav navbar-left">
 My lists

```

**Descendo uma camada, em direção às funções de view (o controlador)**

Isso causará um erro de template, portanto começaremos a nos deslocar para baixo, a partir da camada de apresentação e dos URLs, em direção à camada do controlador, isto é, as funções de view do Django.

Como sempre, começamos com um teste:

## lists/tests/test\_views.py (ch19|003)

```
class MyListsTest(TestCase):

 def test_my_lists_url_renders_my_lists_template(self):
 response = self.client.get('/lists/users/a@b.com/')
 self.assertTemplateUsed(response, 'my_lists.html')
```

O resultado será este:

```
AssertionError: No templates used to render the response
```

Corrigimos esse erro, ainda no nível da apresentação, em *urls.py*:

## lists/urls.py

```
urlpatterns = [
 url(r'^new$', views.new_list, name='new_list'),
 url(r'^(\d+)/$', views.view_list, name='view_list'),
 url(r'^users/(.+)$', views.my_lists, name='my_lists'),
]
```

Isso resulta em uma falha de teste, que nos diz o que devemos fazer à medida que descemos para o próximo nível:

```
AttributeError: module 'lists.views' has no attribute 'my_lists'
```

Passamos da camada de apresentação para a camada de views e criamos um placeholder mínimo:

## lists/views.py (ch19|005)

```
def my_lists(request, email):
 return render(request, 'my_lists.html')
```

E um template mínimo:

## lists/templates/my\_lists.html

```
{% extends 'base.html' %}
```

```
{% block header_text %}My Lists{% endblock %}
```

Com isso, nossos testes de unidade passam, mas o nosso FT continua no mesmo ponto, informando que a página “My Lists” ainda não mostra nenhuma lista. O teste exige que haja links em que seja possível clicar, e estejam nomeados de acordo com o primeiro item:

```
$ python3 manage.py test functional_tests.test_my_lists
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate
element: Reticulate splines
```

## Outro teste que passa usando a abordagem Outside-In

Em cada etapa, continuamos deixando o FT orientar o desenvolvimento que fazemos.

Iniciando novamente na camada mais externa, no template, começamos a escrever o código do template que gostaríamos de usar a fim de fazer com que a página “My Lists” funcione como queremos. À medida que fazemos isso, começamos a especificar a API que queremos no código das camadas inferiores.

## Uma reestruturação rápida da hierarquia de herança dos templates

No momento, não há nenhum lugar em nosso template-base para que possamos colocar qualquer conteúdo novo. Além do mais, a página “My Lists” não precisa do formulário de novo item, portanto colocaremos isso em um bloco também, tornando-o opcional:

### lists/templates/base.html (ch19|007-1)

```
<div class="row">
 <div class="col-md-6 col-md-offset-3 jumbotron">
 <div class="text-center">
 <h1>{% block header_text %}{% endblock %}</h1>
```

```

 {% block list_form %}
 <form method="POST" action="{% block form_action %}"{% endblock
 %}">
 {{ form.text }}
 {% csrf_token %}
 {% if form.errors %}
 <div class="form-group has-error">
 <div class="help-block">{{ form.text.errors }}</div>
 </div>
 {% endif %}
 </form>
 {% endblock %}
</div>
</div>
</div>

```

## lists/templates/base.html (ch19|007-2)

```

<div class="row">
 <div class="col-md-6 col-md-offset-3">
 {% block table %}
 {% endblock %}
 </div>
</div>

<div class="row">
 <div class="col-md-6 col-md-offset-3">
 {% block extra_content %}
 {% endblock %}
 </div>
</div>

</div>
<script src="/static/jquery-3.1.1.min.js"></script>
[...]
```

## Fazendo o design de nossa API usando o template

Enquanto isso, em *my\_lists.html*, sobrescrevemos `list_form` e dizemos que ele deveria ser vazio...

## lists/templates/my\_lists.html

```
{% extends 'base.html' %}

{% block header_text %}My Lists{% endblock %}

{% block list_form %}{% endblock %}
```

Então podemos simplesmente trabalhar no bloco `extra_content`:

## lists/templates/my\_lists.html

```
[...]

{% block list_form %}{% endblock %}

{% block extra_content %}
 <h2>{{ owner.email }}'s lists</h2> ❶

 {% for list in owner.list_set.all %} ❷
 {{ list.name }} ❸
 {% endfor %}

{% endblock %}
```

Tomamos várias decisões de design nesse template, as quais se infiltrarão pelo código das camadas inferiores:

- ❶ Queremos uma variável chamada `owner` para representar o usuário em nosso template.
- ❷ Queremos ter a capacidade de iterar pelas listas criadas pelo usuário utilizando `owner.list_set.all` (por acaso eu sei que teremos isso gratuitamente usando o ORM do Django).
- ❸ Queremos usar `list.name` para exibir o “nome” da lista, que, no momento, está especificado como o texto de seu primeiro elemento.



O TDD Outside-In às vezes é chamado de “programação baseada no que achamos desejável” (programming by wishful thinking), e você pode ver por quê. Começamos escrevendo um código nos níveis mais altos de acordo com o que desejaríamos que houvesse nos níveis mais baixos, mesmo que esse código ainda não exista!

Podemos executar novamente os nossos FTs, a fim de verificar se não causamos falhas em nada, e para ver se conseguimos avançar um pouco:

```
$ python manage.py test functional_tests
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate
element: Reticulate splines
```

```

Ran 8 tests in 77.613s
```

```
FAILED (errors=1)
```

Bem, não avançamos, mas, pelo menos, não causamos falhas em nada. É hora de fazer um commit:

```
$ git add lists
$ git diff --staged
$ git commit -m "url, placeholder view, and first-cut templates for my_lists"
```

## Descendo para a próxima camada: o que a view passa para o template

Nossa camada de views agora deve atender aos requisitos que definimos na camada de template, fornecendo-lhe os objetos necessários. Nesse caso, é o proprietário da lista:

### lists/tests/test\_views.py (ch19|011)

```
from django.contrib.auth import get_user_model
User = get_user_model()
[...]
class MyListsTest(TestCase):
```

```
def test_my_lists_url_renders_my_lists_template(self):
 [...]

def test_passes_correct_owner_to_template(self):
 User.objects.create(email='wrong@owner.com')
 correct_user = User.objects.create(email='a@b.com')
 response = self.client.get('/lists/users/a@b.com/')
 self.assertEqual(response.context['owner'], correct_user)
```

Eis o resultado:

```
KeyError: 'owner'
```

Assim:

## lists/views.py (ch19|012)

```
from django.contrib.auth import get_user_model
User = get_user_model()
[...]
```

```
def my_lists(request, email):
 owner = User.objects.get(email=email)
 return render(request, 'my_lists.html', {'owner': owner})
```

Com isso, o nosso novo teste passa, mas veremos também um erro do teste anterior. Só precisamos adicionar um usuário para ele também:

## lists/tests/test\_views.py (ch19|013)

```
def test_my_lists_url_renders_my_lists_template(self):
 User.objects.create(email='a@b.com')
 [...]
```

E temos um OK:

```
OK
```

## **Próximo “requisito” da camada de views: novas listas devem registrar o proprietário**

Antes de descermos para a camada de modelo, há outra parte do



código na camada de views que terá de utilizar o nosso modelo: precisamos ter uma maneira de atribuir um proprietário às listas recém-criadas, se o usuário atual estiver logado no site.

Eis uma primeira tentativa de escrever o teste:

## lists/tests/test\_views.py (ch19|014)

```
class NewListTest(TestCase):
 [...]

 def test_list_owner_is_saved_if_user_is_authenticated(self):
 user = User.objects.create(email='a@b.com')
 self.client.force_login(user) ❶
 self.client.post('/lists/new', data={'text': 'new item'})
 list_ = List.objects.first()
 self.assertEqual(list_.owner, user)
```

❶ `force_login()` é o modo como você faz o cliente de testes fazer requisições com um usuário logado.

O teste falha da seguinte forma:

```
AttributeError: 'List' object has no attribute 'owner'
```

Para corrigir isso, podemos tentar escrever um código como este:

## lists/views.py (ch19|015)

```
def new_list(request):
 form = ItemForm(data=request.POST)
 if form.is_valid():
 list_ = List()
 list_.owner = request.user
 list_.save()
 form.save(for_list=list_)
 return redirect(list_)
 else:
 return render(request, 'home.html', {"form": form})
```

Contudo, esse código não funcionará, pois não sabemos ainda como salvar um proprietário de lista:

```
self.assertEqual(list_.owner, user)
AttributeError: 'List' object has no attribute 'owner'
```

## **Ponto de decisão: devemos prosseguir para a próxima camada com um teste em falha?**

Para fazer esse teste passar, conforme escrito agora, temos que descer para a camada de modelo. No entanto, isso significa mais trabalho com um teste em falha, o que não é ideal.

A alternativa é reescrever o teste para deixá-lo mais *isolado* do nível abaixo, usando mocks.

Por um lado, usar mocks exige muito mais esforço, e pode resultar em testes mais difíceis de ler. Por outro, suponha que nossa aplicação fosse mais complexa e houvesse muito mais camadas entre a camada externa e a interna. Pense como seria se deixássemos três ou quatro ou cinco camadas de testes, todas falhando, enquanto esperássemos chegar à camada inferior para implementar nossa funcionalidade crítica. Enquanto os testes estiverem falhando, não teremos certeza se uma camada realmente funciona por si só, ou não. Precisaremos esperar até atingirmos a camada inferior.

Esse é um ponto de decisão com o qual provavelmente você vai deparar em seus próprios projetos. Vamos investigar as duas abordagens. Começaremos tomando o atalho e deixando o teste com falha. No próximo capítulo, retornaremos exatamente para esse ponto e investigaremos como seria se usássemos de mais isolamento.

Vamos fazer um commit e, em seguida, atribuir-lhe uma *tag* como forma de lembrar a nossa situação para o próximo capítulo:

```
$ git commit -am "new_list view tries to assign owner but cant"
$ git tag revisit_this_point_with_isolated_tests
```

## **Descendo para a camada de modelo**

Nosso design com abordagem outside-in gerou dois requisitos para a camada de modelo: queremos atribuir um proprietário para uma lista usando o atributo `.owner` e queremos acessar o proprietário da

lista com a API `owner.list_set.all`.

Vamos escrever um teste para isso:

## lists/tests/test\_models.py (ch19|018)

```
from django.contrib.auth import get_user_model
User = get_user_model()
[...]
```

```
class ListModelTest(TestCase):
```

```
 def test_get_absolute_url(self):
 [...]
```

```
 def test_lists_can_have_owners(self):
 user = User.objects.create(email='a@b.com')
 list_ = List.objects.create(owner=user)
 self.assertIn(list_, user.list_set.all())
```

Temos uma nova falha no teste de unidade:

```
list_ = List.objects.create(owner=user)
[...]
```

`TypeError: 'owner' is an invalid keyword argument for this function`

Esta seria a implementação ingênua:

```
from django.conf import settings
[...]
```

```
class List(models.Model):
 owner = models.ForeignKey(settings.AUTH_USER_MODEL)
```

No entanto, queremos garantir que o proprietário da lista seja opcional. Explícito é melhor do que implícito, e os testes servem de documentação, portanto vamos criar um teste para isso também:

## lists/tests/test\_models.py (ch19|020)

```
def test_list_owner_is_optional(self):
 List.objects.create() # não deve gerar erro
```

Eis a implementação correta:

## lists/models.py

```
from django.conf import settings
[...]
```

```
class List(models.Model):
 owner = models.ForeignKey(settings.AUTH_USER_MODEL, blank=True,
 null=True)

 def get_absolute_url(self):
 return reverse('view_list', args=[self.id])
```

Agora a execução dos testes resulta no erro usual de banco de dados:

```
return Database.Cursor.execute(self, query, params)
django.db.utils.OperationalError: no such column: lists_list.owner_id
```

Isso ocorre porque precisamos fazer algumas migrações:

```
$ python manage.py makemigrations
Migrations for 'lists':
 lists/migrations/0006_list_owner.py
 - Add field owner to list
```

Estamos quase lá; temos mais duas falhas:

```
ERROR: test_redirects_after_POST (lists.tests.test_views.NewListTest)
[...]
```

```
ValueError: Cannot assign "<SimpleLazyObject:
<django.contrib.auth.models.AnonymousUser object at 0x7f364795ef90>>":
"List.owner" must be a "User" instance.
ERROR: test_can_save_a_POST_request (lists.tests.test_views.NewListTest)
```

```
[...]
```

```
ValueError: Cannot assign "<SimpleLazyObject:
<django.contrib.auth.models.AnonymousUser object at 0x7f364795ef90>>":
"List.owner" must be a "User" instance.
```

Estamos voltando para a camada de views agora, somente para organizá-la melhor. Observe que esse código está no antigo teste da view `new_list`, quando não tínhamos um usuário logado. Devíamos salvar o proprietário da lista apenas quando o usuário estivesse realmente logado. O atributo `.is_authenticated` que definimos no Capítulo 19 passa a ser conveniente agora (quando os usuários não estiverem logados, o Django os representará usando uma classe

chamada `AnonymousUser`, cujo `.is_authenticated` é sempre `False`):

## lists/views.py (ch19|023)

```
if form.is_valid():
 list_ = List()
 if request.user.is_authenticated:
 list_.owner = request.user
 list_.save()
 form.save(for_list=list_)
 [...]
```

Isso resulta em testes que passam!

```
$ python manage.py test lists
```

```
[...]
```

```

Ran 39 tests in 0.237s
```

```
OK
```

É uma boa hora para um commit:

```
$ git add lists
```

```
$ git commit -m "lists can have owners, which are saved on creation."
```

## Último passo: passando dados pela API `.name` do template

O último recurso desejado pelo nosso design outside-in era proveniente dos templates, que queriam acessar um “nome” de lista baseado no texto de seu primeiro item:

## lists/tests/test\_models.py (ch19|024)

```
def test_list_name_is_first_item_text(self):
 list_ = List.objects.create()
 Item.objects.create(list=list_, text='first item')
 Item.objects.create(list=list_, text='second item')
 self.assertEqual(list_.name, 'first item')
```

## lists/models.py (ch19|025)

```
@property
```

```
def name(self):
 return self.item_set.first().text
```

E isso, acredite ou não, faz com que o nosso teste passe, e temos uma página “My Lists” funcionando (Figura 22.1)!

```
$ python manage.py test functional_tests
[...]
Ran 8 tests in 93.819s
```

OK

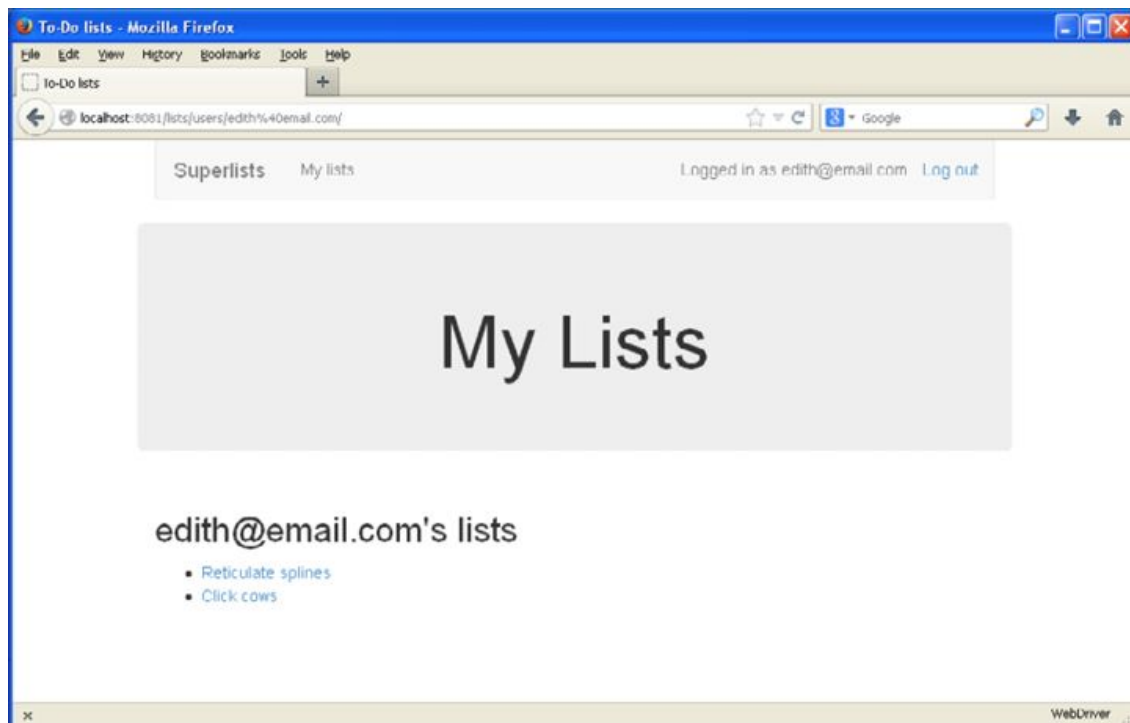
## Decorador @property em Python

Caso você não o tenha visto antes, o decorador @property transforma um método de uma classe de modo que ele apareça para o mundo externo como um atributo.

É um recurso eficaz da linguagem, pois facilita a implementação do “duck typing” (tipagem pato) – alterar a implementação de uma propriedade sem modificar a interface da classe. Em outras palavras, se decidirmos alterar .name para que seja um atributo “de verdade” no modelo, armazenado como um texto no banco de dados, poderemos fazer isso de modo totalmente transparente – no que concerne ao restante de nosso código, eles continuarão podendo simplesmente acessar .name e obter o nome da lista, sem precisar conhecer a implementação. Raymond Hettinger fez uma ótima palestra apropriada aos iniciantes sobre esse assunto na Pycon alguns anos atrás (<https://www.youtube.com/watch?v=HTLu2DFOdTg>), a qual recomendo de forma entusiasmada (além do mais, a apresentação inclui aproximadamente um milhão de boas práticas para design de classes pythônicas).

É claro que, na linguagem do template do Django, .name continuaria chamando o método, mesmo que não tivesse @property, mas essa é uma particularidade do Django, e não se aplica a Python em geral...

Contudo, sabemos que trapaceamos para chegar lá. O Testing Goat está lançando um olhar de suspeita para nós. Deixamos um teste com falha em uma camada enquanto implementamos suas dependências na camada inferior. Vamos ver como a situação se desenrolaria se tivéssemos um melhor isolamento para os testes...



*Figura 22.1 – Página “My Lists” em toda sua glória (e a prova de que testei no Windows).*

## **TDD Outside-In (de fora para dentro)**

## *TDD Outside-In*

É uma metodologia para a implementação de código orientado a testes, que se dá começando pelas camadas “externas” (apresentação, GUI) e prosseguindo para as camadas “internas”, passo a passo, passando pelas camadas de view/controlador e descendo em direção à camada de modelo. A ideia é orientar o design de seu código de acordo com o uso que ele terá, em vez de tentar prever requisitos de baixo para cima.



## *Programação baseada no que achamos desejável*

O processo com abordagem outside-in às vezes é chamado de “programação baseada no que achamos desejável” (programming by wishful thinking). Na verdade, qualquer tipo de TDD envolve um pouco de ideias que achamos desejáveis. Estamos sempre escrevendo testes para algo que ainda não existe.

### *As armadilhas da abordagem outside-in*

A abordagem outside-in não é uma solução para todos os problemas. Ela nos incentiva a colocar o foco em itens que sejam imediatamente visíveis ao usuário, mas não nos lembrará automaticamente de escrever outros testes críticos para recursos menos visíveis aos usuários – como segurança. Você terá que se lembrar deles por conta própria.



## CAPÍTULO 23

# Isolamento de testes e “Ouvindo os seus testes”

No capítulo anterior, tomamos a decisão de deixar um teste de unidade em falha na camada de views enquanto continuamos escrevendo mais testes e mais código na camada de modelos para que o teste passasse.

Nós nos saímos bem com isso porque nossa aplicação era simples, mas devo enfatizar que, em uma aplicação mais complexa, essa seria uma decisão perigosa. Prosseguir com o trabalho em níveis inferiores enquanto você não tem certeza de que os níveis mais altos *realmente* foram concluídos ou não é uma estratégia arriscada.



Sou grato a Gary Bernhardt, que leu uma versão preliminar do capítulo anterior e me incentivou a fazer uma discussão mais extensa sobre isolamento de testes.

Garantir o isolamento entre camadas envolve mais esforços (e mais dos terríveis mocks!), mas também pode ajudar a obter um design melhor, como veremos neste capítulo.

### **Voltando ao nosso ponto de decisão: a camada de views depende de um código não escrito para modelos**

Vamos voltar ao ponto em que estávamos no meio do último capítulo, quando não conseguíamos fazer a view `new_list` funcionar porque as listas ainda não tinham o atributo `.owner`.

Voltaremos no tempo e faremos um checkout da base de código antiga usando a tag que salvamos antes, de modo que vejamos como o situação se desenrolaria se tivéssemos testes mais isolados:

```
$ git checkout -b more-isolation # um branch para esse experimento
$ git reset --hard revisit_this_point_with_isolated_tests
```

Eis a aparência de nosso teste com falha:

## lists/tests/test\_views.py

```
class NewListTest(TestCase):
 [...]

 def test_list_owner_is_saved_if_user_is_authenticated(self):
 user = User.objects.create(email='a@b.com')
 self.client.force_login(user)
 self.client.post('/lists/new', data={'text': 'new item'})
 list_ = List.objects.first()
 self.assertEqual(list_.owner, user)
```

E eis a aparência que tinha a nossa tentativa de solução:

## lists/views.py

```
def new_list(request):
 form = ItemForm(data=request.POST)
 if form.is_valid():
 list_ = List()
 list_.owner = request.user
 list_.save()
 form.save(for_list=list_)
 return redirect(list_)
 else:
 return render(request, 'home.html', {"form": form})
```

Nesse ponto, o teste da view está falhando porque ainda não temos a camada do modelo:

```
self.assertEqual(list_.owner, user)
AttributeError: 'List' object has no attribute 'owner'
```



Você não verá esse erro, a menos que faça de fato o checkout do código antigo e reverta *lists/models.py*. Definitivamente você deve fazer isso; parte do objetivo deste capítulo é ver se podemos realmente escrever testes para uma camada de modelos que ainda não existe.

## Primeira tentativa de usar mocks para isolamento

As listas ainda não têm proprietários, mas podemos deixar os testes da camada de views fingirem que têm, usando um pouco de simulação:

### lists/tests/test\_views.py (ch20I003)

```
from unittest.mock import patch
[...]
@patch('lists.views.List') ❶
@patch('lists.views.ItemForm') ❷
def test_list_owner_is_saved_if_user_is_authenticated(
 self, mockItemFormClass, mockListClass ❸
):
 user = User.objects.create(email='a@b.com')
 self.client.force_login(user)

 self.client.post('/lists/new', data={'text': 'new item'})

 mock_list = mockListClass.return_value ❹
 self.assertEqual(mock_list.owner, user) ❺
```

- ❶ Simulamos a classe `List` para que seja possível ter acesso a qualquer lista que tenha sido criada pela view.
- ❷ Também simularemos o `ItemForm`. Caso contrário, nosso formulário lançará um erro quando `form.save()` for chamado, pois ele não pode usar um objeto mock como chave estrangeira para o `Item` que ele quer criar. Depois que você começar a simular, pode ser difícil parar!
- ❸ Os objetos mock são injetados nos argumentos do teste na ordem inversa em que são declarados. Testes com muitos mocks geralmente têm essa assinatura estranha, com o `)`: pendurado.

Você se acostumará com ele!

- ④ A instância de lista à qual a view terá acesso será o valor de retorno da classe `List` simulada.
- ⑤ Podemos fazer asserções sobre o fato de o atributo `.owner` estar definido ou não.

Se tentarmos executar esse teste agora, ele deverá passar:

```
$ python manage.py test lists
[...]
Ran 37 tests in 0.145s
OK
```

Se não vir o teste passar, certifique-se de que o código de suas views em `views.py` esteja exatamente como eu mostrei, usando `List()`, e não `List.objects.create`.



O uso de mocks compromete você com maneiras específicas de usar uma API. Essa é uma das muitas negociações de custo-benefício envolvidas no uso de objetos mock.

## Usando `side_effects` de mocks para verificar a sequência de eventos

O problema com esse teste é que ele ainda pode permitir que escrevamos o código incorreto por engano. Suponha que chamemos `save` acidentalmente antes da atribuição do proprietário:

### `lists/views.py`

```
if form.is_valid():
 list_ = List()
 list_.save()
 list_.owner = request.user
 form.save(for_list=list_)
 return redirect(list_)
```

O teste, conforme escrito nesse momento, continua passando:

OK

Desse modo, estritamente falando, precisamos verificar não só se o proprietário foi atribuído, mas também se foi atribuído *antes* de `save` ser chamado em nosso objeto de lista.

Eis o modo como poderíamos testar a sequência de eventos usando mocks – você pode simular uma função e usá-la como um espião para verificar o estado do mundo no momento em que ela for chamada:

## lists/tests/test\_views.py (ch20|005)

```
@patch('lists.views.List')
@patch('lists.views.ItemForm')
def test_list_owner_is_saved_if_user_is_authenticated(
 self, mockItemFormClass, mockListClass
):
 user = User.objects.create(email='a@b.com')
 self.client.force_login(user)
 mock_list = mockListClass.return_value

 def check_owner_assigned(): ❶
 self.assertEqual(mock_list.owner, user)
 mock_list.save.side_effect = check_owner_assigned ❷

 self.client.post('/lists/new', data={'text': 'new item'})

 mock_list.save.assert_called_once_with() ❸
```

- ❶ Definimos uma função que faz a asserção sobre aquilo que queremos que aconteça antes: verificamos se o proprietário da lista foi definido.
- ❷ Atribuímos essa função de verificação como um `side_effect` para aquilo que queremos verificar se ocorreu em segundo lugar. Quando a view chama a nossa função `save` simulada, ela passará por essa asserção. Devemos garantir que isso será configurado antes de realmente chamarmos a função que estamos testando.
- ❸ Por fim, garantimos que a função com o `side_effect` tenha realmente sido disparada – isto é, que executamos `.save()`. Caso contrário, nossa asserção pode não ter sido de fato executada.





Dois erros comuns quando usamos os efeitos colaterais de mocks são: atribuir o efeito colateral tarde demais (isto é, *depois* que você chamou a função em teste) e esquecer-se de verificar se a função de efeito colateral foi realmente chamada. E, por comum, quero dizer: “Cometi esses dois erros várias vezes *enquanto estava escrevendo este capítulo*”.

Nesse ponto, se você ainda tiver o código “quebrado” anterior, no qual atribuímos o proprietário, mas chamamos `save` na ordem incorreta, deverá ver agora uma falha:

```
FAIL: test_list_owner_is_saved_if_user_is_authenticated
(lists.tests.test_views.NewListTest)
[...]
File ".../superlists/lists/views.py", line 17, in new_list
 list_.save()
[...]
File ".../superlists/lists/tests/test_views.py", line 74, in
check_owner_assigned
 self.assertEqual(mock_list.owner, user)
AssertionError: <MagicMock name='List().owner' id='140691452447208'> !=
<User:
User object>
```

Observe como a falha ocorre quando tentamos salvar os dados e, então, entramos em nossa função `side_effect`.

Podemos fazer o teste passar novamente assim:

## lists/views.py

```
if form.is_valid():
 list_ = List()
 list_.owner = request.user
 list_.save()
 form.save(for_list=list_)
 return redirect(list_)
```

...

OK

Mas, cara, esse teste está ficando feio!

## Ouçã os seus testes: testes feios sinalizam a necessidade de refatorar

Sempre que você se vir precisando escrever um teste como esse e estiver achando que é uma tarefa difícil, é provável que seus testes estejam tentando lhe dizer algo. Oito linhas de configuração (duas linhas para mocks, três para configurar um usuário e mais três para a nossa função de efeito colateral) é demais.

O que esse teste tenta nos dizer é que a nossa view está fazendo muitas tarefas, lidando com a criação de um formulário, criando um novo objeto de lista e decidindo se deve ou não salvar um proprietário para a lista.

Já vimos que podemos deixar nossas views mais simples e fáceis de entender passando parte das tarefas para uma classe de formulário. Por que a view precisa criar o objeto de lista? Talvez o nosso `ItemForm.save` poderia fazer isso? E por que a view precisa tomar decisões sobre salvar ou não o `request.user`? Novamente, o formulário poderia fazer isso.

Enquanto estamos atribuindo mais responsabilidades a esse formulário, temos a impressão de que, provavelmente, ele deveria ter um novo nome também. Poderíamos chamá-lo de `NewListForm`, pois é uma representação mais apropriada para o que ele faz... ou algo parecido?

### lists/views.py

```
não entre com esse código ainda; estamos só imaginando-o.
```

```
def new_list(request):
 form = NewListForm(data=request.POST)
 if form.is_valid():
 list_ = form.save(owner=request.user) # cria tanto List quanto Item
 return redirect(list_)
 else:
 return render(request, 'home.html', {"form": form})
```

Assim seria melhor! Vamos ver como chegaríamos a esse estado

usando testes totalmente isolados.

## Reescrevendo nossos testes para a view de modo totalmente isolado

Nossa primeira tentativa de uma suíte de testes para essa view foi altamente *integrada*. Ela precisava que a camada de banco de dados e a camada de formulários estivessem totalmente funcionais para que os testes passassem. Começamos tentando deixá-la mais isolada, portanto vamos fazer todo o processo.

## Mantenha a antiga suíte de testes integrados presente, como verificação de sanidade

Vamos renomear a nossa antiga classe `NewListTest` para `NewListViewIntegratedTest` e jogar fora a nossa tentativa de um teste com mocks para salvar o proprietário, trazendo de volta a versão integrada, por enquanto com um skip:

### lists/tests/test\_views.py (ch20I008)

```
import unittest
[...]
```

```
class NewListViewIntegratedTest(TestCase):

 def test_can_save_a_POST_request(self):
 [...]
```

```
@unittest.skip
def test_list_owner_is_saved_if_user_is_authenticated(self):
 user = User.objects.create(email='a@b.com')
 self.client.force_login(user)
 self.client.post('/lists/new', data={'text': 'new item'})
 list_ = List.objects.first()
 self.assertEqual(list_.owner, user)
```



Você já ouviu o termo “teste de integração” e está se perguntando qual é a diferença entre ele e um “teste integrado”? Dê uma olhada na caixa de definição no Capítulo 26.

```
$ python manage.py test lists
```

```
[...]
```

```
Ran 37 tests in 0.139s
```

```
OK
```

## Uma nova suíte de testes com total isolamento

Vamos começar com um quadro em branco e ver se podemos usar testes isolados a fim de obter um substituto para a nossa view `new_list`. Nós a chamaremos de `new_list2`, ela será implementada ao lado da view antiga e, quando estiver pronta, nós a trocaremos e veremos se os testes integrados antigos continuarão passando:

### `lists/views.py` (ch20l009)

```
def new_list(request):
```

```
 [...]
```

```
def new_list2(request):
```

```
 pass
```

## Pensando em termos de colaboradores

Para reescrever os nossos testes de modo que estejam totalmente isolados, precisamos jogar fora o nosso modo antigo de pensar nos testes em termos dos efeitos “reais” da view em recursos como o banco de dados e, de modo alternativo, pensar em termos dos objetos que colaboram com os testes e como é a interação entre eles.

No novo mundo, o principal colaborador da view será um objeto de formulário, portanto vamos simulá-lo para que possamos controlá-lo totalmente, e para que possamos definir, por meio do que achamos ser desejável, a maneira como queremos que o nosso formulário

funcione:

## lists/tests/test\_views.py (ch20|010)

```
from unittest.mock import patch
from django.http import HttpRequest
from lists.views import new_list2
[...]
```

```
@patch('lists.views.NewListForm') ❷
class NewListViewUnitTest(unittest.TestCase): ❶
```

```
 def setUp(self):
 self.request = HttpRequest()
 self.request.POST['text'] = 'new list item' ❸
```

```
 def test_passes_POST_data_to_NewListForm(self, mockNewListForm):
 new_list2(self.request)
 mockNewListForm.assert_called_once_with(data=self.request.POST) ❹
```

- ❶ A classe `TestCase` do Django facilita bastante escrever testes integrados. Como forma de garantir que estamos escrevendo testes de unidade “puros” e isolados, utilizaremos somente `unittest.TestCase`.
- ❷ Simulamos a classe `NewListForm` (que nem mesmo existe ainda). Ela será usada em todos os testes, portanto vamos simulá-la no nível de classe.
- ❸ Criamos uma requisição `POST` básica em `setUp`, montando manualmente a requisição, em vez de utilizar o Django Test Client (excessivamente integrado).
- ❹ Fazemos a primeira verificação em nossa nova view: verificamos se ela inicializa o seu colaborador, o `NewListForm`, com o construtor correto – os dados da requisição.

O teste começará com uma falha informando que ainda não temos um `NewListForm` em nossa view:

```
AttributeError: <module 'lists.views' from '/.../superlists/lists/views.py'>
does not have the attribute 'NewListForm'
```

Vamos criar um placeholder para ele:

## lists/views.py (ch20l011)

```
from lists.forms import ExistingListItemForm, ItemForm, NewListForm
[...]
```

e:

## lists/forms.py (ch20l012)

```
class ItemForm(forms.models.ModelForm):
 [...]
```

```
class NewListForm(object):
 pass
```

```
class ExistingListItemForm(ItemForm):
 [...]
```

Em seguida, temos uma falha de verdade:

```
AssertionError: Expected 'NewListForm' to be called once. Called 0 times.
```

E fazemos uma implementação assim:

## lists/views.py (ch20l012-2)

```
def new_list2(request):
 NewListForm(data=request.POST)
$ python manage.py test lists
[...]
```

```
Ran 38 tests in 0.143s
OK
```

Vamos continuar. Se o formulário for válido, queremos chamar save nele:

## lists/tests/test\_views.py (ch20l013)

```
from unittest.mock import patch, Mock
[...]
```

```
@patch('lists.views.NewListForm')
class NewListViewUnitTest(unittest.TestCase):
```

```

def setUp(self):
 self.request = HttpRequest()
 self.request.POST['text'] = 'new list item'
 self.request.user = Mock()

def test_passes_POST_data_to_NewListForm(self, mockNewListForm):
 new_list2(self.request)
 mockNewListForm.assert_called_once_with(data=self.request.POST)

def test_saves_form_with_owner_if_form_valid(self, mockNewListForm):
 mock_form = mockNewListForm.return_value
 mock_form.is_valid.return_value = True
 new_list2(self.request)
 mock_form.save.assert_called_once_with(owner=self.request.user)

```

Isso nos leva a:

## lists/views.py (ch20|014)

```

def new_list2(request):
 form = NewListForm(data=request.POST)
 form.save(owner=request.user)

```

No caso em que o formulário é válido, queremos que a view devolva um redirecionamento para nos levar a ver o objeto que o formulário acabou de criar. Portanto, simulamos outro dos colaboradores da view – a função `redirect`:

## lists/tests/test\_views.py (ch20|015)

```

@patch('lists.views.redirect') ❶
def test_redirects_to_form_returned_object_if_form_valid(
 self, mock_redirect, mockNewListForm ❷
):
 mock_form = mockNewListForm.return_value
 mock_form.is_valid.return_value = True ❸

 response = new_list2(self.request)

 self.assertEqual(response, mock_redirect.return_value) ❹
 mock_redirect.assert_called_once_with(mock_form.save.return_value) ❺

```

- ❶ Simulamos a função `redirect`, dessa vez no nível de método.
- ❷ Decoradores `patch` são aplicados no nível mais interno antes, portanto o novo mock é injetado em nosso método antes de `mockNewListForm`.
- ❸ Especificamos que estamos testando o caso em que o formulário é válido.
- ❹ Verificamos se a resposta da view é a resposta da função `redirect`.
- ❺ Verificamos se a função `redirect` foi chamada com o objeto que o formulário devolve no `save`.

Isso nos leva ao seguinte:

## lists/views.py (ch20l016)

```
def new_list2(request):
 form = NewListForm(data=request.POST)
 list_ = form.save(owner=request.user)
 return redirect(list_)
```

```
$ python manage.py test lists
```

```
[...]
```

```
Ran 40 tests in 0.163s
```

```
OK
```

E agora veremos o caso com falha – se o formulário for inválido, queremos renderizar o template da página inicial:

## lists/tests/test\_views.py (ch20l017)

```
@patch('lists.views.render')
def test_renders_home_template_with_form_if_form_invalid(
 self, mock_render, mockNewListForm
):
 mock_form = mockNewListForm.return_value
 mock_form.is_valid.return_value = False

 response = new_list2(self.request)

 self.assertEqual(response, mock_render.return_value)
 mock_render.assert_called_once_with(
 self.request, 'home.html', {'form': mock_form}
```



)

Eis o resultado:

```
AssertionError: <HttpResponseRedirect status_code=302, "te[114 chars]%3E">
!=
<MagicMock name='render()' id='140244627467408'>
```



Quando usamos métodos de asserção em mocks, como `assert_called_once_with`, é duplamente importante garantir que você execute o teste e o veja falhar. É muito fácil cometer um erro de digitação no nome de sua função de asserção e acabar chamando um método mock que não faz nada (o meu foi escrever `asssert_called_once_with` com três “esses”; experimente fazer isso!).

Provocamos um erro proposital, somente para garantir que nossos testes estejam completos:

## lists/views.py (ch20|018)

```
def new_list2(request):
 form = NewListForm(data=request.POST)
 list_ = form.save(owner=request.user)
 if form.is_valid():
 return redirect(list_)
 return render(request, 'home.html', {'form': form})
```

O teste passa, mas não deveria! Vamos criar então mais um teste:

## lists/tests/test\_views.py (ch20|019)

```
def test_does_not_save_if_form_invalid(self, mockNewListForm):
 mock_form = mockNewListForm.return_value
 mock_form.is_valid.return_value = False
 new_list2(self.request)
 self.assertFalse(mock_form.save.called)
```

Esse teste falha:

```
self.assertFalse(mock_form.save.called)
AssertionError: True is not false
```

Chegamos à nossa pequena view, finalizada e organizada:

## lists/views.py

```
def new_list2(request):
 form = NewListForm(data=request.POST)
 if form.is_valid():
 list_ = form.save(owner=request.user)
 return redirect(list_)
 return render(request, 'home.html', {'form': form})
...
```

```
$ python manage.py test lists
```

```
[...]
```

```
Ran 42 tests in 0.163s
```

```
OK
```

## Descendo para a camada de formulários

Então implementamos nossa função de view baseada no que “achamos desejável” em um formulário chamado `NewListForm`, que nem sequer existe ainda.

Precisaremos que o método `save` do formulário crie uma nova lista e um novo item baseado no texto dos dados validados de POST do formulário. Se fôssemos simplesmente mergulhar de cabeça e usar o ORM, o código poderia ser um pouco parecido com este:

```
class NewListForm(models.Form):

 def save(self, owner):
 list_ = List()
 if owner:
 list_.owner = owner
 list_.save()
 item = Item()
 item.list = list_
 item.text = self.cleaned_data['text']
 item.save()
```

Essa implementação depende de duas classes da camada de modelo, `Item` e `List`. Então qual seria a aparência de um teste bem isolado?

```
class NewListFormTest(unittest.TestCase):
```

```

@patch('lists.forms.List') ❶
@patch('lists.forms.Item') ❶
def test_save_creates_new_list_and_item_from_post_data(
 self, mockItem, mockList ❶
):
 mock_item = mockItem.return_value
 mock_list = mockList.return_value
 user = Mock()
 form = NewListForm(data={'text': 'new item text'})
 form.is_valid() ❷

 def check_item_text_and_list():
 self.assertEqual(mock_item.text, 'new item text')
 self.assertEqual(mock_item.list, mock_list)
 self.assertTrue(mock_list.save.called)
 mock_item.save.side_effect = check_item_text_and_list ❸

 form.save(owner=user)

 self.assertTrue(mock_item.save.called) ❹

```

- ❶ Simulamos os dois colaboradores de nosso formulário da camada de modelos abaixo.
- ❷ Temos que chamar `is_valid()` para que o formulário preencha o dicionário `.cleaned_data`, no qual os dados validados são armazenados.
- ❸ Usamos o método `side_effect` para garantir que, quando salvamos o objeto do novo item, faremos isso com uma `List` salva e com o texto correto do item.
- ❹ Como sempre, verificamos cuidadosamente se a nossa função de efeito colateral foi realmente chamada.

Credo! Que teste feio!

## Continue ouvindo os seus testes: removendo o código de ORM de nossa aplicação

Novamente, esses testes estão tentando nos dizer algo: o ORM do Django é difícil de simular, e a nossa classe de formulário precisa

saber muito sobre o seu funcionamento. Programando de novo com base no que achamos desejável, como seria uma API mais simples, que nosso formulário pudesse usar? Que tal algo assim?

```
def save(self):
 List.create_new(first_item_text=self.cleaned_data['text'])
```

Nossas ideias sobre o que achamos desejável dizem o seguinte: que tal um método auxiliar que estivesse na classe `List`<sup>1</sup> e encapsulasse toda a lógica de salvar um novo objeto de lista e seu primeiro item associado?

Então vamos escrever um teste para isso:

## lists/tests/test\_forms.py (ch20|021)

```
import unittest
from unittest.mock import patch, Mock
from django.test import TestCase

from lists.forms import (
 DUPLICATE_ITEM_ERROR, EMPTY_ITEM_ERROR,
 ExistingListItemForm, ItemForm, NewListForm
)
from lists.models import Item, List
[...]
```

```
class NewListFormTest(unittest.TestCase):

 @patch('lists.forms.List.create_new')
 def test_save_creates_new_list_from_post_data_if_user_not_authenticated(
 self, mock_List_create_new
):
 user = Mock(is_authenticated=False)
 form = NewListForm(data={'text': 'new item text'})
 form.is_valid()
 form.save(owner=user)
 mock_List_create_new.assert_called_once_with(
 first_item_text='new item text'
)
```

Além disso, aproveitando a ocasião, podemos testar o caso em que o usuário está autenticado também:

## lists/tests/test\_forms.py (ch20l022)

```
@patch('lists.forms.List.create_new')
def test_save_creates_new_list_with_owner_if_user_authenticated(
 self, mock_List_create_new
):
 user = Mock(is_authenticated=True)
 form = NewListForm(data={'text': 'new item text'})
 form.is_valid()
 form.save(owner=user)
 mock_List_create_new.assert_called_once_with(
 first_item_text='new item text', owner=user
)
```

Você pode ver que esse é um teste muito mais legível. Vamos começar implementando o nosso novo formulário. Começaremos pela importação:

## lists/forms.py (ch20l023)

```
from lists.models import Item, List
```

O mock agora nos diz para criar um placeholder para o nosso método `create_new`:

```
AttributeError: <class 'lists.models.List'> does not have the attribute
'create_new'
```

## lists/models.py

```
class List(models.Model):

 def get_absolute_url(self):
 return reverse('view_list', args=[self.id])

 def create_new():
 pass
```

Depois de alguns passos, devemos acabar com um método `save` de formulário como este:

## lists/forms.py (ch20|025)

```
class NewListForm(ItemForm):

 def save(self, owner):
 if owner.is_authenticated:
 List.create_new(first_item_text=self.cleaned_data['text'], owner=owner)
 else:
 List.create_new(first_item_text=self.cleaned_data['text'])
```

E com testes que passam:

```
$ python manage.py test lists
Ran 44 tests in 0.192s
OK
```

## Ocultando o código de ORM por trás de métodos auxiliares

Uma das técnicas que surgiram de nosso uso de testes isolados foi o “método auxiliar de ORM”.

O ORM do Django permite fazer o que é preciso rapidamente, com uma sintaxe razoavelmente legível (sem dúvida é muito melhor que um SQL puro!). Algumas pessoas, porém, gostam de tentar minimizar o volume de código de ORM na aplicação – particularmente o removendo das camadas de views e de formulários.

Um dos motivos é que isso facilita bastante testar essas camadas. Entretanto, outra razão é o fato de nos forçar a implementar funções auxiliares que expressem mais claramente a nossa lógica do domínio. Compare:

```
list_ = List()
list_.save()
item = Item()
item.list = list_
item.text = self.cleaned_data['text']
item.save()
```

Com:

```
List.create_new(first_item_text=self.cleaned_data['text'])
```

Isso se aplica à leitura de consultas assim como às escritas. Pense em algo como:

```
Book.objects.filter(in_print=True, pub_date__lte=datetime.today())
```

Em comparação com um método auxiliar como:

```
Book.all_available_books()
```

Quando implementamos funções auxiliares, podemos lhes dar nomes que expressem o que estamos fazendo em termos do domínio do negócio, o que pode realmente deixar o nosso código mais legível, assim como nos proporcionar a vantagem de manter todas as chamadas de ORM na camada de modelos e, desse modo, deixar a nossa aplicação como um todo com baixo acoplamento.

## Finalmente descendo para a camada de modelos

Na camada de modelos, não precisaremos mais escrever testes isolados – a questão principal sobre a camada de modelos é integrá-lo com o banco de dados, portanto escrever testes integrados será apropriado:

### lists/tests/test\_models.py (ch201026)

```
class ListModelTest(TestCase):

 def test_get_absolute_url(self):
 list_ = List.objects.create()
 self.assertEqual(list_.get_absolute_url(), f'/lists/{list_.id}/')

 def test_create_new_creates_list_and_first_item(self):
 List.create_new(first_item_text='new item text')
 new_item = Item.objects.first()
 self.assertEqual(new_item.text, 'new item text')
 new_list = List.objects.first()
 self.assertEqual(new_item.list, new_list)
```

O que resulta em:

`TypeError: create_new() got an unexpected keyword argument 'first_item_text'`  
Isso nos levará à primeira tentativa de implementação, que tem o seguinte aspecto:

## lists/models.py (ch20|027)

```
class List(models.Model):

 def get_absolute_url(self):
 return reverse('view_list', args=[self.id])

 @staticmethod
 def create_new(first_item_text):
 list_ = List.objects.create()
 Item.objects.create(text=first_item_text, list=list_)
```

Observe que conseguimos chegar até a camada de modelos, criando um bom design para as camadas de views e de formulários, e o modelo List ainda não aceita ter um proprietário!

Vamos agora testar o caso em que a lista deve ter um proprietário, adicionando o seguinte:

## lists/tests/test\_models.py (ch20|028)

```
from django.contrib.auth import get_user_model
User = get_user_model()
[...]

def test_create_new_optionally_saves_owner(self):
 user = User.objects.create()
 List.create_new(first_item_text='new item text', owner=user)
 new_list = List.objects.first()
 self.assertEqual(new_list.owner, user)
```

Aproveitando a ocasião, podemos escrever os testes para o novo atributo de proprietário:

## lists/tests/test\_models.py (ch20|029)

```
class ListModelTest(TestCase):
 [...]

 def test_lists_can_have_owners(self):
 List(owner=User()) # não deve gerar um erro
```



```
def test_list_owner_is_optional(self):
 List().full_clean() # não deve gerar um erro
```

Esses testes são quase iguais aos que usamos no último capítulo, porém eu os reescrevi de maneira um pouco diferente, de modo que eles não salvem realmente os objetos – tê-los somente como objetos em memória é suficiente para esse teste.



Utilize objetos de modelo em memória (não salvos) em seus testes sempre que puder; isso fará com que seus testes sejam mais rápidos.

O resultado será este:

```
$ python manage.py test lists
[...]
ERROR: test_create_new_optionally_saves_owner
TypeError: create_new() got an unexpected keyword argument 'owner'
[...]
ERROR: test_lists_can_have_owners (lists.tests.test_models.ListModelTest)
TypeError: 'owner' is an invalid keyword argument for this function
[...]
Ran 48 tests in 0.204s
FAILED (errors=2)
```

Fazemos a implementação, exatamente como no último capítulo:

## lists/models.py (ch20|030-1)

```
from django.conf import settings
[...]

class List(models.Model):
 owner = models.ForeignKey(settings.AUTH_USER_MODEL, blank=True,
 null=True)
 [...]
```

Isso nos dará as falhas usuais de integridade, até fazermos uma migração:

```
django.db.utils.OperationalError: no such column: lists_list.owner_id
```

Feita a migração, restarão três falhas:

```
ERROR: test_create_new_optionally_saves_owner
TypeError: create_new() got an unexpected keyword argument 'owner'
[...]
ValueError: Cannot assign "<SimpleLazyObject:
<django.contrib.auth.models.AnonymousUser object at 0x7f5b2380b4e0>>":
>List.owner" must be a "User" instance.
ValueError: Cannot assign "<SimpleLazyObject:
<django.contrib.auth.models.AnonymousUser object at 0x7f5b237a12e8>>":
>List.owner" must be a "User" instance.
```

Vamos lidar com a primeira, que está em nosso método `create_new`:

## lists/models.py (ch20I030-3)

```
@staticmethod
def create_new(first_item_text, owner=None):
 list_ = List.objects.create(owner=owner)
 Item.objects.create(text=first_item_text, list=list_)
```

## De volta às views

Dois de nossos antigos testes integrados para a camada de views estão falhando. O que está acontecendo?

```
ValueError: Cannot assign "<SimpleLazyObject:
<django.contrib.auth.models.AnonymousUser object at 0x7fbad1cb6c10>>":
>List.owner" must be a "User" instance.
```

Ah, a antiga view ainda não sabe muito bem o que fazer com os proprietários de lista:

## lists/views.py

```
if form.is_valid():
 list_ = List()
 list_.owner = request.user
 list_.save()
```

Esse é o ponto em que percebemos que o nosso código antigo não estava adequado para a sua finalidade. Vamos fazer a correção para que todos os nossos testes passem:

## lists/views.py (ch20I031)

```

def new_list(request):
 form = ItemForm(data=request.POST)
 if form.is_valid():
 list_ = List()
 if request.user.is_authenticated:
 list_.owner = request.user
 list_.save()
 form.save(for_list=list_)
 return redirect(list_)
 else:
 return render(request, 'home.html', {"form": form})

```

```

def new_list2(request):
 [...]

```



Uma das vantagens dos testes integrados é que eles ajudam você a identificar interações menos previsíveis como essa. Havíamos nos esquecido de escrever um teste para o caso em que o usuário não está autenticado, mas, como os testes integrados utilizam toda a pilha para baixo, erros da camada de modelo se tornam evidentes para nos dizer que esquecemos algo.

```

$ python manage.py test lists

```

```

[...]
Ran 48 tests in 0.175s
OK

```

## A hora da verdade (e os riscos da simulação)

Vamos então experimentar fazer a troca de nossa view antiga e ativar a nossa nova view. Podemos fazer isso em *urls.py*:

### lists/urls.py

```

[...]
url(r'^new$', views.new_list2, name='new_list'),

```

Também devemos remover `unittest.skip` de nossa classe de testes integrados para ver se o nosso novo código para proprietários de lista realmente funciona:

## lists/tests/test\_views.py (ch20|033)

```
class NewListViewIntegratedTest(TestCase):

 def test_can_save_a_POST_request(self):
 [...]

 def test_list_owner_is_saved_if_user_is_authenticated(self):
 [...]
 self.assertEqual(list_.owner, user)
```

O que acontece quando executamos os nossos testes? Oh, não!

```
ERROR: test_list_owner_is_saved_if_user_is_authenticated
[...]
ERROR: test_can_save_a_POST_request
[...]
ERROR: test_redirects_after_POST
(lists.tests.test_views.NewListViewIntegratedTest)
File "/../superlists/lists/views.py", line 30, in new_list2
 return redirect(list_)
[...]
TypeError: argument of type 'NoneType' is not iterable
```

FAILED (errors=3)

Eis uma lição importante para aprender sobre isolamento de testes: ele poderá ajudar você a obter um bom design para as camadas individuais, mas não verificará automaticamente a integração *entre* as suas camadas.

O que aconteceu nesse caso é que a view estava esperando que o formulário devolvesse um item da lista:

## lists/views.py

```
list_ = form.save(owner=request.user)
return redirect(list_)
```

No entanto, nós nos esquecemos de fazê-la devolver algo:

## lists/forms.py

```
def save(self, owner):
```

```
if owner.is_authenticated:
 List.create_new(first_item_text=self.cleaned_data['text'], owner=owner)
else:
 List.create_new(first_item_text=self.cleaned_data['text'])
```

## Pensando nas interações entre as camadas como sendo “contratos”

Em última instância, mesmo que não estivéssemos escrevendo nada além de testes de unidade isolados, nossos testes funcionais teriam identificado esse deslize em particular. Contudo, de modo ideal, iríamos querer que o nosso ciclo de feedback fosse mais rápido – testes funcionais podem demorar alguns minutos para executar, ou até mesmo algumas horas, depois que sua aplicação começar a crescer. Há alguma maneira de evitar esse tipo de problema antes que ele aconteça?

Metodicamente, o modo de fazer isso é pensar na interação entre suas camadas em termos de contratos. Sempre que simulamos o comportamento de uma camada, precisamos tomar nota mentalmente de que agora há um contrato implícito entre as camadas, e que um mock em uma camada provavelmente se traduzirá em um teste na camada abaixo.

Eis a parte do contrato que estava faltando:

### lists/tests/test\_views.py

```
@patch('lists.views.redirect')
def test_redirects_to_form_returned_object_if_form_valid(
 self, mock_redirect, mockNewListForm
):
 mock_form = mockNewListForm.return_value
 mock_form.is_valid.return_value = True

 response = new_list2(self.request)

 self.assertEqual(response, mock_redirect.return_value)
 mock_redirect.assert_called_once_with(mock_form.save.return_value) ❶
```

- ❶ A função `form.save` simulada está devolvendo um objeto, que esperamos que nossa view use.

## Identificando contratos implícitos

Vale a pena rever cada um dos testes em `NewListViewUnitTest` e ver o que cada mock diz sobre o contrato implícito:

### `lists/tests/test_views.py`

```
def test_passes_POST_data_to_NewListForm(self, mockNewListForm):
 [...]
 mockNewListForm.assert_called_once_with(data=self.request.POST) ❶
```

```
def test_saves_form_with_owner_if_form_valid(self, mockNewListForm):
 mock_form = mockNewListForm.return_value
 mock_form.is_valid.return_value = True ❷
 new_list2(self.request)
 mock_form.save.assert_called_once_with(owner=self.request.user) ❸
```

```
def test_does_not_save_if_form_invalid(self, mockNewListForm):
 [...]
 mock_form.is_valid.return_value = False ❷
 [...]
```

```
@patch('lists.views.redirect')
def test_redirects_to_form_returned_object_if_form_valid(
 self, mock_redirect, mockNewListForm
):
 [...]
 mock_redirect.assert_called_once_with(mock_form.save.return_value) ❹
```

```
@patch('lists.views.render')
def test_renders_home_template_with_form_if_form_invalid(
 [...]
```

- ❶ Precisamos inicializar o nosso formulário passando-lhe uma

requisição POST como dado.

- ② Ele deve ter uma função `is_valid()` que devolva `True` ou `False` de modo apropriado, com base no dado de entrada.
- ③ O formulário deve ter um método `.save` que aceitará um `request.user` – esse poderá ou não ser um usuário logado – e deverá lidar com ele de modo apropriado.
- ④ O método `.save` do formulário deve devolver um novo objeto de lista, para a qual a nossa view redirecionará o usuário.

Se observarmos os nossos testes de formulários, veremos que, na verdade, somente o item ③ é testado explicitamente. Nos itens ① e ②, tivemos sorte – são recursos default de um `ModelForm` do Django e estão cobertos pelos nossos testes da classe-pai `ItemForm`.

Porém, a cláusula de número ④ do contrato conseguiu escorregar por entre os dedos.



Quando usamos TDD Outside-In com testes isolados, devemos manter o controle dos pressupostos implícitos de cada teste acerca do contrato que a próxima camada deve implementar, e lembrar de testar cada um deles individualmente, mais tarde. Você poderia usar a sua folha de rascunho para isso, ou poderia criar um teste placeholder com um `self.fail`.

## Corrigindo o que deixamos passar

Vamos acrescentar um novo teste em que o nosso formulário deve devolver a nova lista salva:

### `lists/tests/test_forms.py` (ch20I038-1)

```
@patch('lists.forms.List.create_new')
def test_save_returns_new_list_object(self, mock_List_create_new):
 user = Mock(is_authenticated=True)
 form = NewListForm(data={'text': 'new item text'})
 form.is_valid()
 response = form.save(owner=user)
 self.assertEqual(response, mock_List_create_new.return_value)
```

Na verdade, esse é um bom exemplo – temos um contrato implícito

com `List.create_new`; queremos que ele devolva o novo objeto de lista. Vamos adicionar um teste placeholder para isso:

## lists/tests/test\_models.py (ch20l038-2)

```
class ListModelTest(TestCase):
 [...]

 def test_create_returns_new_list_object(self):
 self.fail()
```

Assim, temos uma falha de teste nos dizendo para corrigir o save do formulário:

```
AssertionError: None != <MagicMock name='create_new()' id='139802647565536'>
FAILED (failures=2, errors=3)
```

Assim:

## lists/forms.py (ch20l039-1)

```
class NewListForm(ItemForm):

 def save(self, owner):
 if owner.is_authenticated:
 return List.create_new(first_item_text=self.cleaned_data['text'],
owner=owner)
 else:
 return List.create_new(first_item_text=self.cleaned_data['text'])
```

É um começo; agora devemos nos voltar para o nosso teste placeholder:

```
[...]
FAIL: test_create_returns_new_list_object
self.fail()
AssertionError: None
```

```
FAILED (failures=1, errors=3)
```

Vamos implementá-lo:

## lists/tests/test\_models.py (ch20l039-2)

```
def test_create_returns_new_list_object(self):
```



```
 returned = List.create_new(first_item_text='new item text')
 new_list = List.objects.first()
 self.assertEqual(returned, new_list)
```

...

```
AssertionError: None != <List: List object>
```

Adicionemos o valor de retorno:

## lists/models.py (ch20I039-3)

```
@staticmethod
def create_new(first_item_text, owner=None):
 list_ = List.objects.create(owner=owner)
 Item.objects.create(text=first_item_text, list=list_)
 return list_
```

Com isso, temos uma suíte de testes passando por completo:

```
$ python manage.py test lists
[...]
Ran 50 tests in 0.169s
```

OK

## Mais um teste

Esse é o nosso código para salvar proprietários de lista, orientado a testes por inteiro, e funcionando. Contudo, o nosso teste funcional ainda não está passando:

```
$ python manage.py test functional_tests.test_my_lists
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate
element: Reticulate splines
```

Isso ocorre porque temos um último recurso a ser implementado, que é o atributo `.name` nos objetos de lista. Novamente, podemos usar o teste e o código do último capítulo:

## lists/tests/test\_models.py (ch20I040)

```
def test_list_name_is_first_item_text(self):
 list_ = List.objects.create()
 Item.objects.create(list=list_, text='first item')
```

```
Item.objects.create(list=list_, text='second item')
self.assertEqual(list_.name, 'first item')
```

(Mais uma vez, como esse é um teste da camada de modelo, tudo bem se usarmos o ORM. Seria possível escrever esse teste usando mocks, porém não haveria motivos para isso.)

## lists/models.py (ch20l041)

```
@property
def name(self):
 return self.item_set.first().text
```

O resultado é um FT que passa!

```
$ python manage.py test functional_tests.test_my_lists
```

```
Ran 1 test in 21.428s
```

OK

## Organizando o código: o que deve ser mantido em nossa suíte de testes integrados

Agora que tudo está funcionando, podemos remover alguns testes redundantes e decidir se queremos manter alguns dos antigos testes integrados.

## Removendo código redundante da camada de formulários

Podemos nos livrar do teste para o antigo método save de ItemForm:

### lists/tests/test\_forms.py

```
--- a/lists/tests/test_forms.py
+++ b/lists/tests/test_forms.py
@@ -23,14 +23,6 @@ class ItemFormTest(TestCase):

 self.assertEqual(form.errors['text'], [EMPTY_ITEM_ERROR])
```

```

- def test_form_save_handles_saving_to_a_list(self):
- list_ = List.objects.create()
- form = ItemForm(data={'text': 'do me'})
- new_item = form.save(for_list=list_)
- self.assertEqual(new_item, Item.objects.first())
- self.assertEqual(new_item.text, 'do me')
- self.assertEqual(new_item.list, list_)
-

```

Em nosso código propriamente dito, podemos nos livrar de dois métodos save redundantes em *forms.py*:

## lists/forms.py

```

--- a/lists/forms.py
+++ b/lists/forms.py
@@ -22,11 +22,6 @@ class ItemForm(forms.models.ModelForm):

 self.fields['text'].error_messages['required'] = EMPTY_ITEM_ERROR

- def save(self, for_list):
- self.instance.list = for_list
- return super().save()
-
-

class NewListForm(ItemForm):

@@ -52,8 +47,3 @@ class ExistingListItemForm(ItemForm):

 e.error_dict = {'text': [DUPLICATE_ITEM_ERROR]}
 self._update_errors(e)
-
-
- def save(self):
- return forms.models.ModelForm.save(self)
-

```

## Removendo a antiga implementação da view

Agora podemos remover totalmente a antiga view `new_list` e

renomear new\_list2 para new\_list:

## lists/tests/test\_views.py

```
-from lists.views import new_list, new_list2
```

```
+from lists.views import new_list
```

```
class HomePageTest(TestCase):
```

```
@@ -75,7 +75,7 @@ class NewListViewIntegratedTest(TestCase):
```

```
 request = HttpRequest()
```

```
 request.user = User.objects.create(email='a@b.com')
```

```
 request.POST['text'] = 'new list item'
```

```
- new_list2(request)
```

```
+ new_list(request)
```

```
 list_ = List.objects.first()
```

```
 self.assertEqual(list_.owner, request.user)
```

```
@@ -91,21 +91,21 @@ class NewListViewUnitTest(unittest.TestCase):
```

```
 def test_passes_POST_data_to_NewListForm(self, mockNewListForm):
```

```
- new_list2(self.request)
```

```
+ new_list(self.request)
```

```
[.. muito mais]
```

## lists/urls.py

```
--- a/lists/urls.py
```

```
+++ b/lists/urls.py
```

```
@@ -3,7 +3,7 @@ from django.conf.urls import url
```

```
from lists import views
```

```
urlpatterns = [
```

```
- url(r'^new$', views.new_list2, name='new_list'),
```

```
+ url(r'^new$', views.new_list, name='new_list'),
```

```
 url(r'^(\d+)/$', views.view_list, name='view_list'),
```

```
 url(r'^users/(.+)/$', views.my_lists, name='my_lists'),
```

```
]
```

## lists/views.py (ch20l047)

```
def new_list(request):
```

```
form = NewListForm(data=request.POST)
if form.is_valid():
 list_ = form.save(owner=request.user)
 [...]
```

Faça uma verificação rápida para ver se todos os testes continuam passando:

OK

## Removendo código redundante da camada de formulários

Por fim, tivemos que decidir sobre o que deve ser mantido (se houver) em nossa suíte de testes integrados.

Uma opção é jogar tudo fora e supor que os FTs identificarão qualquer problema de integração. Isso é perfeitamente válido.

Por outro lado, vimos como os testes integrados podem avisá-lo quando você cometer pequenos erros ao integrar suas camadas. Poderíamos manter somente alguns testes como “verificações de sanidade” para termos um ciclo de feedback mais rápido.

Que tal os três testes a seguir?

### lists/tests/test\_views.py (ch20|048)

```
class NewListViewIntegratedTest(TestCase):

 def test_can_save_a_POST_request(self):
 self.client.post('/lists/new', data={'text': 'A new list item'})
 self.assertEqual(Item.objects.count(), 1)
 new_item = Item.objects.first()
 self.assertEqual(new_item.text, 'A new list item')

 def test_for_invalid_input_doesnt_save_but_shows_errors(self):
 response = self.client.post('/lists/new', data={'text': ''})
 self.assertEqual(List.objects.count(), 0)
 self.assertContains(response, escape(EMPTY_ITEM_ERROR))
```

```
def test_list_owner_is_saved_if_user_is_authenticated(self):
 user = User.objects.create(email='a@b.com')
 self.client.force_login(user)
 self.client.post('/lists/new', data={'text': 'new item'})
 list_ = List.objects.first()
 self.assertEqual(list_.owner, user)
```

Se você pretende manter algum teste de nível intermediário, gosto desses três porque dão a impressão de estarem fazendo a maior parte das tarefas de “integração”: testam a pilha completa, desde a requisição até o banco de dados, e incluem os três casos de uso mais importantes de nossa view.

## **Conclusões: quando escrever testes isolados versus testes integrados**

As ferramentas de teste do Django facilitam bastante compor testes integrados rapidamente. O executor de testes prestativamente cria uma versão rápida, em memória, de seu banco de dados, e reinicia-o para você entre cada teste. A classe `TestCase` e o cliente de teste facilitam testar suas views, desde verificar se os objetos do banco de dados foram modificados, confirmar se seus mapeamentos de URL funcionam e inspecionar a renderização dos templates. Isso permite que você comece a trabalhar com os testes facilmente e tenha uma boa abrangência em toda a sua pilha.

Por outro lado, esses tipos de testes integrados não necessariamente proporcionam todas as vantagens que um teste de unidade e um TDD Outside-In rigorosos devem oferecer no que diz respeito ao design.

Observando o exemplo deste capítulo, compare o código que tínhamos antes e depois:

Antes:

```
def new_list(request):
 form = ItemForm(data=request.POST)
 if form.is_valid():
 list_ = List()
```

```
if not isinstance(request.user, AnonymousUser):
 list_.owner = request.user
list_.save()
form.save(for_list=list_)
return redirect(list_)
else:
 return render(request, 'home.html', {"form": form})
```

Depois:

```
def new_list(request):
 form = NewListForm(data=request.POST)
 if form.is_valid():
 list_ = form.save(owner=request.user)
 return redirect(list_)
 return render(request, 'home.html', {'form': form})
```

Se não tivéssemos nos empenhado em seguir o caminho do isolamento, teríamos nos dado ao trabalho de refatorar a função de view? Sei que não fiz isso na primeira versão preliminar deste livro. Gostaria de pensar que teria feito isso “na vida real”, mas é difícil ter certeza. No entanto, escrever testes isolados realmente faz com que você fique bem ciente dos pontos em que estão as complexidades de seu código.

## **Deixe que a complexidade seja o seu guia**

Eu diria que o ponto em que os testes isolados começam a valer a pena tem a ver com a complexidade. O exemplo deste livro é extremamente simples, portanto, em geral, não tem valido a pena até agora. Mesmo no exemplo deste capítulo, posso me convencer de que realmente não *precisava* escrever esses testes isolados.

Contudo, depois que uma aplicação se torna um pouco mais complexa – se ela começar a adquirir mais camadas entre as views e os modelos, se você se pegar escrevendo mais métodos auxiliares ou se estiver escrevendo suas próprias classes –, provavelmente você terá vantagens se escrever testes mais isolados.

## Você deve ter ambos?

Já temos a nossa suíte de testes funcionais, que servirá para nos informar caso venhamos a cometer algum erro na integração das diferentes partes de nosso código. Escrever testes isolados pode nos ajudar a ter um design melhor para o nosso código e verificar se ele está correto de modo mais detalhado. Uma camada intermediária de testes de integração teria algum propósito adicional?

Acho que a resposta, possivelmente, é sim, se eles puderem proporcionar um ciclo mais rápido de feedback e ajudar a identificar mais claramente quais são os problemas de integração que afligem você – seus tracebacks poderão oferecer melhores informações de depuração em comparação com as que você obteria com um teste funcional, por exemplo.

Talvez haja inclusive um cenário em que eles serão criados como uma suíte de testes separada – você poderia ter uma suíte com testes de unidade rápidos e isolados, que nem sequer usem `manage.py` porque não precisam de nada relacionado à limpeza e à desconexão com o banco de dados, oferecidas pelo executor de testes do Django, e então a camada intermediária que utilize o Django e, por fim, a camada de testes funcionais que, digamos, conversa com um servidor de staging. Talvez valesse a pena se cada camada proporcionasse vantagens incrementais.

É uma questão de avaliar. Espero que, ao ler este capítulo, eu tenha dado a você uma noção de quais são as negociações de custo-benefício. Mais discussões sobre esse assunto estão no Capítulo 26.

## Em frente!

Estamos satisfeitos com a nossa nova versão, portanto vamos levá-la para o branch master:

```
$ git add .
```

```
$ git commit -m "add list owners via forms. more isolated tests"
```



```
$ git checkout master
```

```
$ git checkout -b master-noforms-noisolation-bak # backup opcional
```

```
$ git checkout master
```

```
$ git reset --hard more-isolation # reset do master para o nosso branch.
```

Nesse ínterim – esses FTs estão irritantemente consumindo muito tempo para executar. Pergunto-me se há algo que possamos fazer a esse respeito.

## **Sobre os prós e contras dos diferentes tipos de testes e sobre o desacoplamento do código de ORM**

## *Testes funcionais*

- Oferecem a melhor garantia de que a sua aplicação de fato funciona corretamente, do ponto de vista do usuário.
- No entanto: têm um ciclo de feedback mais lento.
- E não necessariamente ajudam você a escrever um código mais limpo.

*Testes integrados (dependentes, por exemplo, de ORM ou do Django Test Client)*

- São rápidos para serem escritos.
- São fáceis de entender.
- Avisarão você acerca de qualquer problema de integração.
- No entanto: nem sempre levam a um bom design (isso cabe a você!).
- E geralmente são mais lentos que os testes isolados.

### *Testes isolados (“com simulação”)*

- Envolvem o trabalho mais árduo.
- Podem ser mais difíceis de ler e de entender.
- No entanto: são os melhores para orientar você em direção a um design melhor.
- E são os mais rápidos para executar.

### *Desacoplando a nossa aplicação do código de ORM*

Uma das consequências de se esforçar para escrever testes isolados é que nos vemos forçados a remover o código de ORM de locais como views e formulários, ocultando-o por trás de funções e métodos auxiliares. Isso pode ser vantajoso no que concerne ao desacoplamento de sua aplicação do ORM, mas também porque simplesmente deixa o seu código mais legível. Como tudo o mais, é uma questão de avaliar se o esforço adicional vale a pena, em circunstâncias específicas.

---

1 Poderia ser facilmente uma função independente, mas deixá-la na classe de modelo é uma boa maneira de controlar o local em que ela permanecerá, e dará uma pista melhor sobre o que ela fará.



# CAPÍTULO 24

## Integração Contínua (CI)

À medida que o nosso site cresce, demoramos cada vez mais para executar todos os testes funcionais. Se isso continuar, o perigo é de deixarmos de nos importar com eles.

Em vez de permitir que isso aconteça, podemos automatizar a execução dos testes funcionais configurando um servidor de CI (Continuous Integration, ou Integração Contínua). Desse modo, no desenvolvimento cotidiano, podemos simplesmente executar o FT no qual estamos trabalhando no momento, e contar com o servidor de CI para executar todos os testes de modo automático e nos informar se causamos falhas em algo acidentalmente. Os testes de unidade devem permanecer rápidos o bastante de modo que continuem sendo executados a intervalos de alguns segundos.

O servidor de CI preferido atualmente se chama Jenkins. Tem um pouco de Java, falha algumas vezes, é um pouco feio, mas é o que todos usam, e tem um ótimo ecossistema de plugins, portanto vamos instalá-lo e executá-lo.

### Instalando o Jenkins

Há vários serviços de CI hospedado no mercado que, essencialmente, oferecem um servidor Jenkins pronto para ser usado. Eu deparei com o Sauce Labs, Travis, Circle-CI, ShiningPanda, e, provavelmente, há muitos outros. Contudo, vou supor que instalaremos tudo em um servidor controlado por nós.



Instalar o Jenkins no mesmo servidor que os nossos servidores de staging ou de produção não é uma boa ideia. Além disso, queremos que o Jenkins seja capaz de reiniciar o servidor de staging!

Instalaremos a versão mais recente a partir do repositório apt oficial do Jenkins, pois o default do Ubuntu ainda contém alguns bugs irritantes no suporte para localidades/unicode, e também não se configura para ouvir a internet pública, por padrão:

```
root@server:$ wget -q -O - https://pkg.jenkins.io/debian/jenkins-ci.org.key |\
 apt-key add -
root@server:$ echo deb http://pkg.jenkins.io/debian-stable binary/ | tee \
 /etc/apt/sources.list.d/jenkins.list
root@server:$ apt-get update
root@server:$ apt-get install jenkins
```

(Instruções obtidas do site do Jenkins em <https://wiki.jenkins-ci.org/display/JENKINS/Installing+Jenkins+on+Ubuntu>.)

Aproveitando a ocasião, instalaremos mais algumas dependências:

```
root@server:$ apt-get install firefox python3-venv xvfb
e, para construir o fabric3:
root@server:$ apt-get install build-essential libssl-dev libffi-dev
```

Faremos o download, descompactaremos e instalaremos o geckodriver também (estava na v0.17 quando escrevi este livro, mas substitua-o pela versão mais recente quando você estiver lendo-o):

```
root@server:$ wget https://github.com/mozilla/geckodriver/releases\
/download/v0.17.0/geckodriver-v0.17.0-linux64.tar.gz
root@server:$ tar -xvzf geckodriver-v0.17.0-linux64.tar.gz
root@server:$ mv geckodriver /usr/local/bin
root@server:$ geckodriver --version
geckodriver 0.17.0
```

## Acrescentando um pouco de área para swap

O Jenkins consome bastante memória, e, se você estiver executando-o em uma VM pequena, com menos de poucos gigas de RAM, provavelmente perceberá que ele começará a falhar por falta de memória, a menos que você acrescente um pouco de área para swap:

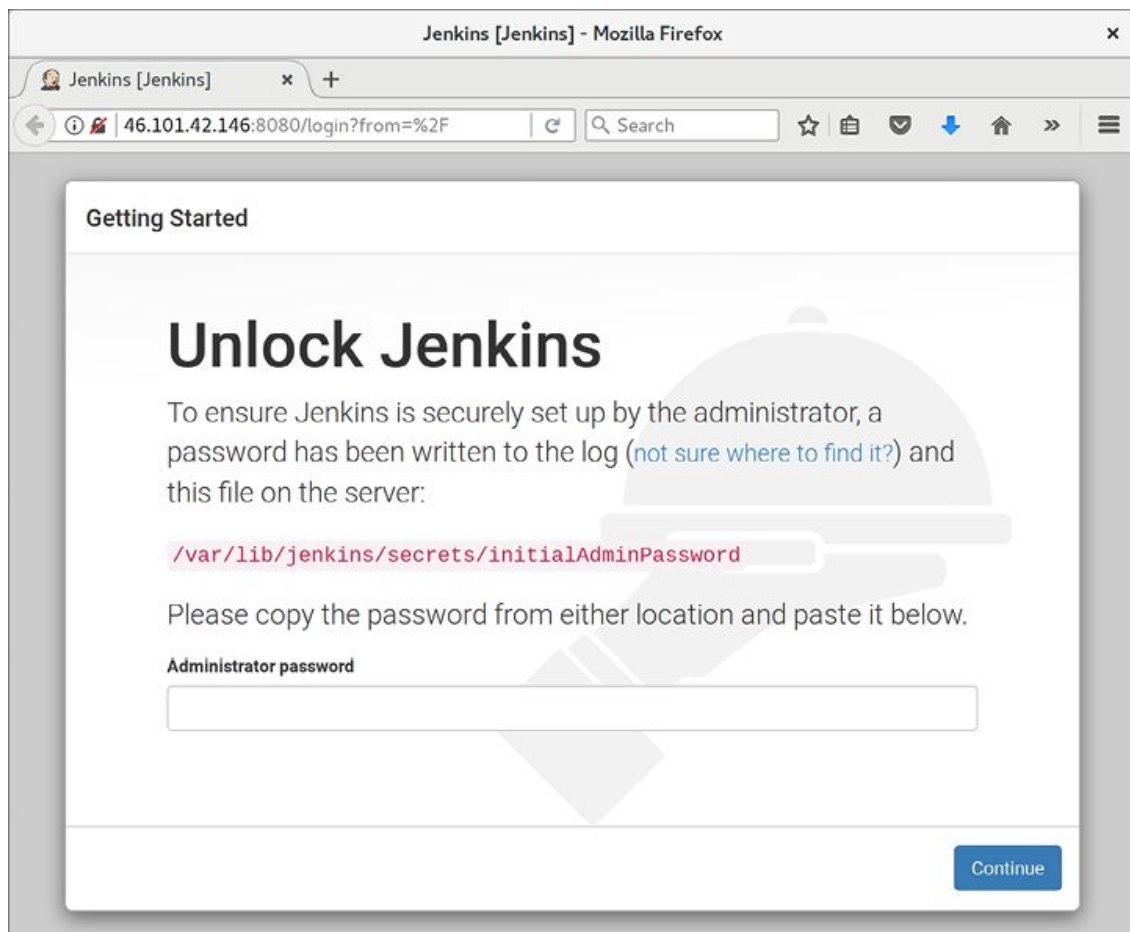


```
$ fallocate -l 4G /swapfile
$ mkswap /swapfile
$ chmod 600 /swapfile
$ swapon /swapfile
```

Isso deve bastar.

## Configurando o Jenkins

Agora você deve ser capaz de acessar o Jenkins no URL/IP de seu servidor na porta 8080, e verá algo como o que mostra a Figura 24.1.



*Figura 24.1 – Tela de desbloqueio do Jenkins.*

## Desbloqueio inicial

A tela de desbloqueio está nos dizendo para ler um arquivo de disco

a fim de desbloquear o servidor na primeira vez em que ele for usado. Acessei um terminal e exibi o arquivo assim:

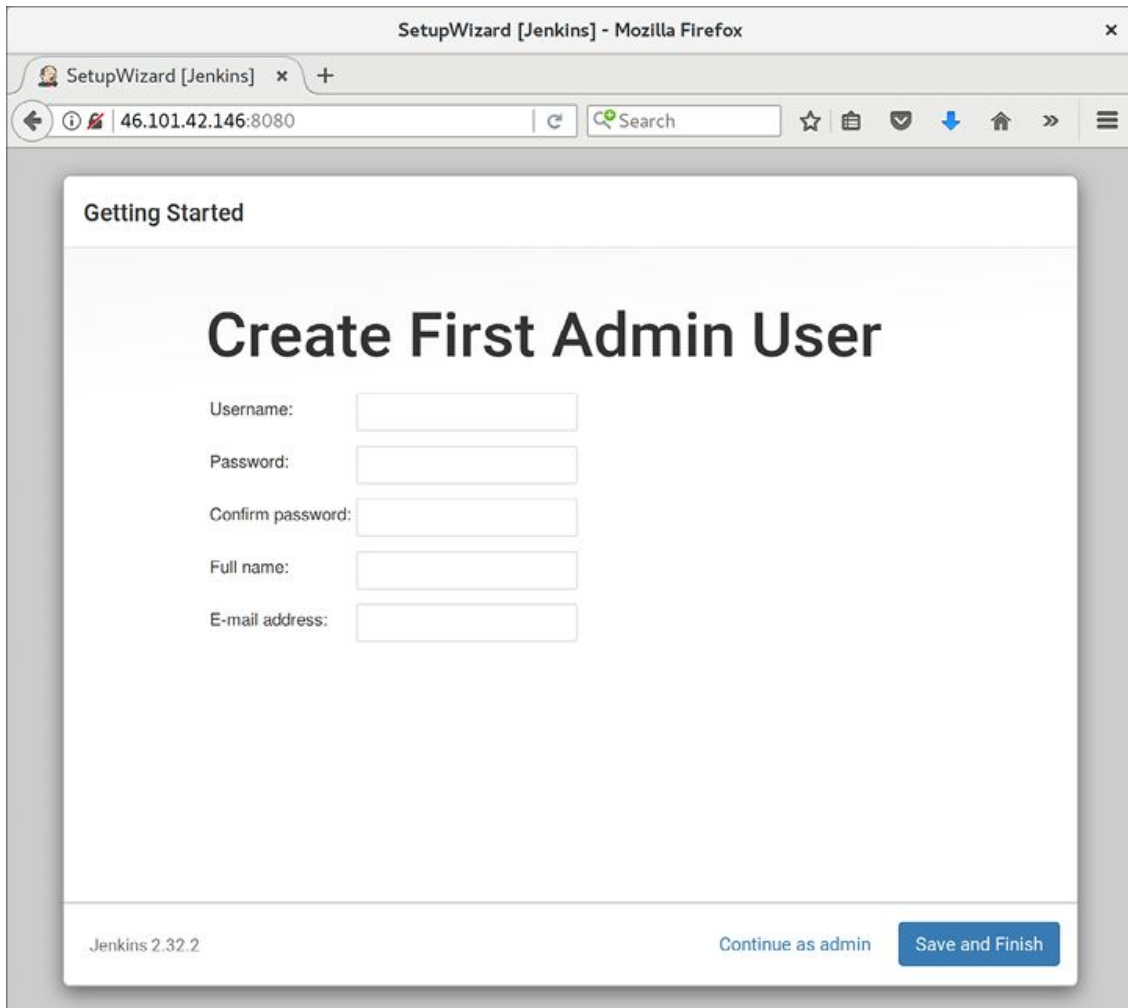
```
root@server$ cat /var/lib/jenkins/secrets/initialAdminPassword
```

## **Plugins sugeridos, por enquanto**

Em seguida, teremos a opção de escolher plugins “sugeridos”. Os plugins sugeridos são apropriados, por enquanto. (Como um nerd que se respeita, o nosso instinto é pressionar “customize” (personalizar) imediatamente, e foi isso que eu fiz na primeira vez, mas o fato é que essa tela não nos dará o que queremos. Não se preocupe: acrescentaremos mais alguns plugins depois.)

## **Configurando o usuário administrador**

A seguir, configure um nome de usuário e uma senha para login no Jenkins; veja a Figura 24.2.



*Figura 24.2 – Configuração do usuário administrador no Jenkins.*  
Depois de fazer o login, devemos ver uma tela de boas-vindas (Figura 24.3).

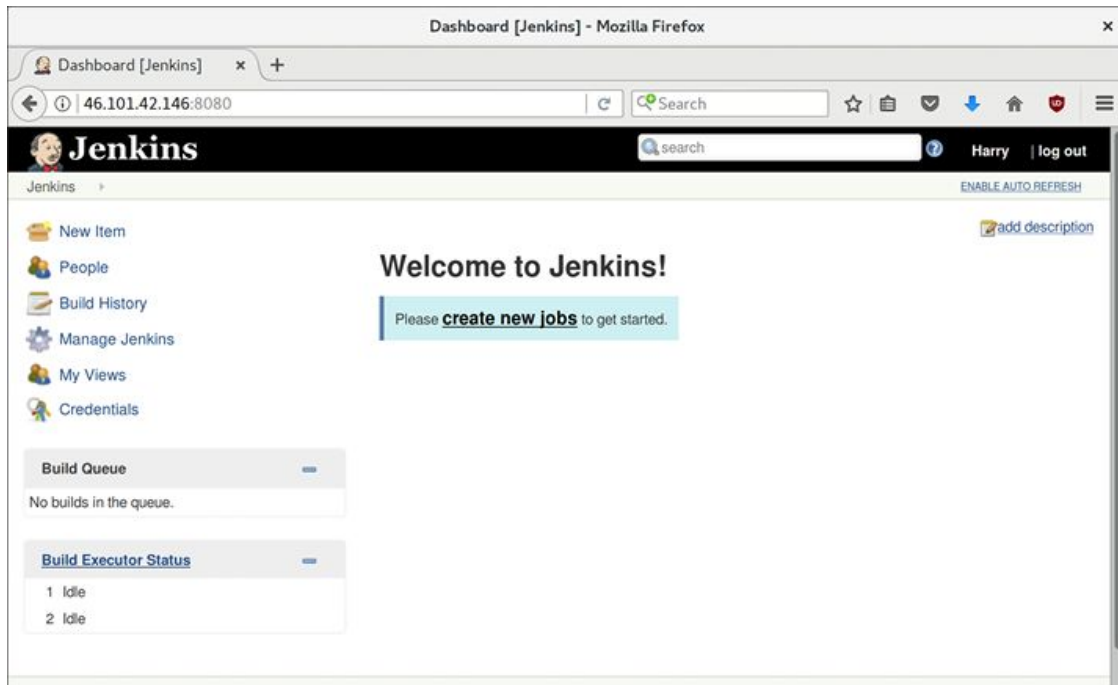


Figura 24.3 – Um mordomo – que singular.

## Adicionando plugins

Siga os links para *Manage Jenkins* > *Manage Plugins* > *Available* (Administrar o Jenkins > Administrar plugins > Disponíveis).

Queremos plugins para:

- *ShiningPanda*
- *Xvfb*

Pressione o botão para instalar (Figura 24.4).

## Informando ao Jenkins o local para encontrar o Python 3 e o Xvfb

Precisamos informar ao plugin ShiningPanda o local em que o Python 3 está instalado (geralmente é em `/usr/bin/python3`, mas você pode verificar com `which python3`):

- *Manage Jenkins* > *Global Tool Configuration* (Administrar o Jenkins > Configuração de ferramentas globais)

- *Python > Python installations > Add Python* (Python > Instalações de Python > Adicionar Python – veja a Figura 24.5; é seguro ignorar a mensagem de advertência)
- *Xvfb installation > Add Xvfb installation* (Instalação do Xvfb > Adicionar instalação do Xvfb); insira `/usr/bin` como o diretório de instalação

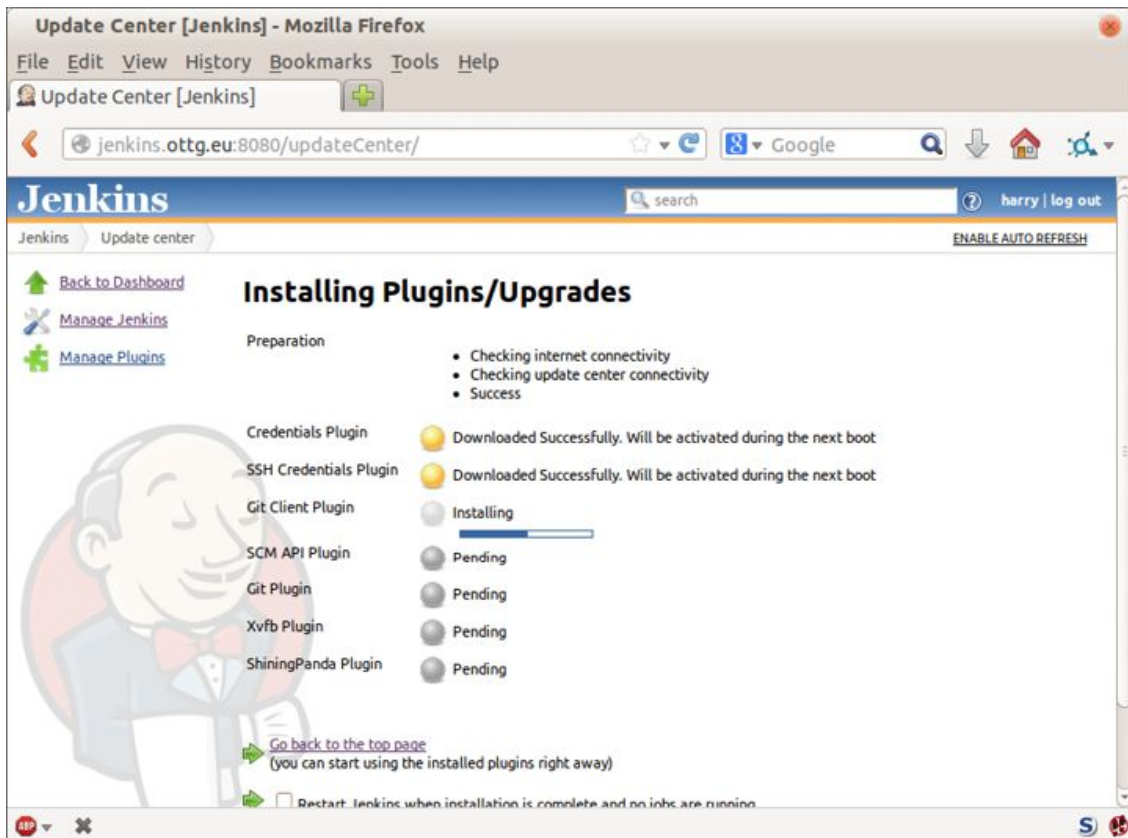


Figura 24.4 – Instalando plugins...

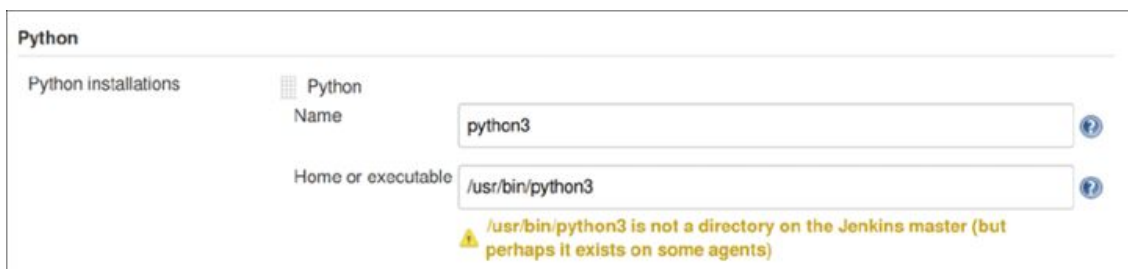


Figura 24.5 – Onde foi que eu deixei aquele Python?

## Concluindo com HTTPS

Para terminar de proteger a sua instância de Jenkins, você deve configurar o HTTPS, fazendo o HTTPS do nginx usar um certificado autoassinado e fazer proxy de requisições da porta 443 para a porta 8080. Então você poderá até mesmo bloquear a porta 8080 no firewall. Não entrarei nos detalhes sobre isso agora, mas eis alguns links com instruções que achei úteis:

- Guia oficial de instalação do Jenkins no Ubuntu (<https://wiki.jenkins-ci.org/display/JENKINS/Installing+Jenkins+on+Ubuntu>)
- Como criar um certificado SSL autoassinado (<https://www.digitalocean.com/community/tutorials/how-to-create-an-ssl-certificate-on-nginx-for-ubuntu-14-04>)
- Como redirecionar HTTP para HTTPS (<http://serverfault.com/questions/250476/howto-force-or-redirect-to-ssl-in-nginx#424016>)

## Configurando o nosso projeto

Agora que temos o Jenkins básico configurado, vamos criar o nosso projeto:

- Pressione o botão New Item (Novo item).
- Digite *Superlists* no nome, então selecione “Freestyle project” (Projeto em estilo livre) e aperte OK.
- Adicione o repositório do Git, como vemos na Figura 24.6.

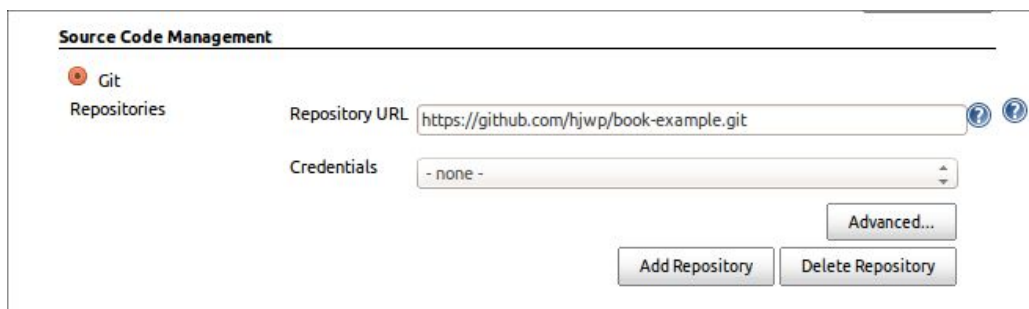
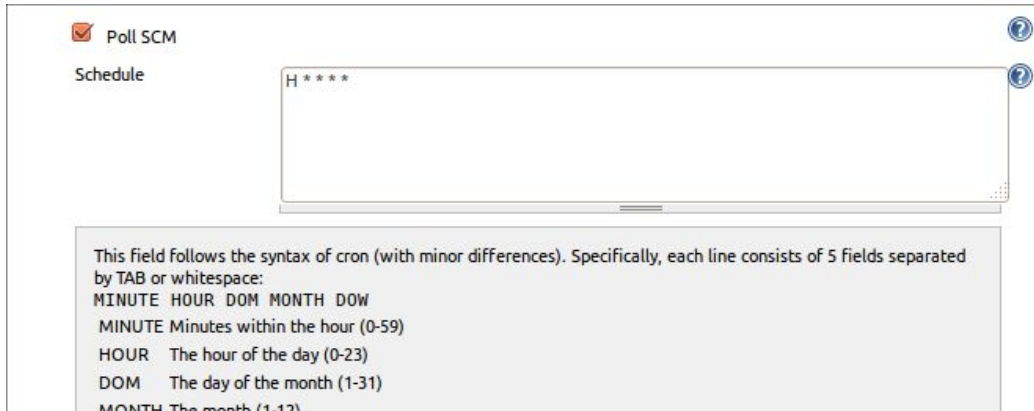
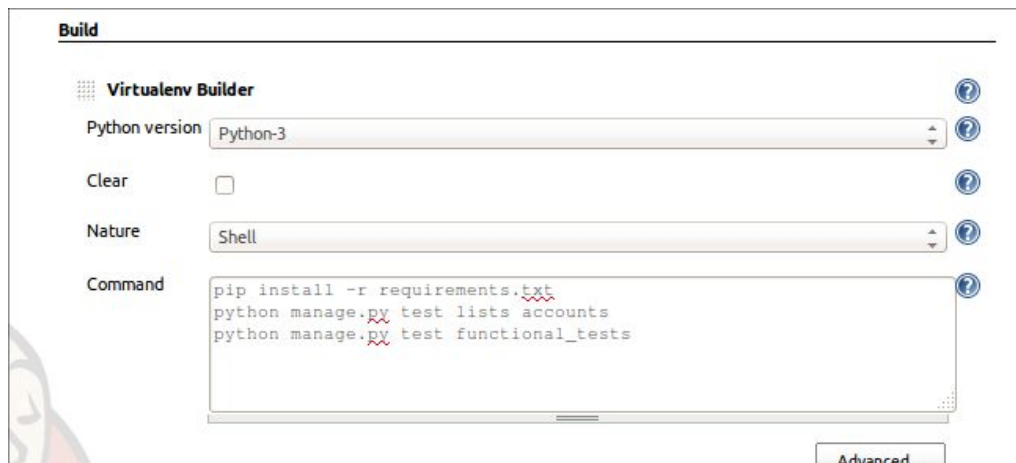


Figura 24.6 – Obtenha do Git.

- Configure-o para fazer polling a cada hora (Figura 24.7; consulte o texto de ajuda ali – há muitas outras opções para os modos de disparar construções).
- Execute os testes em um virtualenv do Python 3.
- Execute os testes de unidade e os testes funcionais separadamente. Veja a Figura 24.8.



*Figura 24.7 – Faça polling do GitHub para ver se houve mudanças.*



*Figura 24.8 – Passos para a construção do virtualenv.*

## Primeira construção!

Pressione “Build Now” (Construir agora) e observe a “Console Output” (Saída no console). Você deverá ver algo como:

```
Started by user harry
Building in workspace /var/lib/jenkins/jobs/Superlists/workspace
```

Fetching changes from the remote Git repository  
Fetching upstream changes from https://github.com/hjwp/book-example.git  
Checking out Revision d515acebf7e173f165ce713b30295a4a6ee17c07  
(origin/master)  
[workspace] \$ /bin/sh -xe /tmp/shiningpanda7260707941304155464.sh  
+ pip install -r requirements.txt  
Requirement already satisfied (use --upgrade to upgrade): Django==1.11 in  
/var/lib/jenkins/shiningpanda/jobs/ddc1aed1/virtualenvs/d41d8cd9/lib/python3.3/s  
ite-packages  
(from -r requirements.txt (line 1))

Requirement already satisfied (use --upgrade to upgrade): gunicorn==17.5 in  
/var/lib/jenkins/shiningpanda/jobs/ddc1aed1/virtualenvs/d41d8cd9/lib/python3.3/s  
ite-packages  
(from -r requirements.txt (line 3))  
Downloading/unpacking requests==2.0.0 (from -r requirements.txt (line 4))  
Running setup.py egg\_info for package requests

Installing collected packages: requests  
Running setup.py install for requests

Successfully installed requests  
Cleaning up...  
+ python manage.py test lists accounts

.....  
-----  
Ran 67 tests in 0.429s

OK  
Creating test database for alias 'default'...  
Destroying test database for alias 'default'...  
+ python manage.py test functional\_tests  
EEEEEE

=====  
=====

ERROR: functional\_tests.test\_layout\_and\_styling (unittest.loader.\_FailedTest)  
-----

ImportError: Failed to import test module:  
functional\_tests.test\_layout\_and\_styling  
[...]  
ImportError: No module named 'selenium'



Ran 6 tests in 0.001s

FAILED (errors=6)

Build step 'Virtualenv Builder' marked build as failure

Ah. Precisamos do Selenium em nosso virtualenv.

Vamos acrescentar uma instalação manual do Selenium em nossos passos de construção:

```
pip install -r requirements.txt
python manage.py test accounts lists
pip install selenium
python manage.py test functional_tests
```



Algumas pessoas gostam de usar um arquivo chamado *test-requirements.txt* a fim de especificar os pacotes necessários aos testes, mas não para a aplicação principal.

Pressione “Build Now” novamente.

Em seguida, uma de duas opções ocorrerá. Você verá algumas mensagens de erro como estas na saída de seu console:

```
self.browser = webdriver.Firefox()
[...]
selenium.common.exceptions.WebDriverException: Message: 'The browser
appears to
have exited before we could connect. The output was: b"\n(process:19757):
GLib-CRITICAL **: g_slice_set_config: assertion '\sys_page_size == 0'
failed\nError: no display specified\n"
[...]
selenium.common.exceptions.WebDriverException: Message: connection
refused
```

Ou, possivelmente, sua construção apenas ficará travada (o que aconteceu comigo pelo menos uma vez). Isso se deve ao fato de o Firefox não poder iniciar, pois não tem um display no qual executar.

## Configurando um display virtual para que FTs

## possam executar em modo headless

Como podemos ver a partir do traceback, o Firefox é incapaz de iniciar porque o servidor não tem um display.

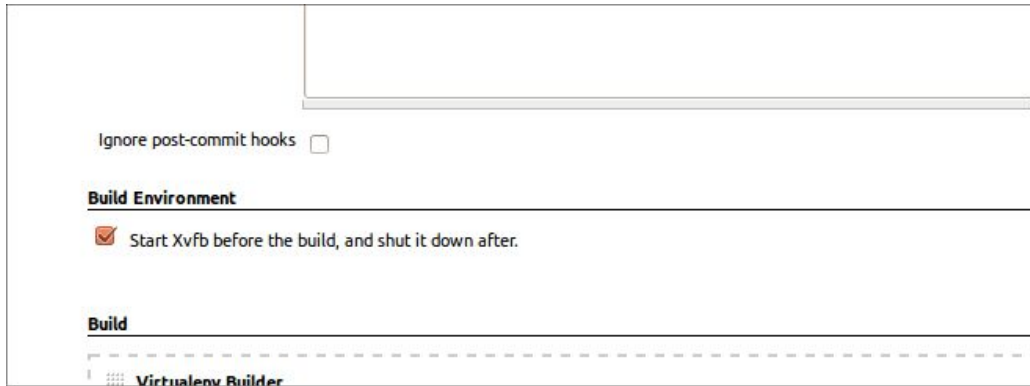
Há duas formas de lidar com esse problema. A primeira é passar a usar um navegador headless, como o PhantomJS ou o SlimerJS. Definitivamente essas ferramentas têm o seu lugar – são mais rápidas, para começar –, mas também apresentam desvantagens. A primeira é que elas não são navegadores web “reais”, portanto não há como ter certeza de que você capturará todas as idiossincrasias e comportamentos dos verdadeiros navegadores que seus usuários utilizarão. A segunda é que eles podem se comportar de modo bem diferente no Selenium e, com frequência, exigem que parte do código dos FTs seja reescrita.



Eu consideraria o uso de navegadores headless como uma ferramenta “somente para desenvolvimento”, para agilizar a execução dos FTs na máquina do desenvolvedor, enquanto os testes no servidor de CI utilizarão navegadores reais.

A alternativa é configurar um display virtual: fazemos o servidor fingir que tem uma tela associada a ele; desse modo o Firefox ficará satisfeito. Há algumas ferramentas no mercado para isso; usaremos uma chamada “Xvfb” (X Virtual Framebuffer)<sup>1</sup> porque é fácil de instalar e usar, e porque tem um plugin conveniente para Jenkins (agora você sabe por que nós o instalamos antes).

Retornamos ao nosso projeto e pressionamos “Configure” (Configurar) novamente; em seguida, localizamos a seção chamada “Build Environment” (Ambiente de construção). Usar o display virtual é simples e basta marcar a caixa chamada “Start Xvfb before the build, and shut it down after” (Iniciar o Xvfb antes da construção e encerrá-lo depois), como vemos na Figura 24.9.



*Figura 24.9 – Às vezes a configuração é simples.*

O resultado da construção é muito melhor agora:

```
[...]
Xvfb starting$ /usr/bin/Xvfb :2 -screen 0 1024x768x24 -fbdir
/var/lib/jenkins/2013-11-04_03-27-221510012427739470928xvfb
[...]
+ python manage.py test lists accounts

Ran 63 tests in 0.410s

OK
Creating test database for alias 'default'...
Destroying test database for alias 'default'...

+ pip install selenium
Requirement already satisfied (use --upgrade to upgrade): selenium in
/var/lib/jenkins/shiningpanda/jobs/ddc1aed1/virtualenvs/d41d8cd9/lib/python3.5/s
ite-packages
Cleaning up...
+ python manage.py test functional_tests
.....F.
=====
=====
FAIL: test_can_start_a_list_for_one_user
(functional_tests.test_simple_list_creation.NewVisitorTest)

Traceback (most recent call last):
 File "../superlists/functional_tests/test_simple_list_creation.py", line
43, in test_can_start_a_list_for_one_user
 self.wait_for_row_in_list_table('2: Use peacock feathers to make a fly')
 File "../superlists/functional_tests/base.py", line 51, in
```

```
wait_for_row_in_list_table
 raise e
 File ".../superlists/functional_tests/base.py", line 47, in
wait_for_row_in_list_table
 self.assertIn(row_text, [row.text for row in rows])
AssertionError: '2: Use peacock feathers to make a fly' not found in ['1: Buy
peacock feathers']
```

-----  
Ran 8 tests in 89.275s

```
FAILED (errors=1)
Creating test database for alias 'default'...
[{'secure': False, 'domain': 'localhost', 'name': 'sessionid', 'expiry':
1920011311, 'path': '/', 'value': 'a8d8bbde33nreq6gihw8a7r1cc8bf02k'}]
Destroying test database for alias 'default'...
Build step 'Virtualenv Builder' marked build as failure
Xvfb stopping
Finished: FAILURE
```

Estamos quase lá! Contudo, para depurar essa falha, precisaremos de capturas de tela.



Esse erro ocorreu por causa do desempenho de minha instância de Jenkins – talvez você veja um erro diferente, ou não veja nenhum. Qualquer que seja o caso, as ferramentas a seguir para obter capturas de tela e lidar com condições de concorrência (race conditions) serão úteis. Continue lendo!

## Capturando imagens de tela

Para depurar falhas inesperadas que ocorram em um computador remoto, seria bom ver uma imagem da tela no momento da falha e, quem sabe, também um dump do HTML da página. Podemos fazer isso usando alguma lógica personalizada no `tearDown` de nossa classe de FT. Temos que realizar uma pequena introspecção no funcionamento interno de `unittest` – um atributo privado chamado `_outcomeForDoCleanups` –, mas isso funcionará:

`functional_tests/base.py` (ch21I006)

```

import os
from datetime import datetime
[...]

SCREEN_DUMP_LOCATION = os.path.join(
 os.path.dirname(os.path.abspath(__file__)), 'screendumps'
)
[...]

def tearDown(self):
 if self._test_has_failed():
 if not os.path.exists(SCREEN_DUMP_LOCATION):
 os.makedirs(SCREEN_DUMP_LOCATION)
 for ix, handle in enumerate(self.browser.window_handles):
 self._windowid = ix
 self.browser.switch_to_window(handle)
 self.take_screenshot()
 self.dump_html()
 self.browser.quit()
 super().tearDown()

def _test_has_failed(self):
 # um pouco obscuro, mas não consegui pensar em uma maneira melhor!
 return any(error for (method, error) in self._outcome.errors)

```

Inicialmente criamos um diretório para nossas capturas de tela, se for necessário. Então iteramos por todas as abas e páginas abertas do navegador e usamos alguns métodos do Selenium, `get_screenshot_as_file` e `browser.page_source`, para nossas imagens e os dumps de HTML:

## functional\_tests/base.py (ch21|007)

```

def take_screenshot(self):
 filename = self._get_filename() + '.png'
 print('screenshotting to', filename)
 self.browser.get_screenshot_as_file(filename)

def dump_html(self):

```

```

filename = self._get_filename() + '.html'
print('dumping page HTML to', filename)
with open(filename, 'w') as f:
 f.write(self.browser.page_source)

```

Por fim, eis uma maneira de gerar um identificador único para o nome do arquivo, que inclui o nome do teste e sua classe, assim como um timestamp:

## functional\_tests/base.py (ch21I008)

```

def _get_filename(self):
 timestamp = datetime.now().isoformat().replace(':', '.')[0:19]
 return '{folder}/{classname}.{method}-window{windowid}-
{timestamp}'.format(
 folder=SCREEN_DUMP_LOCATION,
 classname=self.__class__.__name__,
 method=self._testMethodName,
 windowid=self._windowid,
 timestamp=timestamp
)

```

Você pode testar isso localmente antes, causando uma falha proposital em um dos testes, por exemplo, com um `self.fail()`; você verá algo como:

```

[...]
screenshotting to /.../superlists/functional_tests/screendumps/MyListsTest.test
_logged_in_users_lists_are_saved_as_my_lists-window0-2014-03-
09T11.19.12.png
dumping page HTML to
/.../superlists/functional_tests/screendumps/MyListsTest.t
est_logged_in_users_lists_are_saved_as_my_lists-window0-[...]

```

Remova o `self.fail()`, em seguida faça um commit e um push:

```

$ git diff # mudanças em base.py
$ echo "functional_tests/screendumps" >> .gitignore
$ git commit -am "add screenshot on failure to FT runner"
$ git push

```

Ao executar novamente a construção no Jenkins, veremos algo como:

```

screenshotting to /var/lib/jenkins/jobs/Superlists/.../functional_tests/

```

screendumps/LoginTest.test\_login\_with\_persona-window0-2014-01-22T17.45.12.png

dumping page HTML to /var/lib/jenkins/jobs/Superlists/.../functional\_tests/screendumps/LoginTest.test\_login\_with\_persona-window0-2014-01-22T17.45.12.html

Podemos ver isso no “workspace” (área de trabalho), que é a pasta usada pelo Jenkins para armazenar o nosso código-fonte e executar os testes, como mostra a Figura 24.10.

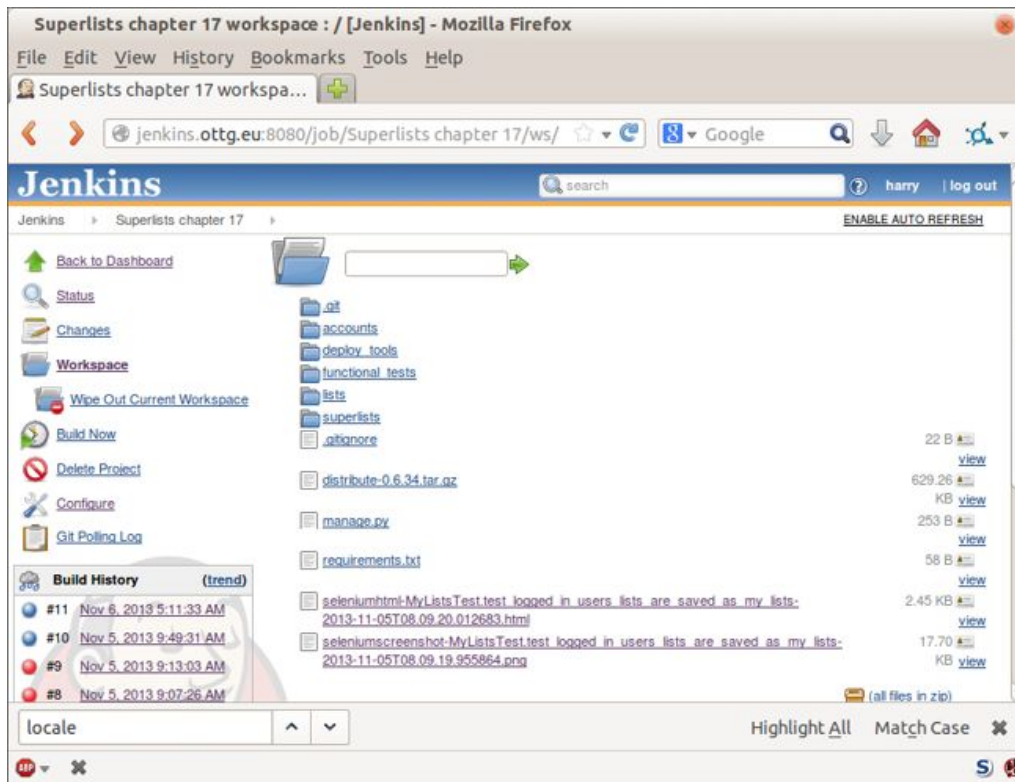
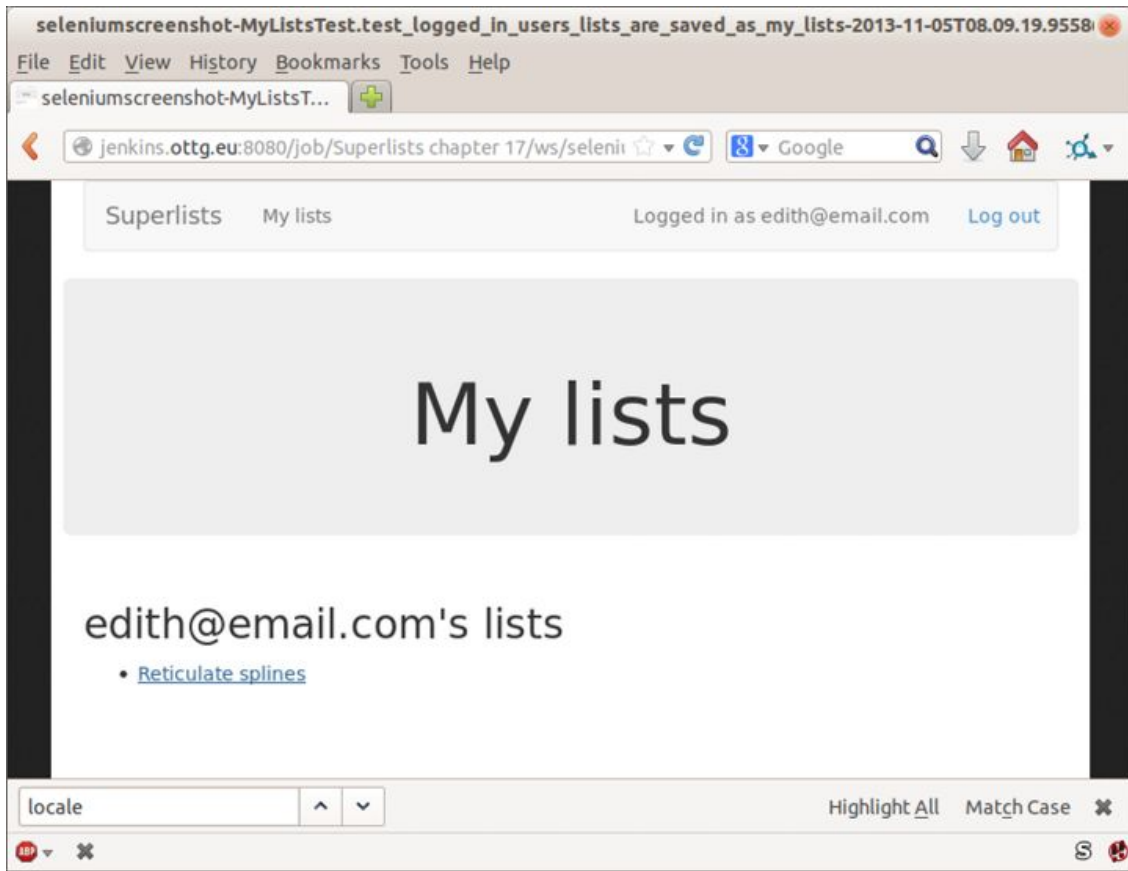


Figura 24.10 – Acessando o workspace do projeto.

Então vemos a captura de tela, conforme mostra a Figura 24.11.



*Figura 24.11 – Captura de tela com aspecto normal.*

## **Na dúvida, tente aumentar o timeout!**

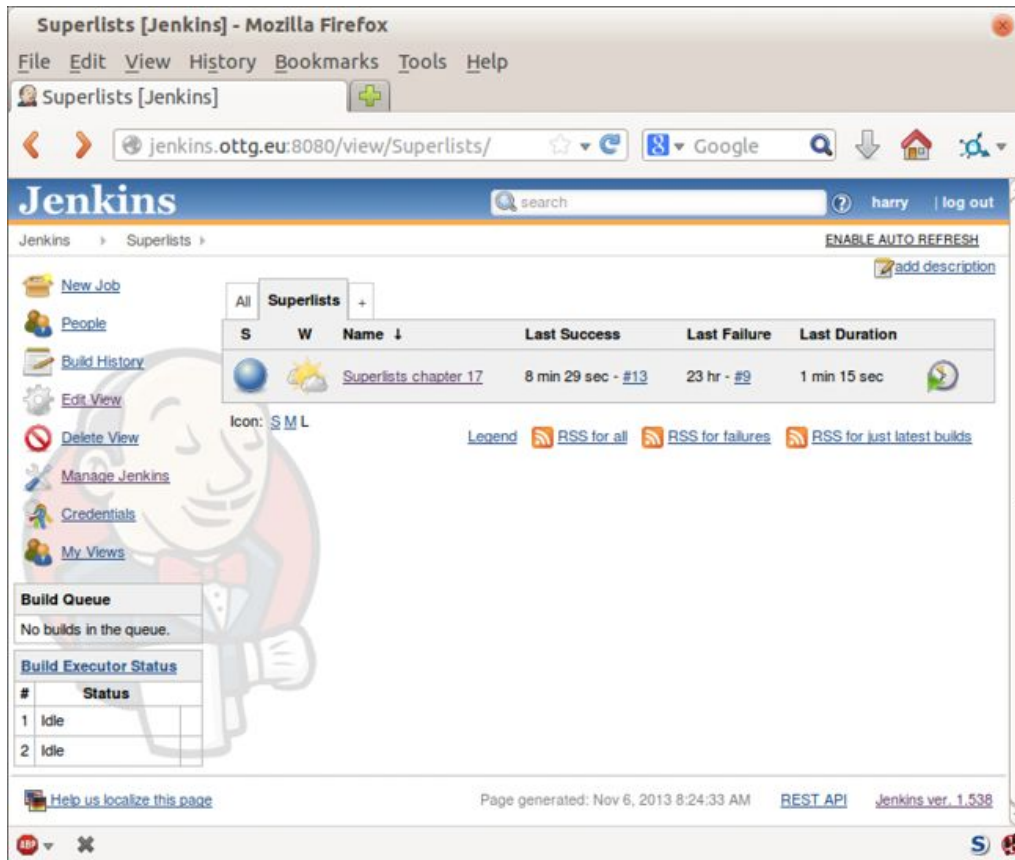
Hum. Não há nenhuma pista evidente nesse caso. Bem, na dúvida, aumente o timeout, como diz o velho ditado:

```
functional_tests/base.py
```

```
MAX_WAIT = 20
```

Então podemos executar a construção novamente no Jenkins usando “Build Now” (Construir agora), e confirmar se funciona, como mostra a Figura 24.12.





*Figura 24.12 – O resultado parece melhor.*

O Jenkins utiliza a cor azul para indicar construções com sucesso, em vez de usar verde, o que é um pouco decepcionante, mas veja o sol aparecendo por entre as nuvens: é animador! É um indicador de uma razão média mutável entre construções com sucesso e construções com falha. A situação está melhorando!

## **Executando nossos testes de JavaScript com QUnit no Jenkins com o PhantomJS**

Há um conjunto de testes que quase esquecemos – os testes de JavaScript. No momento, o nosso “executor de testes” é um navegador web. Para fazer o Jenkins executá-los, precisamos de um executor de testes de linha de comando. Eis uma oportunidade para usar o PhantomJS.

## **Instalando o node**

É hora de parar de fingir que não estamos no jogo do JavaScript. Estamos fazendo um desenvolvimento web. Isso significa que usamos JavaScript. Significa que acabaremos com o node.js em nossos computadores. É simplesmente assim que deve ser.

Siga as instruções na página de download do node.js (<http://nodejs.org/download/>). Há instaladores para Windows e Mac, além de repositórios para distribuições populares de Linux.<sup>2</sup>

Depois que tivermos o node, podemos instalar o phantom:

```
root@server $ npm install -g phantomjs # o -g significa "para todo o sistema".
```

Em seguida, temos que obter um executor de testes QUnit/PhantomJS. Há vários por aí (eu até mesmo escrevi um executor básico para testar as listagens de QUnit deste livro), mas o melhor provavelmente é aquele cujo link está na página de plugins da QUnit (<http://qunitjs.com/plugins/>). Quando escrevi este livro, seu repositório estava em <https://github.com/jonkemp/qunit-phantomjs-runner>. O único arquivo de que você precisará é *runner.js*.

Você deverá ter o seguinte no final:

```
$ tree lists/static/tests/
lists/static/tests/
├── qunit-2.0.1.css
├── qunit-2.0.1.js
├── runner.js
└── tests.html
```

```
0 directories, 4 files
```

Vamos testar:

```
$ phantomjs lists/static/tests/runner.js lists/static/tests/tests.html
Took 24ms to run 2 tests. 2 passed, 0 failed.
```

Somente para ter certeza, vamos provocar uma falha em algo propositalmente:

### lists/static/list.js (ch21I019)

```
$('input[name="text"]').on('keypress', function () {
 // $(':has-error').hide();
```

```
});
```

Com certeza:

```
$ phantomjs lists/static/tests/runner.js lists/static/tests/tests.html
```

```
Test failed: errors should be hidden on keypress
 Failed assertion: expected: false, but was: true
file:///.../superlists/lists/static/tests/tests.html:27:15
```

```
Took 27ms to run 2 tests. 1 passed, 1 failed.
```

Tudo certo! Vamos remover essa falha, fazer commit e push do executor e, então, adicioná-lo à nossa construção no Jenkins:

```
$ git checkout lists/static/list.js
$ git add lists/static/tests/runner.js
$ git commit -m "Add phantomjs test runner for javascript tests"
$ git push
```

## Acrescentando os passos de construção no Jenkins

Edite novamente a configuração do projeto e adicione um passo para cada conjunto de testes de JavaScript, conforme mostra a Figura 24.13.



*Figura 24.13 – Adicione um passo de construção para nossos testes de unidade de JavaScript.*

Você também precisará instalar o PhantomJS no servidor:

```
root@server:$ add-apt-repository -y ppa:chris-lea/node.js
root@server:$ apt-get update
root@server:$ apt-get install nodejs
root@server:$ npm install -g phantomjs
```

Pronto! Temos uma construção completa no CI com todos os nossos

testes!

```
Started by user harry
Building in workspace /var/lib/jenkins/jobs/Superlists/workspace
Fetching changes from the remote Git repository
Fetching upstream changes from https://github.com/hjwp/book-example.git
Checking out Revision 936a484038194b289312ff62f10d24e6a054fb29
(origin/chapter_1
Xvfb starting$ /usr/bin/Xvfb :1 -screen 0 1024x768x24 -fbdir /var/lib/jenkins/20
[workspace] $ /bin/sh -xe /tmp/shiningpanda7092102504259037999.sh
```

```
+ pip install -r requirements.txt
[...]
```

```
+ python manage.py test lists
```

```

Ran 43 tests in 0.229s
```

```
OK
Creating test database for alias 'default'...
Destroying test database for alias 'default'...
```

```
+ python manage.py test accounts
```

```

Ran 18 tests in 0.078s
```

```
OK
Creating test database for alias 'default'...
Destroying test database for alias 'default'...
```

```
[workspace] $ /bin/sh -xe /tmp/hudson2967478575201471277.sh
+ phantomjs lists/static/tests/runner.js lists/static/tests/tests.html
Took 32ms to run 2 tests. 2 passed, 0 failed.
+ phantomjs lists/static/tests/runner.js accounts/static/tests/tests.html
Took 47ms to run 11 tests. 11 passed, 0 failed.
```

```
[workspace] $ /bin/sh -xe /tmp/shiningpanda7526089957247195819.sh
+ pip install selenium
Requirement already satisfied (use --upgrade to upgrade): selenium in /var/lib/
```

```
Cleaning up...
```

```
[workspace] $ /bin/sh -xe /tmp/shiningpanda2420240268202055029.sh
+ python manage.py test functional_tests
```

```

Ran 8 tests in 76.804s
```

OK

É bom saber que, independentemente do nível de preguiça que eu tiver para executar a suíte de testes completa em meu próprio computador, o servidor de CI vai me proteger. É outro dos agentes do Testing Goat no ciberespaço, tomando conta de nós...

## **Outras tarefas que um servidor de CI pode fazer**

Mal toquei a superfície do que podemos fazer com o Jenkins e os servidores de CI. Por exemplo, você pode deixar muito mais inteligente o modo como o seu repositório será monitorado para saber se houve novos commits.

Talvez mais interessante ainda seja usar o seu servidor de CI para automatizar os testes de seu ambiente de staging, assim como os testes funcionais usuais. Se todos os FTs passarem, é possível adicionar um passo na construção para implantar o código no servidor de staging e então executar os FTs novamente aí – automatizando mais um passo do processo e garantindo que o seu servidor de staging permaneça automaticamente atualizado com o código mais recente.

Algumas pessoas até mesmo usam um servidor de CI como uma forma de implantar suas versões de produção!

## **Dicas sobre as melhores práticas associadas ao CI e ao Selenium**

### *Configure um CI o mais rápido possível para o seu projeto*

Assim que seus testes funcionais passarem a demorar mais que alguns segundos para executar, você se verá evitando executá-los. Passe essa tarefa para um servidor de CI a fim de garantir que todos os seus testes sejam executados em algum lugar.

## *Configure capturas de tela e dumps de HTML para falhas*

Depurar falhas de teste será mais fácil se você puder ver como era a aparência da página quando a falha ocorreu. Isso é particularmente útil para depuração de falhas no CI, mas é muito útil também para testes executados localmente.

### *Esteja preparado para aumentar seus timeouts*

Um servidor de CI talvez não seja tão rápido quanto o seu notebook, especialmente se estiver sujeito a cargas, executando vários testes ao mesmo tempo. Esteja preparado para ser mais generoso ainda com seus timeouts a fim de minimizar a chance de falhas aleatórias.



## *Investigue como associar o CI ao servidor de staging*

Testes que utilizem `LiveServerTestCase` são muito bons para ambientes de desenvolvimento, mas a verdadeira garantia vem da execução de seus testes em um servidor real. Investigue como o seu servidor de CI pode fazer a implantação em seu servidor de staging e executar aí os testes funcionais. Isso tem o efeito colateral de testar seus scripts automatizados para implantação.

- 
- 1 Dê uma olhada em `pyvirtualdisplay` (<https://pypi.python.org/pypi/PyVirtualDisplay>) como uma forma de controlar displays virtuais a partir de Python.
  - 2 Certifique-se de que obterá a versão mais recente. No Ubuntu, utilize o PPA em vez do pacote default.



## CAPÍTULO 25

# Aspecto social, padrão Page e exercício para o leitor

As piadas sobre como “tudo tem que ser social hoje em dia” não estão um pouco fora de moda? Tudo tem que ter testes A/B, big data, Tenha mais Cliques, Listas de 10 Coisas que Esse Professor Inspirador Disse, Isso Mudará a sua Mente Sobre blá-blá-blá Agora...enfim. Listas, sejam elas inspiradoras ou não, geralmente serão melhores se forem compartilhadas. Vamos permitir que nossos usuários contribuam com outros usuários com suas listas.

Nesse processo, melhoraremos os nossos FTs começando a implementar algo chamado padrão de objeto Page (Página).

Então, em vez de mostrar explicitamente o que deve ser feito, deixarei que você escreva seus testes de unidade e o código da aplicação por conta própria. Não se preocupe; você não estará totalmente sozinho! Apresentarei um esquema dos passos a serem executados, além de dar algumas pistas e dicas.

## Um FT com vários usuários, e a função `addCleanup`

Vamos começar – precisaremos de dois usuários neste FT:

`functional_tests/test_sharing.py (ch22I001)`

```
from selenium import webdriver
from .base import FunctionalTest
```

```
def quit_if_possible(browser):
 try: browser.quit()
 except: pass
```

```
class SharingTest(FunctionalTest):
```

```
 def test_can_share_a_list_with_another_user(self):
 # Edith é uma usuária logada
 self.create_pre_authenticated_session('edith@example.com')
 edith_browser = self.browser
 self.addCleanup(lambda: quit_if_possible(edith_browser))

 # Seu amigo Oniciferous também está no site de listas
 oni_browser = webdriver.Firefox()
 self.addCleanup(lambda: quit_if_possible(oni_browser))
 self.browser = oni_browser
 self.create_pre_authenticated_session('oniciferous@example.com')

 # Edith acessa a página inicial e começa uma lista
 self.browser = edith_browser
 self.browser.get(self.live_server_url)
 self.add_list_item('Get help')

 # Ela percebe que há uma opção "Share this list" (Compartilhar essa lista)
 share_box = self.browser.find_element_by_css_selector(
 'input[name="sharee"]'
)
 self.assertEqual(
 share_box.get_attribute('placeholder'),
 'your-friend@example.com'
)
```

O recurso interessante a ser observado nesta seção é a função `addCleanup`, cuja implementação você poderá encontrar online (<https://docs.python.org/3/library/unittest.html#unittest.TestCase.addCleanup>). Ela pode ser usada como alternativa para a função `tearDown` como uma forma de limpar os recursos usados durante o teste. É mais útil quando o recurso é alocado somente no meio da execução de um teste, de modo que você não precisará gastar

tempo em `tearDown` descobrindo o que precisa e o que não precisa de limpeza.

`addCleanup` é executado depois de `tearDown`, e é por isso que precisamos daquela construção `try/except` em `quit_if_possible`; o que quer que seja atribuído também a `self.browser` – `edith_browser` ou `oni_browser` – no ponto em que o teste termina, já terá sido encerrado pela função `tearDown`.

Também precisaremos passar `create_pre_authenticated_session` de `test_my_lists.py` para `base.py`.

Tudo bem, vamos ver se tudo isso funciona:

```
$ python manage.py test functional_tests.test_sharing
[...]
Traceback (most recent call last):
 File "../superlists/functional_tests/test_sharing.py", line 31, in
 test_can_share_a_list_with_another_user
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate
element: input[name="sharee"]
```

Ótimo! Parece que conseguimos criar as duas sessões de usuário e chegamos a uma falha esperada – não há nenhum dado de entrada com um endereço de email de uma pessoa com quem a lista será compartilhada na página.

Vamos fazer um commit nesse ponto, pois temos pelo menos um placeholder para o nosso FT, fizemos uma modificação conveniente na função `create_pre_authenticated_session` e estamos prestes a embarcar em uma pequena refatoração de um FT:

```
$ git add functional_tests
$ git commit -m "New FT for sharing, move session creation stuff to base"
```

## Padrão Page

Antes de avançar mais, gostaria de apresentar um método alternativo para reduzir a duplicação em seus FTs, chamado “objetos Page” (<http://bit.ly/2uWBvsM>).

Já construímos vários métodos auxiliares para os nossos FTs, incluindo `add_list_item`, que foi usado nesse caso, mas, se continuarmos simplesmente adicionando outros, os testes ficarão muito entulhados. Já trabalhei com uma classe-base de FT que tinha mais de mil e quinhentas linhas, e era muito difícil lidar com ela.

Os objetos `Page` são uma alternativa que nos incentiva a armazenar todas as informações e métodos auxiliares sobre os diferentes tipos de páginas de nosso site em um só local. Vamos ver como seria a aparência disso em nosso site, começando com uma classe para representar qualquer página de listas:

## functional\_tests/list\_page.py

```
from selenium.webdriver.common.keys import Keys
from .base import wait
```

```
class ListPage(object):
```

```
 def __init__(self, test):
 self.test = test ❶
```

```
 def get_table_rows(self): ❸
 return self.test.browser.find_elements_by_css_selector('#id_list_table tr')
```

```
 @wait
```

```
 def wait_for_row_in_list_table(self, item_text, item_number): ❷
 expected_row_text = f'{item_number}: {item_text}'
 rows = self.get_table_rows()
 self.test.assertIn(expected_row_text, [row.text for row in rows])
```

```
 def get_item_input_box(self): ❷
 return self.test.browser.find_element_by_id('id_text')
```

```
def add_list_item(self, item_text): ❶
 new_item_no = len(self.get_table_rows()) + 1
 self.get_item_input_box().send_keys(item_text)
 self.get_item_input_box().send_keys(Keys.ENTER)
 self.wait_for_row_in_list_table(item_text, new_item_no)
 return self ❷
```

- ❶ É inicializado com um objeto que representa o teste atual. Isso nos possibilita fazer asserções, acessar a instância do navegador por meio de `self.test.browser` e utilizar a função `self.test.wait_for`.
- ❷ Copiei alguns dos métodos auxiliares existentes em `base.py`, mas fiz pequenos ajustes...
- ❸ Por exemplo, eles fazem uso desse novo método.
- ❹ Devolver `self` é apenas uma conveniência. Permite fazer encadeamento de métodos ([https://en.wikipedia.org/wiki/Method\\_chaining](https://en.wikipedia.org/wiki/Method_chaining)), que veremos em ação imediatamente.

Vamos ver como usar isso em nosso teste:

## functional\_tests/test\_sharing.py (ch22I004)

```
from .list_page import ListPage
[...]

Edith acessa a página inicial e começa uma lista
self.browser = edith_browser
list_page = ListPage(self).add_list_item('Get help')
```

Vamos continuar reescrevendo o nosso teste usando o objeto `Page` sempre que quisermos acessar elementos da página de listas:

## functional\_tests/test\_sharing.py (ch22I008)

```
Ela percebe que há uma opção "Share this list" (Compartilhar essa lista)
share_box = list_page.get_share_box()
self.assertEqual(
 share_box.get_attribute('placeholder'),
 'your-friend@example.com'
)
```

```
Ela compartilha sua lista.
A página é atualizada para informar que a lista foi compartilhada
com Oniciferous:
list_page.share_list_with('oniciferous@example.com')
```

Adicionamos as três funções a seguir em nossa ListPage:

## functional\_tests/list\_page.py (ch22I009)

```
def get_share_box(self):
 return self.test.browser.find_element_by_css_selector(
 'input[name="sharee"]'
)

def get_shared_with_list(self):
 return self.test.browser.find_elements_by_css_selector(
 '.list-sharee'
)
def share_list_with(self, email):
 self.get_share_box().send_keys(email)
 self.get_share_box().send_keys(Keys.ENTER)
 self.test.wait_for(lambda: self.test.assertIn(
 email,
 [item.text for item in self.get_shared_with_list()]
))
```

A ideia por trás do padrão Page é que ele deve capturar todas as informações sobre uma página em particular de seu site, de modo que se, mais tarde, você quiser fazer alterações nessa página – mesmo que sejam apenas ajustes simples em seu layout de HTML, por exemplo – terá um único local para ajustar seus testes funcionais, em vez de precisar vasculhar dezenas de FTs.

O próximo passo seria visar à refatoração do FT com nossos outros testes. Mostrarei isso neste capítulo, mas é algo que você poderia fazer para exercitar, a fim de ter uma noção de como são as negociações de custo-benefício entre o princípio DRY e a legibilidade dos testes...

## **Estendendo o FT para um segundo usuário, e a**



## página “My Lists”

Vamos fazer a especificação um pouco mais detalhada de como queremos que seja a nossa história de usuário sobre compartilhamento. Edith viu em sua página que a lista agora está “compartilhada” com Oniciferous, e então podemos fazer Oni realizar login e ver a lista em sua página “My Lists”, talvez em uma seção chamada “lists shared with me” (listas compartilhadas comigo):

### functional\_tests/test\_sharing.py (ch22I010)

```
from .my_lists_page import MyListsPage
[...]
```

```
list_page.share_list_with('oniciferous@example.com')
```

```
Oniciferous agora acessa a página de listas com o seu navegador
self.browser = oni_browser
MyListsPage(self).go_to_my_lists_page()
```

```
Ele vê aí a lista de Edith!
self.browser.find_element_by_link_text('Get help').click()
```

Isso implica outra função em nossa classe MyListsPage:

### functional\_tests/my\_lists\_page.py (ch22I011)

```
class MyListsPage(object):
```

```
 def __init__(self, test):
 self.test = test
```

```
 def go_to_my_lists_page(self):
 self.test.browser.get(self.test.live_server_url)
 self.test.browser.find_element_by_link_text('My lists').click()
 self.test.wait_for(lambda: self.test.assertEqual(
 self.test.browser.find_element_by_tag_name('h1').text,
 'My Lists'
))
 return self
```

Mais uma vez, essa é uma função que seria conveniente levar para *test\_my\_lists.py*, junto, talvez, com um objeto *MyListsPage*.

Nesse ínterim, *Oniciferous* também pode adicionar itens na lista:

## functional\_tests/test\_sharing.py (ch22I012)

```
Na página de lista, Oniciferous pode ver que a lista é de Edith
self.wait_for(lambda: self.assertEqual(
 list_page.get_list_owner(),
 'edith@example.com'
))

Ele adiciona um item na lista
list_page.add_list_item('Hi Edith!')

Quando Edith atualiza a página, ela vê o acréscimo feito por Oniciferous
self.browser = edith_browser
self.browser.refresh()
list_page.wait_for_row_in_list_table('Hi Edith!', 2)
```

Há outro acréscimo em nosso objeto *ListPage*:

## functional\_tests/list\_page.py (ch22I013)

```
class ListPage(object):
 [...]

 def get_list_owner(self):
 return self.test.browser.find_element_by_id('id_list_owner').text
```

Já está mais do que na hora de executar o FT e verificar se tudo isso funciona!

```
$ python manage.py test functional_tests.test_sharing
```

```
share_box = list_page.get_share_box()
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate
element: input[name="sharee"]
```

Essa é a falha esperada; não temos um dado de entrada com os endereços de email das pessoas com quem o compartilhamento

será feito. Vamos fazer um commit:

```
$ git add functional_tests
```

```
$ git commit -m "Create Page objects for list pages, use in sharing FT"
```

## Um exercício para o leitor

Provavelmente eu não estava entendendo o que estava fazendo até ter concluído o “Exercício para o leitor” no Capítulo 25.

– IAIN H. (LEITOR)

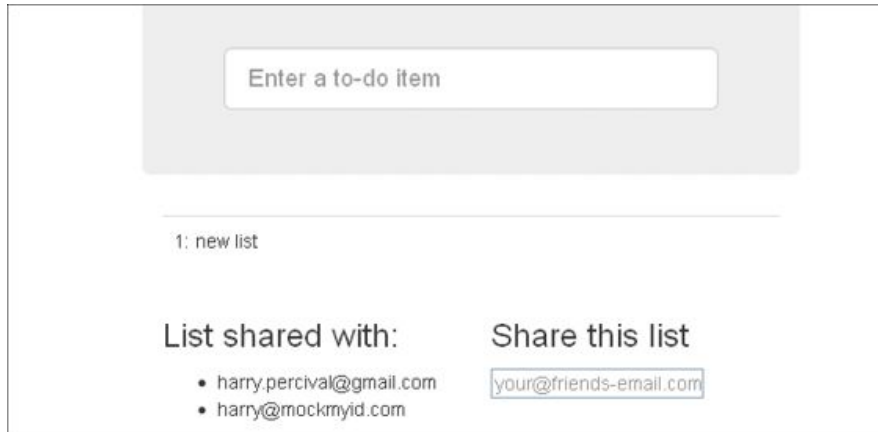
Não há nada como tirar as rodinhas da bicicleta e fazer algo funcionar por conta própria para consolidar o conhecimento, portanto espero que você tente fazer o que apresentamos a seguir.

Eis um esquema dos passos que você deve executar:

1. Precisaremos de uma nova seção em *list.html*, inicialmente com um formulário contendo uma caixa de entrada para um endereço de email. Isso deverá fazer o FT avançar mais um passo.
2. Em seguida, precisaremos de uma view para a qual o formulário fará a submissão. Comece definindo o URL no template, talvez com algo como *lists/<list\_id>/share*.
3. Então escreva o primeiro teste de unidade. Pode ser apenas o suficiente para ter um placeholder para a view. Queremos que a view responda a requisições POST, e ela deve responder com um redirecionamento para a página de lista, portanto o teste poderia ter um nome semelhante a `ShareListTest.test_post_redirects_to_lists_page`.
4. Vamos construir nossa view placeholder simplesmente com duas linhas para encontrar uma lista e fazer o redirecionamento.
5. Podemos então escrever um novo teste de unidade que crie um usuário e uma lista, faça um POST com seu endereço de email e verifique se o usuário é adicionado em `list_.shared_with.all()` (um uso de ORM semelhante a “My Lists”). Esse atributo `shared_with` ainda não existe; usaremos a abordagem *outside-in* (de fora para dentro).

6. Assim, antes de fazer esse teste passar, precisaremos descer até a camada de modelo. O próximo teste, em *test\_models.py*, pode verificar se uma lista tem um método `shared_with.add`, que pode ser chamado com o endereço de email de um usuário, e então verificar a queryset `shared_with.all()` das listas, que conterá esse usuário depois.
7. Então você precisará de um `ManyToManyField`. Provavelmente verá uma mensagem de erro sobre um conflito com `related_name`, para o qual encontrará uma solução se consultar a documentação do Django.
8. Será necessário fazer uma migração do banco de dados.
9. Isso deve fazer os testes de modelo passarem. Volte para corrigir o teste da view.
10. Talvez você descubra que o teste de redirecionamento da view falhe porque ele não está enviando uma requisição POST válida. Você pode optar por ignorar dados de entrada inválidos ou adaptar o teste para enviar um POST válido.
11. Volte então para o nível do template; na página “My Lists”, queremos um `<ul>` com um laço `for` nas listas compartilhadas com o usuário. Na página de listas, também queremos mostrar com quem a lista está sendo compartilhada, bem como fazer uma menção de quem é o proprietário da lista. Observe novamente o FT para ver as classes e os IDs corretos a serem usados. Você também poderia ter testes de unidade rápidos para cada um deles, se quiser.
12. Talvez você ache que iniciar o site com `runserver` vá ajudá-lo a limpar qualquer bug, assim como fazer ajustes finos no layout e na estética. Se usar uma sessão privada de navegador, você poderá fazer login com vários usuários.

No final, poderá acabar com algo que se assemelhe à Figura 25.1.



*Figura 25.1 – Compartilhando listas.*

## **Padrão Page e o verdadeiro exercício para o leitor**

### *Aplique o princípio DRY em seus testes funcionais*

Depois que sua suíte de FT começar a crescer, você perceberá que testes diferentes, inevitavelmente, usarão partes semelhantes da UI. Procure evitar ter constantes, por exemplo, os IDs de HTML ou classes de elementos específicos da UI, duplicadas entre os seus FTs.

## *Padrão Page*

Passar os métodos auxiliares para uma classe-base `FunctionalTest` pode deixá-la difícil de ser administrada. Considere o uso de objetos `Page` individuais a fim de armazenar toda a lógica para lidar com partes específicas de seu site.

### *Um exercício para o leitor*

Espero que você tenha de fato tentado executar esses passos! Procure seguir o método outside-in e, ocasionalmente, experimente executar manualmente, caso se veja sem saber o que fazer. É claro que o verdadeiro exercício para o leitor consiste em aplicar o TDD em seu próximo projeto. Espero que você aprecie!

No próximo capítulo, concluiremos com uma discussão sobre as “melhores práticas” para os testes.





## CAPÍTULO 26

# Testes rápidos, testes lentos e Lava Quente

O banco de dados é Lava Quente!<sup>1</sup>

– CASEY KINSEY ([HTTPS://WWW.YOUTUBE.COM/WATCH?V=BSMFVB8GUMU](https://www.youtube.com/watch?v=BSMFVB8GUMU))

Até o Capítulo 23, quase todos os testes de “unidade” do livro talvez devessem ter sido chamados de testes *integrados* porque contavam com o banco de dados ou usavam o Django Test Client, que aplica uma boa dose de magia nas camadas intermediárias entre as requisições, as respostas e as funções de view.

Há uma discussão em que se argumenta que um verdadeiro teste de unidade deve ser sempre isolado, pois seu propósito é testar uma única unidade de software. Se entrar em contato com o banco de dados, o teste não poderá ser um teste de unidade. O banco de dados é lava quente!

Alguns veteranos em TDD dizem que você deve se empenhar em escrever testes de unidade “puros” e isolados sempre que possível, em vez de escrever testes integrados. É um daqueles debates (ocasionalmente acalorados) em andamento na comunidade de testes.

Sendo somente um jovem pretencioso, conheço apenas partes das sutilezas da discussão. Neste capítulo, porém, gostaria de discutir por que as pessoas têm opiniões fortes sobre o assunto e tentarei dar uma ideia de quando você poderá se sair bem lidando com testes integrados (devo confessar que faço muito isso!) e quando

vale a pena se empenhar em ter testes de unidade mais “puros”.

## **Terminologia: diferentes tipos de testes**

## *Testes isolados (testes de unidade “puros”) versus testes integrados*

O principal propósito de um teste de unidade deve ser verificar se a lógica de sua aplicação está correta. Um teste *isolado* é aquele que testa exatamente uma porção de código, cujo sucesso ou a falha não dependa de nenhum outro código externo. É isso que eu chamo de teste de unidade “puro”: um teste para uma única função, por exemplo, escrita de modo que somente essa função possa fazê-lo falhar. Se a função depender de outro sistema, e se uma falha nesse sistema fizer o nosso teste falhar, teremos um teste *integrado*. Esse sistema poderia ser um sistema externo, como um banco de dados, mas poderia também ser outra função da qual não temos controle. Qualquer que seja o caso, se a falha no sistema fizer o nosso teste falhar, este não estará devidamente isolado; o teste não será um teste de unidade “puro”. Isso não é necessariamente ruim, mas pode significar que o teste está fazendo duas tarefas ao mesmo tempo.

## *Testes de integração*

Um teste de integração verifica se o código que você controla está corretamente integrado com algum sistema externo que você não controla.

*Testes de integração geralmente são também testes integrados.*

## *Testes de sistema*

Se um teste de integração verifica a integração com um sistema externo, um teste de sistema verifica a integração entre vários sistemas em sua aplicação – por exemplo, verificar se associamos o nosso banco de dados, os arquivos estáticos e a configuração do sistema em conjunto, de modo que todos eles funcionem.

## *Testes funcionais e testes de aceitação*

Um teste de aceitação tem como propósito testar se o nosso sistema funciona do ponto de vista do usuário (“o usuário aceitaria esse comportamento?”). É difícil escrever um teste de aceitação que não seja um teste de pilha completa (full-stack), fim a fim (end-to-end). Estamos usando nossos testes funcionais de modo que desempenhem a função tanto de testes de aceitação quanto de testes de sistema.

Se você me perdoar a terminologia filosófica pretenciosa, gostaria de estruturar a nossa discussão sobre essas questões na forma de uma dialética hegeliana:

- A Tese: a questão dos testes de unidade “puros” e rápidos.
- A Antítese: alguns dos riscos associados a uma abordagem (ingênua) de testes de unidade puros.
- A Síntese: uma discussão sobre as melhores práticas, como “Ports and Adapters” (Portas e Adaptadores) ou “Functional Core, Imperative Shell” (Núcleo Funcional, Shell Imperativo), e sobre o que é que queremos de nossos testes, afinal de contas.

## **Tese: testes de unidade são super-rápidos e bons, além de tudo**

Uma das afirmações que você ouvirá com frequência sobre os testes de unidade é que eles são muito rápidos. Não acho que essa seja a principal vantagem dos testes de unidade, mas vale a pena explorar o tema sobre rapidez.

## **Testes mais rápidos significam desenvolvimento mais rápido**

Sem considerar outros fatores, quanto mais rápidos seus testes de unidade executarem, melhor será. Em menor grau, quanto mais rápidos *todos* os seus testes executarem, melhor.

Descrevi o ciclo de testes/código do TDD neste livro. Você começou

a ter uma noção do fluxo de trabalho do TDD, isto é, o modo como você alterna entre escrever pequenas porções de código e executar seus testes. Você acabará executando seus testes de unidade várias vezes por minuto, e seus testes funcionais várias vezes por dia.

Assim, em um nível bem básico, quanto mais eles demorarem, mais tempo você gastará esperando pelos seus testes, e isso deixará seu desenvolvimento mais lento. No entanto, além desse, há outros fatores envolvidos.

## **O sagrado estado de fluxo**

Pensando em sociologia por um momento, nós, programadores, temos, de certo modo, a nossa própria cultura e a nossa própria religião tribal. Há muitas congregações aí, como a cultura do TDD à qual você está sendo iniciado agora. Temos os seguidores de vi e os hereges do emacs. Porém um ponto sobre o qual todos nós concordamos – uma prática espiritual em particular, a nossa meditação transcendental própria – é o sagrado estado de fluxo. Essa sensação de foco puro, de concentração, em que nem vemos a hora passar, o código flui naturalmente de nossos dedos, os problemas são complicados apenas o suficiente para serem interessantes, mas não tão difíceis a ponto de nos derrotar...

Não há absolutamente nenhuma esperança de conseguir esse fluxo se você gastar seu tempo esperando uma suíte de testes lenta ser executada. Qualquer demora que não seja de alguns segundos, e você terá sua atenção desviada, mudará de contexto e o estado de fluxo se perderá. E o estado de fluxo é um sonho frágil. Se for perdido, demorará no mínimo 15 minutos para que possamos vivenciá-lo novamente.

## **Testes lentos não são executados com tanta frequência, o que resulta em código ruim**

Se a sua suíte de testes for lenta e arruinar a sua concentração, o



perigo será você começar a evitar a execução de seus testes, o que poderá resultar no aparecimento de bugs. Ou poderá nos levar a uma timidez na refatoração de código, pois saberemos que qualquer refatoração significará ter de esperar anos para todos os testes executarem. Qualquer que seja o caso, um código ruim poderá resultar disso.

## **Não há problemas agora, mas os testes integrados se tornam mais lentos com o passar do tempo**

Você pode estar pensando, tudo bem, mas nossa suíte de testes contém muitos testes integrados – mais de 50 deles, e demora apenas 0,2 segundo para executar.

Lembre-se, porém, de que temos uma aplicação bem simples. Depois que ela começar a se tornar mais complexa, à medida que o seu banco de dados crescer e adquirir mais e mais tabelas e colunas, os testes integrados se tornarão cada vez mais lentos. Fazer o Django reiniciar o banco de dados entre cada teste passará a demorar cada vez mais.

## **Não sou eu quem está falando**

Gary Bernhardt, um homem com muito mais experiência em testes do que eu, apresenta essas questões de forma eloquente em uma palestra chamada Fast Test, Slow Test (Teste rápido, teste lento, <https://www.youtube.com/watch?v=RAXiiRPHS9k>). Incentivo você a assistir a ela.

## **E os testes de unidade levam a um bom design**

Contudo, mais importante do que tudo isso é lembrar-se da lição do Capítulo 23. Passar pelo processo de escrever testes integrados bons e isolados pode nos ajudar a ter melhores designs para o nosso código, forçando-nos a identificar dependências e

incentivando-nos em direção a uma arquitetura desacoplada, de um modo que os testes integrados não o fazem.

## **Os problemas com os testes de unidade “puros”**

Tudo isso vem com um enorme “porém”. Escrever testes de unidade isolados tem os próprios perigos, particularmente se, assim como você ou eu, ainda não somos usuários experientes em TDD.

## **Testes isolados podem ser mais difíceis de ler e de escrever**

Relembre o primeiro teste de unidade isolado que escrevemos. Não era feio? Vamos admitir que a situação melhorou quando refatoramos o código em formulários, mas suponha que não tivéssemos continuado. Teríamos ficado com um teste bastante ilegível em nossa base de código. Até mesmo a versão dos testes com que acabamos ficando no final contêm algumas partes que dão um pouco de nó na cabeça.

## **Testes isolados não testam automaticamente a integração**

Como vimos um pouco depois, os testes isolados, por sua natureza, somente verificam a unidade em teste em isolamento. Eles não testam a integração entre suas unidades.

Esse problema é bem conhecido, e há formas de atenuá-lo. Entretanto, conforme vimos, essas atenuações envolvem um pouco de trabalho árduo da parte do programador – você deve se lembrar de manter o controle das interfaces entre suas unidades a fim de identificar o contrato implícito que cada componente precisa honrar, e escrever testes para esses contratos, assim como para a funcionalidade interna de sua unidade.

## **Testes de unidade raramente capturam bugs**

## inesperados

Os testes de unidade ajudarão você a identificar erros de deslocamento de um (off-by-one) e uma confusão na lógica, que são os tipos de bugs que sabemos que introduzimos o tempo todo, portanto, de certo modo, são esperados. No entanto, os testes não nos advertem sobre os bugs mais inesperados. Eles não o lembram caso você tenha se esquecido de criar uma migração de banco de dados. Não lhe dizem quando a camada intermediária está fazendo algum escaping inteligente de entidade HTML que esteja interferindo com o modo como seus dados são renderizados... Algo como os “desconhecidos desconhecidos” de Donald Rumsfeld<sup>2</sup>.

## Testes com simulação podem se tornar extremamente vinculados à implementação

Por fim, testes com simulação podem se tornar altamente acoplados à implementação. Se optar por usar `List.objects.create()` para construir seus objetos, mas seus mocks estiverem esperando que você use `List()` e `.save()`, você terá testes que falham, apesar de o efeito propriamente dito do código ser o mesmo. Se você não tiver cuidado, isso pode começar a atuar contra uma das supostas vantagens de ter testes, que é incentivar a refatoração. Você poderá se ver tendo que alterar dezenas de testes com simulação e testes de contratos quando quiser alterar uma API interna.

Observe que a situação pode ser mais problemática quando você estiver lidando com uma API que não esteja em seu controle. Talvez você se lembre da ginástica que precisamos fazer para testar o nosso formulário, simulando duas classes de modelo de Django e usando `side_effect` para verificar o estado do mundo. Se você estiver escrevendo um código totalmente em seu próprio controle, é provável que vá fazer o design de suas APIs internas de modo que elas sejam mais claras e exijam menos ginástica para serem testadas.

## **Entretanto todos os problemas podem ser superados**

No entanto, as pessoas que defendem o isolamento replicarão dizendo que tudo isso pode ser atenuado; basta aperfeiçoar o modo de escrever os testes isolados. Você se lembra do sagrado estado de fluxo? O sagrado estado de fluxo!

Então, temos que escolher um ou outro lado?

## **Síntese: afinal de contas, o que queremos de nossos testes?**

Vamos dar um passo para trás e pensar nos benefícios que queremos ter com nossos testes. Antes de tudo, por que estamos escrevendo esses testes?

### **Aplicação correta**

Queremos que nossa aplicação esteja livre de bugs – tanto os erros de lógica de baixo nível, como erros de deslocamento de um, quanto os bugs de alto nível, como, em última instância, o software não estar oferecendo o que os usuários querem. Queremos descobrir se, em algum momento, introduzimos regressões, que causem falhas em algo que costumava funcionar, e queremos descobrir isso antes que nossos usuários vejam algo com falha. Esperamos que nossos testes nos digam se nossa aplicação está correta.

### **Código limpo, possível de manter**

Queremos que o nosso código obedeça a regras como YAGNI e DRY. Queremos um código que expresse claramente suas intenções, separado em componentes sensatos, com responsabilidades bem definidas e facilmente compreensíveis. Esperamos que nossos testes nos deem confiança para refatorar nossa aplicação constantemente, de modo que jamais fiquemos

com medo de tentar melhorar o seu design, e também apreciaríamos se eles nos ajudassem ativamente a fazer o design correto.

## Fluxo de trabalho produtivo

Por fim, queremos que nossos testes nos ajudem a possibilitar um fluxo de trabalho rápido e produtivo. Queremos que nos ajudem a eliminar parte do estresse de nosso desenvolvimento, e queremos que nos protejam contra erros estúpidos. Queremos que contribuam para nos manter no estado de “fluxo”, não só porque gostamos disso, mas também porque é altamente produtivo. Queremos que nossos testes nos deem feedback sobre o nosso trabalho o mais rápido possível, de modo que possamos testar novas ideias e fazer com que elas evoluam rapidamente. Além disso, não queremos pensar em nossos testes mais como um incômodo do que como uma ajuda quando se trata da evolução de nossa base de código.

## Avalie seus testes em relação aos benefícios que você quer que eles proporcionem

Não acho que haja qualquer regra universal sobre a quantidade de testes que você deva escrever e qual deve ser o equilíbrio correto entre testes funcionais, integrados e isolados. As circunstâncias variam de acordo com cada projeto. No entanto, ao pensar em todos os seus testes e se perguntar se eles estão proporcionando as vantagens desejadas, você poderá tomar algumas decisões.

*Tabela 26.1 – Como os diferentes tipos de teste nos ajudam a alcançar os nossos objetivos?*

Objetivo	Algumas considerações
<i>Aplicação correta</i>	<ul style="list-style-type: none"><li>• Tenho testes funcionais suficientes para me sentir seguro de que minha aplicação <i>realmente</i> funciona do ponto de vista do usuário?</li><li>• Estou testando todos os casos extremos de forma abrangente? Isso parece uma tarefa para testes isolados, de baixo nível.</li><li>• Tenho testes que verificam se todos os meus componentes se encaixam apropriadamente? Alguns testes integrados poderiam</li></ul>

	fazer isso, ou os testes funcionais são suficientes?
<i>Código limpo, possível de manter</i>	<ul style="list-style-type: none"> <li>• Meus testes estão me dando confiança para refatorar o meu código, sem medo e com frequência?</li> <li>• Meus testes estão me ajudando a obter um bom design? Se eu tiver vários testes integrados e poucos testes isolados, há alguma parte de minha aplicação em que investir mais esforços para escrever testes mais isolados resultaria em um feedback melhor sobre o meu design?</li> </ul>
<i>Fluxo de trabalho produtivo</i>	<ul style="list-style-type: none"> <li>• Meus ciclos de feedback estão tão rápidos quanto eu gostaria? Quando sou avisado a respeito de bugs, e há alguma forma prática de fazer isso acontecer mais cedo?</li> <li>• Se eu tenho muitos testes funcionais de alto nível, que demorem bastante tempo para executar, e tenho que esperar uma noite para obter um feedback sobre regressões acidentais, há alguma maneira com a qual eu poderia escrever testes mais rápidos, quem sabe, testes integrados, que me dariam feedbacks mais rapidamente?</li> <li>• Posso executar um subconjunto da suíte de testes completa quando for necessário?</li> <li>• Estou gastando tempo demais esperando os testes executarem e, desse modo, menos tempo em um estado de fluxo produtivo?</li> </ul>

## Soluções arquitetônicas

Há também algumas soluções arquitetônicas que podem ajudar a obter o máximo de proveito de sua suíte de testes e, particularmente, que ajudem a evitar algumas das desvantagens dos testes isolados.

Elas envolvem principalmente a tentativa de identificar as fronteiras de seu sistema – os pontos em que o seu código interage com sistemas externos, como o banco de dados ou o sistema de arquivos, ou a internet ou a UI – e tentar mantê-las separadas da lógica de negócios nuclear de sua aplicação.

## Portas e adaptadores/arquitetura hexagonal/limpa

Os testes integrados são mais úteis nas *fronteiras* de um sistema – nos pontos em que o nosso código é integrado com sistemas externos, como um banco de dados, o sistema de arquivos ou

componentes de UI.

De modo semelhante, é nas fronteiras que as desvantagens do isolamento de testes e dos mocks são mais evidentes, pois é nesses locais que há mais chances de você ficar irritado caso seus testes estejam altamente acoplados a uma implementação, ou que você terá mais necessidade de ter certeza de que tudo está devidamente integrado.

De modo recíproco, o código no *núcleo* de nossa aplicação – o código que está exclusivamente preocupado com o domínio de nosso negócio e com as regras de negócio, o código que está totalmente em nosso controle – tem menos necessidade de testes integrados, pois nós o controlamos e o compreendemos por completo.

Assim, uma forma de obter o que queremos é tentar minimizar o volume de nosso código que precise lidar com as fronteiras. Então testamos a lógica de negócios nuclear com testes isolados e testamos nossos pontos de integração com testes integrados.

Steve Freeman e Nat Pryce, em seu livro *Growing Object-Oriented Software, Guided by Tests*, chamam essa abordagem de “Ports and Adapters” (Portas e Adaptadores – veja a Figura 26.1).

Na verdade, nós começamos a nos mover em direção à arquitetura de portas e adaptadores no Capítulo 23, quando descobrimos que escrever testes de unidade isolados estava nos incentivando a remover o código de ORM da aplicação principal e ocultá-lo em funções auxiliares da camada de modelo.

Esse padrão às vezes é conhecido também como “arquitetura limpa” (clean architecture) ou “arquitetura hexagonal” (hexagonal architecture). Consulte a seção de “Leituras complementares” para ver mais informações.

## **Functional Core, Imperative Shell**

Gary Bernhardt leva essa questão além, recomendando uma

arquitetura que ele chama de “Functional Core, Imperative Shell” (Núcleo funcional, Shell imperativo), segundo a qual o “shell” da aplicação – o local em que a interação com as fronteiras ocorre – segue o paradigma da programação imperativa, e pode ser testada por testes integrados, testes de aceitação, ou até mesmo (pasmem!) por nada, se for mantida em um nível mínimo. No entanto, o núcleo da aplicação é escrito de acordo com o paradigma da programação funcional (completo, com o corolário do “sem efeitos colaterais”), o que permite ter testes de unidade totalmente isolados e “puros”, *totalmente sem mocks*.

Dê uma olhada na apresentação de Gary intitulada “Boundaries” (Fronteiras, <https://www.youtube.com/watch?v=eOYal8elnZk>) para saber mais sobre essa abordagem.

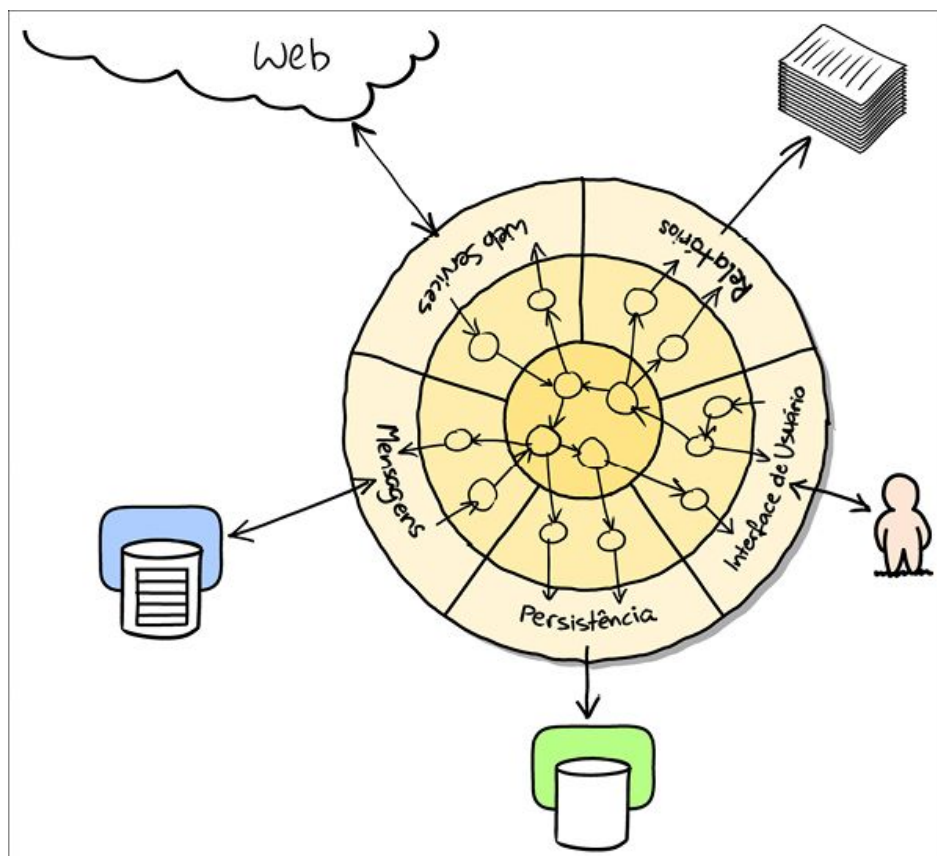


Figura 26.1 – Portas e Adaptadores (diagrama de Nat Pryce).

## Conclusão



Procurei apresentar uma visão geral de algumas das considerações mais avançadas que surgem no processo de TDD. Dominar esses assuntos é algo que vem com anos de prática, e eu ainda não cheguei lá, de forma alguma. Portanto, sinceramente, incentivo você a considerar tudo o que eu disse com certa cautela; saia por aí, experimente diversas abordagens, ouça o que outras pessoas têm a dizer também e descubra o que funciona em seu caso.

Eis algumas referências para leituras complementares.

## **Leituras complementares**

*Fast Test, Slow Test (Teste rápido, teste lento) e Boundaries (Fronteiras)*

Palestras de Gary Bernhardt na Pycon 2012 (<https://www.youtube.com/watch?v=RAXiiRPHS9k>) e 2013 (<https://www.youtube.com/watch?v=eOYal8elnZk>). Também vale a pena dar uma olhada em seus vídeos de capturas de tela (<http://www.destroyallsoftware.com>).

*Ports and Adapters (Portas e adaptadores)*

Steve Freeman e Nat Pryce escreveram sobre esse assunto em seu livro. Você também pode acompanhar uma boa discussão na palestra em <http://vimeo.com/83960706>. Veja também a descrição de Uncle Bob sobre a arquitetura limpa (<http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html>), e a cunhagem do termo “arquitetura hexagonal” (hexagonal architecture) por Alistair Cockburn (<http://alistair.cockburn.us/Hexagonal+architecture>).

*Lava quente*

A frase memorável de Casey Kinsey (<https://www.youtube.com/watch?v=bsmFVb8guMU>) incentivando você a evitar entrar em contato com o banco de dados sempre que puder.

### *Invertendo a pirâmide*

A ideia de que os projetos acabam com uma razão demasiadamente elevada entre testes de alto nível lentos e testes de unidade, e uma metáfora visual para o esforço de inverter essa razão (<http://watirmelon.com/tag/testing-pyramid/>).

### *Testes integrados são um golpe*

J.B. Rainsberger tem um famoso discurso acalorado (<http://blog.thecodewhisperer.com/2010/10/16/integrated-tests-are-a-scam/>) sobre o modo como os testes integrados arruinaram sua vida. Em seguida, dê uma olhada em algumas postagem de resposta, em particular nesta defesa dos testes de aceitação (<http://www.jbrains.ca/permalink/using-integration-tests-mindfully-acase-study>) (que eu chamo de testes funcionais), e nesta análise de como testes lentos acabam com a produtividade (<http://www.jbrains.ca/permalink/part-2-some-hidden-costs-ofintegration-tests>).

### *A wiki de testes Test-Double*

O recurso online de Justin Searls é uma ótima fonte para definições e discussões sobre os prós e contras dos testes, e chega às próprias conclusões sobre a forma correta de trabalhar: veja a wiki de testes (<https://github.com/testdouble/contributing-tests/wiki/Test-Driven-Development>).

### *Uma visão pragmática*

Martin Fowler (autor do livro *Refatoração*) apresenta uma abordagem pragmática, razoavelmente equilibrada (<http://martinfowler.com/bliki/UnitTest.html>).

## **Sobre ter o equilíbrio correto entre os diferentes tipos de testes**

### *Comece sendo pragmático*

Gastar muito tempo em agonia sobre os tipos de testes a serem escritos é uma ótima maneira de prevaricar. É melhor começar escrevendo qualquer que seja o tipo de teste que ocorrer inicialmente a você e alterá-lo mais tarde, se for necessário. Aprenda fazendo.

### *Mantenha o foco no que você quer com seus testes*

Seus objetivos são: *aplicação correta*, um *bom design* e *ciclos rápidos de feedback*. Tipos distintos de teste ajudarão você a alcançar cada um desses objetivos em diferente medida. A Tabela 26.1 tem algumas perguntas boas para você fazer a si mesmo.

## *Arquitetura é importante*

Em certa medida, sua arquitetura determina os tipos de testes necessários. Quanto mais você separar sua lógica de negócios de suas dependências externas, e quanto mais modular for o seu código, mais próximo chegará de um bom equilíbrio entre testes de unidade, testes de integração e testes fim a fim.

---

1 N.T.: No original: “The database is Hot Lava!”.

2 N.T.: Donald Rumsfeld é ex-Secretário de Defesa dos Estados Unidos e a expressão “desconhecidos desconhecidos” encontra-se em uma famosa citação sua. Em tradução livre: “Relatos que informam que algo não aconteceu são sempre interessantes para mim porque, como sabemos, há conhecidos conhecidos; há coisas que sabemos que sabemos. Também sabemos que há conhecidos desconhecidos; isso equivale a dizer que sabemos que há algumas coisas que não sabemos. Porém, há também os desconhecidos desconhecidos – coisas que não sabemos que não sabemos. E, se observarmos a história de nosso país e de outros países livres, é a última categoria que tende a ser a difícil.” (Baseado em [https://en.wikipedia.org/wiki/There\\_are\\_known\\_knowns](https://en.wikipedia.org/wiki/There_are_known_knowns)).



# EPÍLOGO

## Obedeça ao Testing Goat!

De volta ao Testing Goat (bode dos testes).

*Humpf*, ouço você dizendo: *Harry*, o Testing Goat deixou de ser engraçado há cerca de 17 capítulos. Fique comigo; eu o usarei para fazer uma afirmação séria.

### Testar é difícil

Acho que o motivo para a frase “Obedeça ao Testing Goat” (siga o bode dos testes) ter me conquistado quando a vi pela primeira vez foi o fato de ela realmente falar que testar é difícil – não é difícil por si só, mas é difícil *ater-se* a ele e é difícil continuar testando.

Sempre parece mais fácil tomar atalhos e pular alguns testes. Sem dúvida, é bem mais difícil do ponto de vista psicológico, pois a recompensa está muito desconectada do ponto em que você investe seus esforços. Um teste para o qual você invista tempo escrevendo agora não lhe trará recompensas imediatas, mas só ajudará muito tempo depois – talvez meses mais tarde, quando evitará que você introduza um bug enquanto faz uma refatoração, ou identificará uma regressão quando você fizer um upgrade de uma dependência. Ou, quem sabe, compensará de uma forma que é difícil de mensurar, incentivando você a escrever um código com um design melhor, mas você se convence de que poderia tê-lo escrito de modo igualmente elegante sem os testes.

Eu mesmo comecei a cometer deslizes quando estava escrevendo o framework de testes para este livro (<https://github.com/hjwp/Book-TDD-Web-Dev-Python/tree/master/tests>). Por ser um criatura

selvagem bastante complexa, ele tem testes próprios, mas tomei vários atalhos, a abrangência não é perfeita e agora me arrependo disso, pois o fato é que acabou ficando bem desajeitado e deselegante (vá em frente: eu deixei o código aberto agora, portanto todos podem apontar o dedo e rir).

## **Mantenha suas construções no CI com sinal verde**

Outra área que exige um esforço realmente árduo é a de integração contínua. Vimos no Capítulo 24 que bugs estranhos e imprevisíveis às vezes ocorrem no CI. Quando você estiver olhando para eles e pensando “funciona bem na minha máquina”, há uma forte tentação de simplesmente os ignorar... No entanto, se você não for cuidadoso, começará a tolerar uma suíte de testes com falha no CI, e, muito em breve, a sua construção no CI será realmente inútil, e voltar a executá-la novamente parecerá muito trabalhoso. Não caia nessa armadilha. Seja persistente, e você encontrará o motivo pelo qual o seu teste está falhando, e desvendará uma maneira de cercá-lo e torná-lo determinístico, ficando com o sinal verde novamente.

## **Tenha orgulho de seus testes, assim como você tem de seu código**

Um dos aspectos que pode ajudar é parar de pensar em seus testes como um acréscimo secundário ao código “real”, e começar a pensar neles como parte do produto final que você está construindo – uma parte que deve estar igualmente bem polida, deve ser igualmente agradável do ponto de vista estético e da qual você poderá igualmente se orgulhar em entregar...

Então faça isso porque o Testing Goat diz para fazer. Faça porque você sabe que a recompensa valerá a pena, mesmo que não seja imediata. Faça por um senso de dever ou por profissionalismo ou porque você tem TOC ou por pura teimosia. Faça porque é algo bom para ser exercitado. E, por fim, faça porque isso deixa o



desenvolvimento de software mais divertido.

## **Lembre-se de dar gorjetas para o pessoal do bar**

Este livro não teria sido possível sem o apoio de minha editora, a incrível O'Reilly Media. Se estiver lendo a edição gratuita online (em inglês), espero que considere a compra de uma cópia de verdade (<https://shop.oreilly.com/product/0636920051091.do>)... Se não precisar de uma para si mesmo, quem sabe como um presente para um amigo ou uma amiga?

## **Não seja um desconhecido!**

Espero que tenha gostado do livro. Entre em contato comigo e diga o que achou dele!

Harry – [@hjwp](https://twitter.com/hjwp) (<https://twitter.com/hjwp>) –  
[obeythetestinggoat@gmail.com](mailto:obeythetestinggoat@gmail.com)



# APÊNDICE A

## PythonAnywhere

Este livro está baseado no pressuposto de que você está executando Python e programando em seu próprio computador. É claro que essa não é a única maneira de programar em Python atualmente; você poderia utilizar uma plataforma online, como o PythonAnywhere (que, por acaso, é onde eu trabalho).

É possível acompanhar o livro com o PythonAnywhere, mas isso exige vários ajustes e mudanças – você precisará configurar uma aplicação web no lugar do servidor de testes, terá que usar o Xvfb para executar os Testes Funcionais e, quando chegar nos capítulos sobre implantação, precisará realizar um upgrade para uma conta paga. Portanto é possível, mas poderá ser mais fácil acompanhar o livro em seu próprio computador.

Com essa ressalva, se ainda estiver disposto a experimentar, apresentaremos alguns detalhes sobre o que você deve realizar.

Se ainda não o fez, será necessário fazer a inscrição para ter uma conta no PythonAnywhere. Uma conta gratuita deve ser suficiente.

Em seguida, inicie um *Bash Console* a partir da página de consoles. É aí que faremos a maior parte de nosso trabalho.

### **Executando sessões de Firefox no Selenium com o Xvfb**

A primeira informação é que o PythonAnywhere é um ambiente somente para console, portanto não tem um display no qual o Firefox será apresentado. Entretanto, podemos usar um display virtual.

No Capítulo 1, em que escrevemos o nosso primeiríssimo teste, você verá que nem tudo funciona conforme esperado. Eis a aparência do primeiro teste, e você poderá digitá-lo usando o editor do PythonAnywhere, sem problemas:

```
from selenium import webdriver
browser = webdriver.Firefox()
browser.get('http://localhost:8000')
assert 'Django' in browser.title
```

Porém, quando tentar executá-lo (em um *console Bash*), você verá um erro:

```
(superlists)$ python functional_tests.py
Traceback (most recent call last):
File "tests.py", line 3, in <module>
browser = webdriver.Firefox()
[...]
selenium.common.exceptions.WebDriverException: Message: 'geckodriver'
executable
needs to be in PATH.
```

Como o PythonAnywhere está associado a uma versão mais antiga do Firefox, não precisamos realmente do GeckoDriver. No entanto, teremos que voltar para o Selenium 2, em vez de usar o Selenium 3:

```
(superlists) $ pip install "selenium<3"
Collecting selenium<3
Installing collected packages: selenium
 Found existing installation: selenium 3.4.3
 Uninstalling selenium-3.4.3:
 Successfully uninstalled selenium-3.4.3
Successfully installed selenium-2.53.6
```

Agora, deparamos com um segundo problema:

```
(superlists)$ python functional_tests.py
Traceback (most recent call last):
File "tests.py", line 3, in <module>
browser = webdriver.Firefox()
[...]
selenium.common.exceptions.WebDriverException: Message: The browser
appears to
have exited before we could connect. If you specified a log_file in the
```

FirefoxBinary constructor, check it for details.

O Firefox não consegue iniciar porque não há nenhum display para que ele execute, pois o PythonAnywhere é um ambiente de servidor. A solução alternativa é usar o *Xvfb*, que significa X Virtual Framebuffer. Ele iniciará um display “virtual”, que o Firefox pode usar, apesar de o servidor não ter um display de verdade.

O comando `xvfb-run` executará o próximo comando no Xvfb. Ao usá-lo, veremos a nossa falha esperada:

```
(superlists)$ xvfb-run -a python functional_tests.py
Traceback (most recent call last):
 File "tests.py", line 11, in <module>
 assert 'Django' in browser.title
AssertionError
```

Então a lição é usar `xvfb-run -a` sempre que precisar executar os testes funcionais.

## Configurando o Django como uma aplicação web no PythonAnywhere

Logo depois disso, configuramos o Django usando o comando `django-admin.py startproject`. Porém, em vez de usar `manage.py runserver` para executar o servidor de desenvolvimento local, configuraremos o nosso site como uma verdadeira aplicação web no PythonAnywhere.

Acesse a aba Web e pressione o botão para adicionar uma nova aplicação web. Selecione “Manual configuration” (Configuração manual) e, em seguida, “Python 3.4”.

Na próxima tela, forneça o nome de seu virtualenv (“superlists”); quando fizer a submissão, ele deverá ser preenchido automaticamente para `/home/yourusername/.virtualenvs/superlists`.

Por fim, clique no link para *editar o seu arquivo wsgi*, localize e remova o caractere de comentário da seção para Django. Pressione Save (Salvar) e então Reload (Recarregar) para atualizar a sua aplicação web.

A partir de agora, em vez de executar o servidor de testes a partir de um console em localhost:8000, você poderá usar o verdadeiro URL de sua aplicação web no PythonAnywhere:

```
browser.get('http://my-username.pythonanywhere.com')
```



Será preciso lembrar de pressionar Reload sempre que você fizer alterações no código, a fim de atualizar o site.

Isso deve funcionar melhor.<sup>1</sup> Você deverá continuar usando esse padrão de apontar os FTs para a versão do site no PythonAnywhere e pressionar Reload antes de executar cada FT até o Capítulo 7, quando passamos a usar `LiveServerTestCase` e `self.live_server_url`.

## Limpendo o /tmp

O Selenium e o Xvfb tendem a deixar muito lixo no `/tmp`, especialmente quando não são encerrados de forma ordenada (é por isso que incluí um `try/finally` antes).

Com efeito, eles deixam tantos dados por lá que poderão consumir toda a sua quota de área de armazenagem. Portanto, faça uma limpeza em `/tmp` com certa frequência:

```
$ rm -rf /tmp/*
```

## Capturas de tela

No Capítulo 5, sugeri usar um `time.sleep` para fazer uma pausa no FT durante a sua execução, de modo que pudéssemos ver o que o navegador do Selenium estava mostrando na tela. Não podemos fazer isso no PythonAnywhere, pois o navegador executa em um display virtual. Como alternativa, é possível inspecionar o site live, ou “acreditar em minha palavra” no que concerne ao que você deverá ver.

A melhor maneira de fazer inspeções visuais de testes que executam em um display virtual é usar capturas de tela. Dê uma

olhada no Capítulo 24 se estiver curioso – há alguns códigos de exemplo lá.

## **O capítulo sobre implantação**

Quando chegar ao Capítulo 9, você terá a opção de continuar usando o PythonAnywhere ou aprender a construir um servidor “real”. Eu recomendo a última opção, pois você terá mais vantagens com ela.

Se realmente quiser ater-se ao PythonAnywhere, o que é de fato uma trapaça, poderá se inscrever em uma segunda conta aí e usá-lo como o seu site de staging. Ou você pode adicionar um segundo domínio à sua conta existente. Contudo, a maior parte das instruções do capítulo será irrelevante (não há necessidade de ter o Nginx nem o Gunicorn nem sockets de domínio no PythonAnywhere).

De uma forma ou de outra, nesse ponto, é provável que você precise de uma conta paga:

- Se quiser executar o seu site de staging em um domínio que não seja do PythonAnywhere
- Se quiser ser capaz de executar os FTs em um domínio que não seja do PythonAnywhere (porque ele não estará em nossa lista branca)
- Ao chegar no Capítulo 11, se quiser executar o Fabric em uma conta PythonAnywhere (porque você precisará do SSH)

Se quiser simplesmente “trapacear”, tente executar os FTs em modo “staging” em sua aplicação web existente e ignorar as partes relacionadas ao Fabric, embora seja uma enorme fraude, se quer saber a minha opinião. Ei, você sempre pode fazer um upgrade de sua conta e, então, cancelá-la de imediato, solicitando um reembolso dentro do período de 30 dias de garantia. ;)



Se estiver usando o PythonAnywhere para acompanhar o livro, gostaria muito de saber como você está se saindo! Envie um email para mim em [obeythetestinggoat@gmail.com](mailto:obeythetestinggoat@gmail.com).

---

- 1 Você *podia* executar o servidor de desenvolvimento do Django a partir de um console, mas o problema é que os consoles do PythonAnywhere nem sempre executam no mesmo servidor, portanto não haverá garantias de que o console em que você está executando seus testes seja o mesmo em que você estará executando o servidor. Além disso, ao executar no console, não há nenhuma maneira fácil de inspecionar visualmente a aparência do site.





## APÊNDICE B

# Views baseadas em classe do Django

Este apêndice dá continuidade ao Capítulo 15, no qual implementamos formulários do Django para validação e refatoramos nossas views. No final daquele capítulo, nossas views ainda usavam funções.

A grande novidade no mundo Django, porém, são as views baseadas em classe. Neste apêndice, vamos refatorar a nossa aplicação de modo a usá-las no lugar das funções de view. Mais especificamente, faremos uma tentativa de usar views *genéricas* baseadas em classe.

### Views genéricas baseadas em classe

Há uma diferença entre views baseadas em classe e views *genéricas* baseadas em classe. As CBVs (Class-based views, ou Views baseadas em classe) são apenas outra forma de definir funções de view. Elas fazem algumas suposições sobre o que suas views realizarão e oferecem uma vantagem principal em relação às funções de view, que é o fato de poderem ter subclasses. Sem dúvida, isso vem à custa de serem menos legíveis do que as views tradicionais, baseadas em funções. O caso de uso principal para views *simples* baseadas em classe é aquele em que temos várias views que reutilizam a mesma lógica. Queremos obedecer ao princípio DRY. Com views baseadas em funções, utilizaríamos funções auxiliares ou decoradores. A teoria é que usar uma estrutura de classes pode resultar em uma solução mais elegante.

As CBGVs (Class-based generic views, ou Views *genéricas* baseadas em classe) são views baseadas em classe que tentam oferecer soluções prontas para casos de uso comuns: buscar um objeto no banco de dados e passá-lo para um template, buscar uma lista de objetos, salvar dados de entrada de usuário provenientes de uma requisição POST usando um ModelForm, e assim por diante. Soam como muito semelhantes aos nossos casos de uso, mas, como veremos em breve, o problema está nos detalhes.

A essa altura, devo dizer que ainda não usei muito nenhum dos tipos de views baseadas em classe. Com certeza sou capaz de ver sentido nelas, e há possivelmente muitos casos de uso em aplicações Django em que as CBGVs seriam perfeitamente apropriadas. No entanto, assim que o seu caso de uso se desvia um pouco do básico – assim que você tiver mais de um modelo que queira usar, por exemplo –, acho que usar views baseadas em classe (novamente, é questionável) resulta em um código muito mais difícil de ler do que uma função de view clássica.

Apesar disso, como somos forçados a usar diversas opções de personalização para views baseadas em classe, implementá-las nesse caso pode nos ensinar bastante acerca de como elas funcionam e como criar seus testes de unidade.

Minha esperança é que os mesmos testes de unidade que usarmos para views baseadas em função deverão funcionar de forma igualmente apropriada em views baseadas em classe. Vamos ver como nos saímos.

## **Página inicial como uma FormView**

Nossa página inicial simplesmente exibe um formulário em um template:

### **lists/views.py**

```
def home_page(request):
 return render(request, 'home.html', {'form': ItemForm()})
```

Ao observar as opções (<https://docs.djangoproject.com/en/1.11/ref/class-based-views/>), vemos que o Django tem uma view genérica chamada FormView – vamos ver como é isso:

## lists/views.py (ch31I001)

```
from django.views.generic import FormView
[...]
```

```
class HomePageView(FormView):
 template_name = 'home.html'
 form_class = ItemForm
```

Nós lhe dizemos qual template e qual formulário queremos usar. Em seguida, basta atualizar *urls.py*, substituindo a linha que costumava conter `lists.views.home_page`:

## superlists/urls.py (ch31I002)

```
[...]
urlpatterns = [
 url(r'^$', list_views.HomePageView.as_view(), name='home'),
 url(r'^lists/', include(list_urls)),
]
```

Todos os testes passam! Essa foi fácil...

```
$ python manage.py test lists
[...]
```

```
Ran 34 tests in 0.119s
```

```
OK
```

```
$ python manage.py test functional_tests
```

```
[...]
```

```
Ran 5 tests in 15.160s
```

```
OK
```

Até agora, tudo bem. Substituímos uma função de view de uma só linha por uma classe com duas linhas, mas ela continua bem legível. É uma boa hora para um commit...

## Usando `form_valid` para personalizar uma `CreateView`

Em seguida, temos uma ruptura na view que usamos para criar uma nova lista – atualmente é a função `new_list`. Eis a sua aparência agora:

### `lists/views.py`

```
def new_list(request):
 form = ItemForm(data=request.POST)
 if form.is_valid():
 list_ = List.objects.create()
 form.save(for_list=list_)
 return redirect(list_)
 else:
 return render(request, 'home.html', {"form": form})
```

Observando as CBGVs possíveis, provavelmente vamos querer uma `CreateView`, e sabemos que estamos usando a classe `ItemForm`, portanto vamos ver como nos saímos com elas, e se os testes nos ajudarão:

### `lists/views.py (ch311003)`

```
from django.views.generic import FormView, CreateView
[...]

class NewListView(CreateView):
 form_class = ItemForm

def new_list(request):
 [...]
```

Deixarei a antiga função de view em `views.py`, de modo que possamos copiar o código a partir daí. Podemos apagá-la depois que tudo estiver funcionando. Ela será inócua assim que alterarmos os mapeamentos de URL, dessa vez em:

### `lists/urls.py (ch311004)`

```
[...]
```

```
urlpatterns = [
 url(r'^new$', views.NewListView.as_view(), name='new_list'),
 url(r'^(id+)/$', views.view_list, name='view_list'),
]
```

Agora a execução dos testes resulta em seis erros:

```
$ python manage.py test lists
```

```
[...]
```

```
ERROR: test_can_save_a_POST_request (lists.tests.test_views.NewListTest)
TypeError: save() missing 1 required positional argument: 'for_list'
```

```
ERROR: test_for_invalid_input_passes_form_to_template
(lists.tests.test_views.NewListTest)
django.core.exceptions.ImproperlyConfigured: TemplateResponseMixin requires
either a definition of 'template_name' or an implementation of
'get_template_names()'
```

```
ERROR: test_for_invalid_input_renders_home_template
(lists.tests.test_views.NewListTest)
django.core.exceptions.ImproperlyConfigured: TemplateResponseMixin requires
either a definition of 'template_name' or an implementation of
'get_template_names()'
```

```
ERROR: test_invalid_list_items_arent_saved (lists.tests.test_views.NewListTest)
django.core.exceptions.ImproperlyConfigured: TemplateResponseMixin requires
either a definition of 'template_name' or an implementation of
'get_template_names()'
```

```
ERROR: test_redirects_after_POST (lists.tests.test_views.NewListTest)
TypeError: save() missing 1 required positional argument: 'for_list'
```

```
ERROR: test_validation_errors_are_shown_on_home_page
(lists.tests.test_views.NewListTest)
django.core.exceptions.ImproperlyConfigured: TemplateResponseMixin requires
either a definition of 'template_name' or an implementation of
'get_template_names()'
```

```
FAILED (errors=6)
```

Vamos começar pelo terceiro – talvez possamos simplesmente adicionar o template?

## lists/views.py (ch31I005)

```
class NewListView(CreateView):
 form_class = ItemForm
 template_name = 'home.html'
```

Agora nos restam somente duas falhas: podemos ver que ambas ocorrem na função `form_valid` da view genérica, e essa é uma das que podem ser sobrescritas para oferecer um comportamento personalizado em uma CBGV. Como o nome implica, ela é executada quando a view detectar um formulário válido. Podemos apenas copiar parte do código de nossa antiga função de view, que costumava estar após `if form.is_valid():`:

## lists/views.py (ch31I006)

```
class NewListView(CreateView):
 template_name = 'home.html'
 form_class = ItemForm

 def form_valid(self, form):
 list_ = List.objects.create()
 form.save(for_list=list_)
 return redirect(list_)
```

Com isso, todos os testes passam!

```
$ python manage.py test lists
Ran 34 tests in 0.119s
OK
$ python manage.py test functional_tests
Ran 5 tests in 15.157s
OK
```

*Poderíamos* até mesmo economizar mais duas linhas, tentando obedecer ao princípio “DRY”, usando uma das principais vantagens das CBVs: a herança!

## lists/views.py (ch31I007)

```
class NewListView(CreateView, HomePageView):
```

```
 def form_valid(self, form):
 list_ = List.objects.create()
 form.save(for_list=list_)
 return redirect(list_)
```

Todos os testes continuariam passando:

OK



Essa não é realmente uma boa prática de orientação a objetos. A herança implica um relacionamento “é um” (is-a), e provavelmente não faz sentido dizer que a nossa nova view de lista “é uma” view de página inicial... Desse modo, provavelmente é melhor não fazer isso.

Com ou sem esse último passo, como esse código se compara à versão antiga? Eu diria que não é ruim. Evitamos um pouco de código boilerplate, e a view continua razoavelmente legível. Até agora, eu diria que temos um ponto para as CBGVs e um empate.

## **Uma view mais complexa para lidar tanto com a visualização quanto com a adição em uma lista**

Isso me custou *várias* tentativas. Devo dizer que, embora os testes tenham me dito quando eu havia feito algo correto, eles realmente não me ajudaram a identificar os passos para chegar lá... Em sua maior parte, foram apenas tentativas e erros, lidando com funções como `get_context_data`, `get_form_kwargs` e assim por diante.

Um dos pontos que pude perceber foi a importância de ter muitos testes individuais, cada um para testar um só aspecto. Como resultado, voltei atrás e reescrevi alguns dos testes dos Capítulos 10 a 12.

## **Os testes nos orientam por um tempo**

Eis o modo como a situação pode se desenrolar. Comece pensando que queremos uma `DetailView` – algo que mostre os detalhes de um



objeto a você:

## lists/views.py (ch311009)

```
from django.views.generic import FormView, CreateView, DetailView
[...]
```

```
class ViewAndAddToList(DetailView):
 model = List
```

Faça a sua associação em *urls.py*:

## lists/urls.py (ch311010)

```
url(r'^(\d+)/$', views.ViewAndAddToList.as_view(), name='view_list'),
```

O resultado será este:

```
[...]
```

```
AttributeError: Generic detail view ViewAndAddToList must be called with either
an object pk or a slug.
```

```
FAILED (failures=5, errors=6)
```

Não é totalmente óbvio, mas um pouco de pesquisa por aí no Google me levou a entender que eu precisava usar um grupo de captura “nameado” de regex:

## lists/urls.py (ch311011)

```
@@ -3,6 +3,6 @@ from lists import views
```

```
urlpatterns = [
 url(r'^new$', views.NewListView.as_view(), name='new_list'),
 - url(r'^(\d+)/$', views.view_list, name='view_list'),
 + url(r'^(?P<pk>\d+)/$', views.ViewAndAddToList.as_view(), name='view_list')
]
```

O próximo conjunto de erros continha um que foi razoavelmente prestativo:

```
[...]
```

```
django.template.exceptions.TemplateDoesNotExist: lists/list_detail.html
```

FAILED (failures=5, errors=6)

Esse foi facilmente resolvido:

## lists/views.py (ch311012)

```
class ViewAndAddToList(DetailView):
 model = List
 template_name = 'list.html'
```

Ficamos reduzidos a cinco e dois:

```
[...]
ERROR: test_displays_item_form (lists.tests.test_views.ListViewTest)
KeyError: 'form'
```

FAILED (failures=5, errors=2)

## Até sermos reduzidos a tentativa e erro

Assim, descobri que nossa view não só mostrava os detalhes de um objeto, mas também nos permitia criar novos objetos. Vamos fazer dela tanto uma *DetailView* quanto uma *CreateView* e talvez adicionar `form_class`:

## lists/views.py (ch311013)

```
class ViewAndAddToList(DetailView, CreateView):
 model = List
 template_name = 'list.html'
 form_class = ExistingListItemForm
```

Porém, isso resulta em muitos erros informando o seguinte:

```
[...]
TypeError: __init__() missing 1 required positional argument: 'for_list'
```

E o `KeyError: 'form'` continuava lá também!

Nesse ponto, os erros deixaram de ser tão prestativos, e o que devíamos fazer em seguida não era mais tão óbvio. Tive que recorrer às tentativas e aos erros. Apesar disso, os testes pelo menos me diziam quando eu estava fazendo algo mais correto ou mais incorreto.

Minhas primeiras tentativas de usar `get_form_kwargs` não funcionaram realmente, mas descobri que eu poderia usar `get_form`:

## lists/views.py (ch311014)

```
def get_form(self):
 self.object = self.get_object()
 return self.form_class(for_list=self.object, data=self.request.POST)
```

Contudo, isso só funcionaria se eu também fizesse uma atribuição a `self.object`, como efeito colateral, durante o processo, o que era um pouco irritante. No entanto, com isso ficamos reduzidos a apenas três erros, mas aparentemente ainda não chegamos lá!

```
django.core.exceptions.ImproperlyConfigured: No URL to redirect to. Either
provide a url or define a get_absolute_url method on the Model.
```

## De volta ao caminho

Para essa falha final, os testes estão sendo úteis novamente. É bem fácil definir um `get_absolute_url` na classe `Item`, de modo que os itens apontem para a página de sua lista-pai:

## lists/models.py (ch311015)

```
class Item(models.Model):
 [...]

 def get_absolute_url(self):
 return reverse('view_list', args=[self.list.id])
```

## Essa é a sua resposta final?

Acabamos com uma classe de view com a seguinte aparência:

## lists/views.py

```
class ViewAndAddToList(DetailView, CreateView):
 model = List
 template_name = 'list.html'
 form_class = ExistingListItemForm

 def get_form(self):
```

```
self.object = self.get_object()
return self.form_class(for_list=self.object, data=self.request.POST)
```

## Comparando o antigo e o novo

Vamos ver a versão antiga para comparar?

### lists/views.py

```
def view_list(request, list_id):
 list_ = List.objects.get(id=list_id)
 form = ExistingListItemForm(for_list=list_)
 if request.method == 'POST':
 form = ExistingListItemForm(for_list=list_, data=request.POST)
 if form.is_valid():
 form.save()
 return redirect(list_)
 return render(request, 'list.html', {'list': list_, "form": form})
```

Bem, o número de linhas de código foi reduzido de novo para sete. Apesar disso, acho a versão baseada em função um pouco mais fácil de entender, pois tem um pouco menos de mágica – “explícito é melhor do que implícito”, como diria o Zen de Python. Quero dizer... SingleObjectMixin? O quê? Pior ainda, tudo se desestrutura se não fizermos uma atribuição para `self.object` em `get_form`? Argh.

Contudo, acho que parte disso está no olhar de quem está vendo.

## Melhores práticas para testes de unidade de CBGVs?

Enquanto eu estava trabalhando neste texto, achei que meus testes de “unidade” às vezes eram demasiadamente de alto nível. Não é nenhuma surpresa, pois os testes para views que envolvem o Django Test Client provavelmente sejam mais apropriadamente chamados de testes integrados.

Eles me diziam se eu estava fazendo algo certo ou errado, mas nem sempre ofereciam pistas suficientes sobre como fazer as correções.

Ocasionalmente, fiquei me perguntando se haveria alguma

vantagem em um teste que fosse mais próximo da implementação – algo como:

### lists/tests/test\_views.py

```
def test_cbv_gets_correct_object(self):
 our_list = List.objects.create()
 view = ViewAndAddToList()
 view.kwargs = dict(pk=our_list.id)
 self.assertEqual(view.get_object(), our_list)
```

Todavia o problema é que isso exige muito conhecimento do funcionamento interno das CBVs do Django a fim de fazermos a configuração correta para esses tipos de testes. Você ainda continuará muito confuso por causa da hierarquia de herança complexa.

## **Resultado final: ter vários testes de view isolados com asserções únicas ajuda**

Uma conclusão a que, sem dúvida, cheguei com este apêndice foi o fato de que ter muitos testes de unidade pequenos para views era muito mais conveniente do que ter alguns testes como uma série de asserções narrativas.

Considere o seguinte teste monolítico:

### lists/tests/test\_views.py

```
def test_validation_errors_sent_back_to_home_page_template(self):
 response = self.client.post('/lists/new', data={'text': ''})
 self.assertEqual(List.objects.all().count(), 0)
 self.assertEqual(Item.objects.all().count(), 0)
 self.assertTemplateUsed(response, 'home.html')
 expected_error = escape("You can't have an empty list item")
 self.assertContains(response, expected_error)
```

Definitivamente, é menos útil do que ter três testes individuais, assim:

### lists/tests/test\_views.py

```
def test_invalid_input_means_nothing_saved_to_db(self):
 self.post_invalid_input()
 self.assertEqual(List.objects.all().count(), 0)
 self.assertEqual(Item.objects.all().count(), 0)

def test_invalid_input_renders_list_template(self):
 response = self.post_invalid_input()
 self.assertTemplateUsed(response, 'list.html')

def test_invalid_input_renders_form_with_errors(self):
 response = self.post_invalid_input()
 self.assertIsInstance(response.context['form'], ExistingListItemForm)
 self.assertContains(response, escape(empty_list_error))
```

O motivo para isso é que, no primeiro caso, uma falha prematura significa que nem todas as asserções serão verificadas. Assim, se a view estivesse acidentalmente salvando dados de um POST inválido no banco de dados, você obteria uma falha mais cedo e, desse modo, não descobriria se ela estava usando o template correto ou renderizando o formulário. A segunda construção facilita bastante identificar de fato o que estava ou não funcionando.

## **Lições aprendidas com as CBGVs**

## *As views genéricas baseadas em classe podem fazer de tudo*

Talvez nem sempre esteja claro o que está acontecendo, mas você pode fazer praticamente de tudo com views genéricas baseadas em classe.

### *Testes de unidade com uma única asserção ajudam na refatoração*

Com cada teste de unidade oferecendo orientação individual sobre o que funciona e o que não funciona, é muito mais fácil alterar a implementação de nossas views para que usem esse paradigma fundamentalmente diferente.





# APÊNDICE C

## Provisionamento com o Ansible

Usamos o Fabric para automatizar a implantação de novas versões do código-fonte em nossos servidores. Contudo, fazer o provisionamento de um novo servidor e atualizar os arquivos de configuração do Nginx e do Gunicorn foram deixados como processos manuais.

Esse é o tipo de tarefa cada vez mais atribuída a ferramentas chamadas de “Gerenciamento de Configuração” (Configuration Management) ou “Implantação Contínua” (Continuous Deployment). Chef e Puppet foram as primeiras bem conhecidas; no mundo Python, temos o Salt e o Ansible.

De todas essas ferramentas, o Ansible é a mais fácil para começar a usar. Podemos fazê-lo funcionar com apenas dois arquivos:

```
pip2 install --user ansible # Python 2 infelizmente
```

Um “arquivo de inventário” em *deploy\_tools/inventory.ansible* define em quais servidores podemos executá-lo:

### deploy\_tools/inventory.ansible

```
[live]
```

```
superlists.ottg.eu ansible_become=yes ansible_ssh_user=elspeth
```

```
[staging]
```

```
superlists-staging.ottg.eu ansible_become=yes ansible_ssh_user=elspeth
```

```
[local]
```

```
localhost ansible_ssh_user=root ansible_ssh_port=6666 ansible_host=127.0.0.1
```

(A entrada local é apenas um exemplo; no meu caso, é uma VM do Virtualbox, configurada com encaminhamento de porta para as

portas 22 e 80).

## Instalando pacotes de sistema e o Nginx

Em seguida, vemos o “playbook” (manual de regras) do Ansible, que define o que fazer no servidor. Uma sintaxe chamada YAML é usada:

### deploy\_tools/provision.ansible.yaml

---

- hosts: all

vars:

host: "{{ inventory\_hostname }}"

tasks:

- name: Deadsnakes PPA to get Python 3.6

apt\_repository:

repo='ppa:fkruill/deadsnakes'

- name: make sure required packages are installed

apt: pkg=nginx,git,python3.6,python3.6-venv state=present

- name: allow long hostnames in nginx

lineinfile:

dest=/etc/nginx/nginx.conf

regexp='(\s+)#? ?server\_names\_hash\_bucket\_size'

backrefs=yes

line='\1server\_names\_hash\_bucket\_size 64;'

- name: add nginx config to sites-available

template: src=./nginx.conf.j2 dest=/etc/nginx/sites-available/{{ host }}

notify:

- restart nginx

- name: add symlink in nginx sites-enabled

file:

src=/etc/nginx/sites-available/{{ host }}

dest=/etc/nginx/sites-enabled/{{ host }}

```
state=link
notify:
 - restart nginx
```

A variável `inventory_hostname` é o nome de domínio do servidor para o qual estamos executando. Estou usando a seção `vars` a fim de renomeá-lo para “host”, somente por conveniência.

Nesta seção, instalamos os softwares necessários usando `apt`, ajustamos a configuração do Nginx para permitir nomes de host longos usando substituição com expressões regulares e então escrevemos o arquivo de configuração do Nginx usando um template. Essa é uma versão modificada do arquivo de template que salvamos em `deploy_tools/nginx.template.conf` no Capítulo 9, mas ela agora utiliza uma sintaxe específica de template – o Jinja2, que, na verdade, é muito parecida com a sintaxe de template do Django:

## deploy\_tools/nginx.conf.j2

```
server {
 listen 80;
 server_name {{ host }};

 location /static {
 alias /home/{{ ansible_ssh_user }}/sites/{{ host }}/static;
 }

 location / {
 proxy_set_header Host {{ host }};
 proxy_pass http://unix:/tmp/{{ host }}.socket;
 }
}
```

## Configurando o Gunicorn e usando handlers para reiniciar serviços

Eis a segunda parte de nosso playbook:

## deploy\_tools/provision.ansible.yaml

```
- name: write gunicorn service script
```

```
template:
 src=./gunicorn.service.j2
 dest=/etc/systemd/system/gunicorn-{{ host }}.service
notify:
 - restart gunicorn
```

```
handlers:
 - name: restart nginx
 service: name=nginx state=restarted

 - name: restart gunicorn
 systemd:
 name=gunicorn-{{ host }}
 daemon_reload=yes
 enabled=yes
 state=restarted
```

Mais uma vez, usamos um template para a nossa configuração do Gunicorn:

## deploy\_tools/gunicorn.service.j2

```
[Unit]
Description=Gunicorn server for {{ host }}

[Service]
User={{ ansible_ssh_user }}
WorkingDirectory=/home/{{ ansible_ssh_user }}/sites/{{ host }}/source
Restart=on-failure
ExecStart=/home/{{ ansible_ssh_user }}/sites/{{ host }}/virtualenv/bin/gunicorn \
 --bind unix:/tmp/{{ host }}.socket \
 --access-logfile ../access.log \
 --error-logfile ../error.log \
 superlists.wsgi:application

[Install]
WantedBy=multi-user.target
```

Então temos dois “handlers” para reiniciar o Nginx e o Gunicorn. O Ansible é inteligente, de modo que, se vir vários passos chamando os mesmos handlers, ele esperará até o último para chamá-lo.

É isso! O comando para disparar tudo isso é:

```
ansible-playbook -i inventory.ansible provision.ansible.yaml --limit=staging --ask-become-pass
```

Há muito mais informações na documentação do Ansible (<https://docs.ansible.com/>).

## O que fazer em seguida

Acabei de dar uma pequena amostra do que é possível fazer com o Ansible. No entanto, quanto mais você automatizar suas implantações, mais confiança você terá nelas. A seguir, apresentamos mais alguns pontos que você deverá verificar.

## Passe as implantações do Fabric para o Ansible

Vimos que o Ansible é capaz de ajudar em alguns aspectos do provisionamento, mas também pode fazer praticamente toda a nossa implantação para nós. Veja se você consegue estender o playbook de modo a realizar tudo que fazemos atualmente com o nosso script de implantação do Fabric, incluindo notificar as reinicializações, conforme necessário.

## Use o Vagrant para iniciar uma VM local

Executar os testes no site de staging, em última instância, nos dá a confiança de que tudo funcionará quando estivermos no ambiente live, mas também podemos usar uma VM em nossa máquina local.

Faça download do Vagrant e do Virtualbox, e veja se você consegue fazer o Vagrant construir um servidor de desenvolvimento em seu próprio computador usando o nosso playbook do Ansible para implantar aí o código. Adapte o executor do FT para testar na VM local.

Ter um arquivo de configuração do Vagrant é particularmente útil quando trabalhamos em equipe – isso ajuda os novos desenvolvedores a iniciar servidores que se pareçam exatamente

como o seu.





## APÊNDICE D

# Testando migrações de banco de dados

As migrações do Django e de seu antecessor, o South, existem há anos, portanto, geralmente não é necessário testar migrações de banco de dados. Porém, o fato é que, por acaso, estamos introduzindo um tipo perigoso de migração – isto é, um que introduz uma nova restrição de integridade em nossos dados. Quando executei o script de migração no staging pela primeira vez, vi um erro.

Em projetos maiores, em que há dados confidenciais, talvez você queira ter um nível adicional de confiança resultante do teste de suas migrações em um ambiente seguro antes de aplicá-las em dados de produção, portanto esperamos que esse exemplo lúdico seja um ensaio útil.

Outro motivo comum para querer testar as migrações é a velocidade – com frequência, as migrações envolvem downtime (tempo inativo) e, às vezes, quando são aplicadas em conjuntos de dados muito grandes, elas podem demorar. É bom saber com antecedência quanto tempo será necessário.

## **Uma tentativa de implantação no ambiente de staging**

Eis o que aconteceu comigo na primeira vez que tentei fazer a implantação de nossas novas restrições de validação no Capítulo 17:

```
$ cd deploy_tools
$ fab deploy:host=elspeth@superlists-staging.ottg.eu
[...]
Running migrations:
 Applying lists.0005_list_item_unique_together...Traceback (most recent call
last):
 File "/usr/local/lib/python3.6/dist-packages/django/db/backends/utils.py",
line 61, in execute
 return self.cursor.execute(sql, params)
 File
"/usr/local/lib/python3.6/dist-packages/django/db/backends/sqlite3/base.py",
line 475, in execute
 return Database.Cursor.execute(self, query, params)
sqlite3.IntegrityError: columns list_id, text are not unique
[...]
```

O que aconteceu é que alguns dos dados existentes no banco de dados violaram a restrição de integridade, de modo que o banco de dados reclamou quando tentei aplicá-la.

Para lidar com esse tipo de problema, precisaremos construir uma “migração de dados”. Vamos inicialmente configurar um ambiente local para fazermos o teste.

## Executando uma migração de testes localmente

Usaremos uma cópia do banco de dados de produção para testar a nossa migração.



Tome muito, muito, muito cuidado quando usar dados reais para testes. Por exemplo, você poderá ter endereços de email de clientes reais aí, e não vai querer enviar-lhes acidentalmente uma porção de emails de teste. Pergunte-me como sei disso.

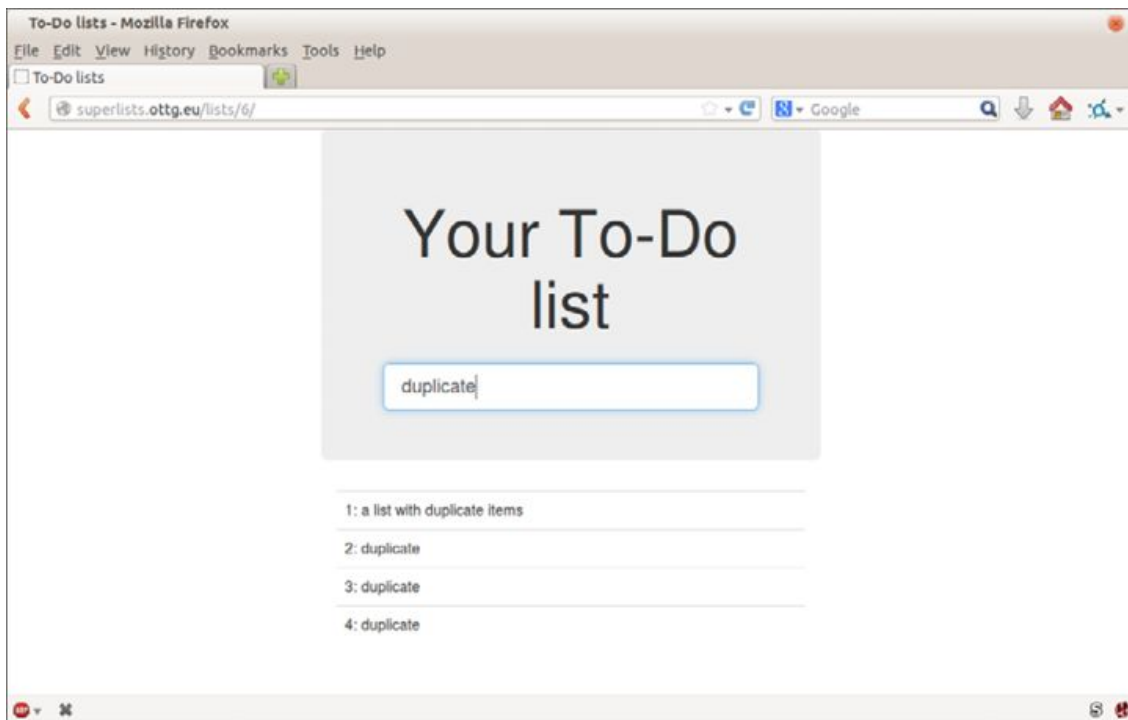
## Fornecendo dados problemáticos

Inicie uma lista com alguns itens duplicados em seu site live, como vemos na Figura D.1.

# Copiando dados de teste do site live

Copie o banco de dados do site live:

```
$ scp elspeth@superlists.ottg.eu:\
/home/elspeth/sites/superlists.ottg.eu/database/db.sqlite3 .
$ mv ../database/db.sqlite3 ../database/db.sqlite3.bak
$ mv db.sqlite3 ../database/db.sqlite3
```



*Figura D.1 – Uma lista com itens duplicados.*

## Confirmando o erro

Agora temos um banco de dados local que não foi migrado e que contém alguns dados problemáticos. Devemos ver um erro se tentarmos executar migrate:

```
$ python manage.py migrate --migrate
python manage.py migrate
Operations to perform:
[...]
Running migrations:
[...]
Applying lists.0005_list_item_unique_together...Traceback (most recent call
last):
```

[...]

```
return Database.Cursor.execute(self, query, params)
sqlite3.IntegrityError: columns list_id, text are not unique
```

## Inserindo uma migração de dados

Migrações de dados (<https://docs.djangoproject.com/en/1.11/topics/migrations/#data-migrations>) são um tipo especial de migração que modifica dados do banco de dados, em vez de alterar o esquema. Devemos criar uma que executará antes de aplicarmos a restrição de integridade, a fim de remover qualquer duplicação de dados de forma preventiva. Eis o modo de fazer isso:

```
$ git rm lists/migrations/0005_list_item_unique_together.py
$ python manage.py makemigrations lists --empty
Migrations for 'lists':
 0005_auto_20140414_2325.py:
$ mv lists/migrations/0005_*.py
lists/migrations/0005_remove_duplicates.py
```

Consulte a documentação do Django sobre migrações de dados (<https://docs.djangoproject.com/en/1.11/topics/migrations/#data-migrations>) para obter mais informações, mas eis o modo de adicionar algumas instruções para alterar dados existentes:

### lists/migrations/0005\_remove\_duplicates.py

```
encoding: utf8
from django.db import models, migrations

def find_dupes(apps, schema_editor):
 List = apps.get_model("lists", "List")
 for list_ in List.objects.all():
 items = list_.item_set.all()
 texts = set()
 for ix, item in enumerate(items):
 if item.text in texts:
 item.text = '{} ({}).format(item.text, ix)
 item.save()
 texts.add(item.text)
```

```
class Migration(migrations.Migration):

 dependencies = [
 ('lists', '0004_item_list'),
]

 operations = [
 migrations.RunPython(find_dupes),
]
```

## Recriando a migração antiga

Recriamos a migração antiga usando makemigrations, que garantirá que essa agora é a sexta migração e que tem uma dependência explícita da 0005, a migração de dados:

```
$ python manage.py makemigrations
Migrations for 'lists':
 0006_auto_20140415_0018.py:
 - Alter unique_together for item (1 constraints)
$ mv lists/migrations/0006_* lists/migrations/0006_unique_together.py
```

## Testando as novas migrações juntas

Agora estamos prontos para executar o nosso teste nos dados live:

```
$ cd deploy_tools
$ fab deploy:host=elspeth@superlists-staging.ottg.eu
[...]
```

Precisaremos reiniciar o job do Gunicorn live também:

```
elspeth@server:$ sudo systemctl restart gunicorn-superlists.ottg.eu
```

Agora podemos executar nossos FTs no ambiente de staging:

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test
functional_tests
[...]
```

```

Ran 4 tests in 17.308s
```

OK

Tudo parece em ordem! Vamos fazer isso no servidor live:

```
$ fab deploy --host=superlists.ottg.eu
[superlists.ottg.eu] Executing task 'deploy'
[...]
```

Com isso, concluímos a tarefa. Execute `git add lists/migrations`, `git commit` e assim por diante.

## Conclusões

Esse exercício tinha como objetivo principal construir uma migração de dados e testá-la em alguns dados reais. Inevitavelmente, isso é apenas uma gota no oceano dos possíveis testes que você poderia executar para uma migração. É possível pensar em construir testes automatizados para verificar se todos os seus dados foram preservados, comparando o conteúdo do banco de dados antes e depois. Você poderia escrever testes de unidade individuais para as funções auxiliares em uma migração de dados. Poderia investir mais tempo calculando o tempo consumido pelas migrações e fazendo experimentos com maneiras de agilizá-las, por exemplo, dividindo as migrações em mais ou menos passos.

Lembre-se de que esse deve ser um caso relativamente raro. Em minha experiência, não senti necessidade de testar 99% das migrações em que trabalhei. Todavia, caso, em algum momento, você sinta essa necessidade em seu projeto, espero que tenha encontrado algumas referências aqui para começar a trabalhar.

## Sobre testes de migrações de banco de dados

### *Tome cuidado com migrações que introduzam restrições*

99% das migrações ocorrem sem percalços, mas tome cuidado com qualquer situação, como essa, em que você esteja introduzindo uma nova restrição em colunas que já existam.

### *Teste as migrações quanto à velocidade*

Quando tiver um projeto maior, você deverá pensar em testar quanto tempo suas migrações consumirão. Geralmente as migrações de banco de dados envolvem downtime, pois, conforme o seu banco de dados, a operação de atualização de esquema poderá bloquear a tabela em que elas estiverem atuando até que sejam concluídas. Usar seu site de staging para descobrir quanto tempo uma migração consumirá é uma boa ideia.



## *Seja extremamente cuidadoso se estiver usando um dump dos dados de produção*

Para isso, você deverá preencher o banco de dados de seu site de staging com um volume de dados que seja proporcional ao volume dos seus dados de produção. Explicar como fazer isso está fora do escopo deste livro, mas digo o seguinte: se você se sentir tentado a simplesmente obter um dump do seu banco de dados de produção e carregá-lo no ambiente de staging, tome  *muito* cuidado. Dados de produção contêm detalhes de clientes de verdade e, pessoalmente, já fui responsável por enviar acidentalmente algumas centenas de faturas incorretas depois que um processo automatizado em meu servidor de staging começou a processar os dados copiados de produção que eu havia acabado de carregar ali. Não foi uma tarde divertida.



## APÊNDICE E

# Desenvolvimento orientado a comportamento (BDD)

Não usei BDD (Behaviour-Driven Development, ou Desenvolvimento Orientado a Comportamento) “furiosamente”, portanto não posso argumentar que tenho qualquer tipo de expertise, mas de fato gosto do que já vi e achei que você mereceria pelo menos um tour rápido. Neste apêndice, usaremos alguns testes que escrevemos em um FT “normal” e os converteremos usando ferramentas de BDD.

### O que é BDD?

*Estritamente* falando, BDD é uma metodologia, e não um conjunto de ferramentas – é a abordagem segundo a qual testamos a sua aplicação testando o comportamento que esperamos que ela exiba a um usuário (a entrada na Wikipédia em [https://en.wikipedia.org/wiki/Behavior-driven\\_development](https://en.wikipedia.org/wiki/Behavior-driven_development) tem uma boa apresentação geral). Assim, em certos aspectos, os FTs baseados em Selenium que apresentei no restante do livro *poderiam* ser chamados de BDD.

Contudo, o termo se tornou intimamente associado a um conjunto particular de ferramentas para BDD, principalmente à sintaxe Gherkin (<https://github.com/cucumber/cucumber/wiki/Gherkin>), que é uma DSL legível aos seres humanos para escrever testes funcionais (ou de aceitação). O Gherkin, originalmente, surgiu no mundo Ruby, associado a um executor de testes chamado Cucumber (<http://cukes.info/>).

No mundo Python, temos duas ferramentas equivalentes de execução de testes: Lettuce (<http://lettuce.it/>) e Behave (<http://pythonhosted.org/pytest-behavior/>). Entre essas ferramentas, somente o Behave era compatível com o Python 3 quando escrevi este livro, portanto é ele que usaremos. Também utilizaremos um plugin chamado behave-django (<https://pythonhosted.org/pytest-behavior-django/>).

## Obtendo o código para esses exemplos

Usarei o exemplo do Capítulo 22. Temos um site básico de listas de tarefas e queremos acrescentar uma nova funcionalidade: usuários logados devem ser capazes de visualizar as listas que criaram em um só lugar. Até esse ponto, todas as listas eram efetivamente anônimas.

Se você vem acompanhando o livro, vou supor que poderá retornar ao código naquele ponto. Se quiser extraí-lo de meu repositório, o local a acessar é o branch `chapter_17` ([https://github.com/hjwp/book-example/tree/chapter\\_17](https://github.com/hjwp/book-example/tree/chapter_17)).

## Organização básica

Criamos um diretório para nossas “features” (funcionalidades) de BDD, adicionamos um diretório `steps` (descobriremos para que servem, em breve!) e um placeholder para a nossa primeira feature:

```
$ mkdir -p features/steps
$ touch features/my_lists.feature
$ touch features/steps/my_lists.py
$ tree features
features
├── my_lists.feature
└── steps
 └── my_lists.py
```

Instalamos o behave-django e o adicionamos em `settings.py`:

```
$ pip install behave-django
```

```
superlists/settings.py
```

```
--- a/superlists/settings.py
```

```
+++ b/superlists/settings.py
@@ -40,6 +40,7 @@ INSTALLED_APPS = [
 'lists',
 'accounts',
 'functional_tests',
+ 'behave_django',
]
```

Então execute `python manage.py behave` para uma verificação de sanidade:

```
$ python manage.py behave
Creating test database for alias 'default'...
0 features passed, 0 failed, 0 skipped
0 scenarios passed, 0 failed, 0 skipped
0 steps passed, 0 failed, 0 skipped, 0 undefined
Took 0m0.000s
Destroying test database for alias 'default'...
```

## Escrevendo um FT como uma “feature” usando a sintaxe Gherkin

Até agora, escrevemos nossos FTs usando comentários legíveis aos seres humanos, que descrevem a nova funcionalidade em termos de uma história de usuário, intercalados com o código Selenium necessário para executar cada passo da história.

O BDD impõe uma distinção entre ambos – escrevemos nossa história legível aos seres humanos usando uma sintaxe legível a eles (embora, ocasionalmente, seja meio desajeitada), chamada “Gherkin”, e isso é chamado de “Feature” (Funcionalidade). Mais tarde mapearemos cada linha do Gherkin a uma função contendo o código Selenium necessário para implementar esse “passo” (step).

Eis como seria a aparência de uma Feature para a nossa nova página “My lists”:

### features/my\_lists.feature

```
Feature: My Lists
 As a logged-in user
```

I want to be able to see all my lists in one page  
So that I can find them all after I've written them

Scenario: Create two lists and see them on the My Lists page

Given I am a logged-in user

When I create a list with first item "Reticulate Splines"  
And I add an item "Immanentize Eschaton"  
And I create a list with first item "Buy milk"

Then I will see a link to "My lists"

When I click the link to "My lists"  
Then I will see a link to "Reticulate Splines"  
And I will see a link to "Buy milk"

When I click the link to "Reticulate Splines"  
Then I will be on the "Reticulate Splines" list page

## **As-a /I want to/So that**

No início, você perceberá a cláusula As-a/I want to/So that. Ela é opcional e não tem nenhuma contrapartida executável – é apenas uma maneira levemente formal de capturar os aspectos ligados a “quem e por quê” de uma história de usuário, gentilmente incentivando a equipe a pensar nas justificativas para cada feature.

## **Given/When/Then**

Given/When/Then compõem o verdadeiro núcleo de um teste de BDD. Essa construção tripla corresponde ao padrão configurar/exercitar/fazer asserção, que vimos em nossos testes de unidade, e representa a fase de configuração e pressuposições, uma fase de exercício/ação e uma fase subsequente de asserção/observação. Há mais informações na wiki do Cucumber (<https://github.com/cucumber/cucumber/wiki/Given-When-Then>).

## Nem sempre encaixa perfeitamente!

Como podemos ver, nem sempre é fácil encaixar uma história de usuário exatamente em três passos! Podemos usar a cláusula And para expandir um passo, e adicionei vários passos When e Then subsequentes para ilustrar outros aspectos de nossa página “My lists”.

## Implementando as funções para os passos

Construiremos agora a contrapartida de nossa feature descrita com a sintaxe Gherkin, que são as funções de “step” (passos) para implementá-las no código.

## Gerando passos placeholders

Quando executamos behave, ele nos informa solicitamente quais são todos os passos que devemos implementar:

```
$ python manage.py behave
```

```
Feature: My Lists # features/my_lists.feature:1
```

```
As a logged-in user
```

```
I want to be able to see all my lists in one page
```

```
So that I can find them all after I've written them
```

```
Scenario: Create two lists and see them on the My Lists page #
```

```
features/my_lists.feature:6
```

```
Given I am a logged-in user # None
```

```
Given I am a logged-in user # None
```

```
When I create a list with first item "Reticulate Splines" # None
```

```
And I add an item "Immanentize Eschaton" # None
```

```
And I create a list with first item "Buy milk" # None
```

```
Then I will see a link to "My lists" # None
```

```
When I click the link to "My lists" # None
```

```
Then I will see a link to "Reticulate Splines" # None
```

```
And I will see a link to "Buy milk" # None
```

```
When I click the link to "Reticulate Splines" # None
```

```
Then I will be on the "Reticulate Splines" list page # None
```

Failing scenarios:

features/my\_lists.feature:6 Create two lists and see them on the My Lists page

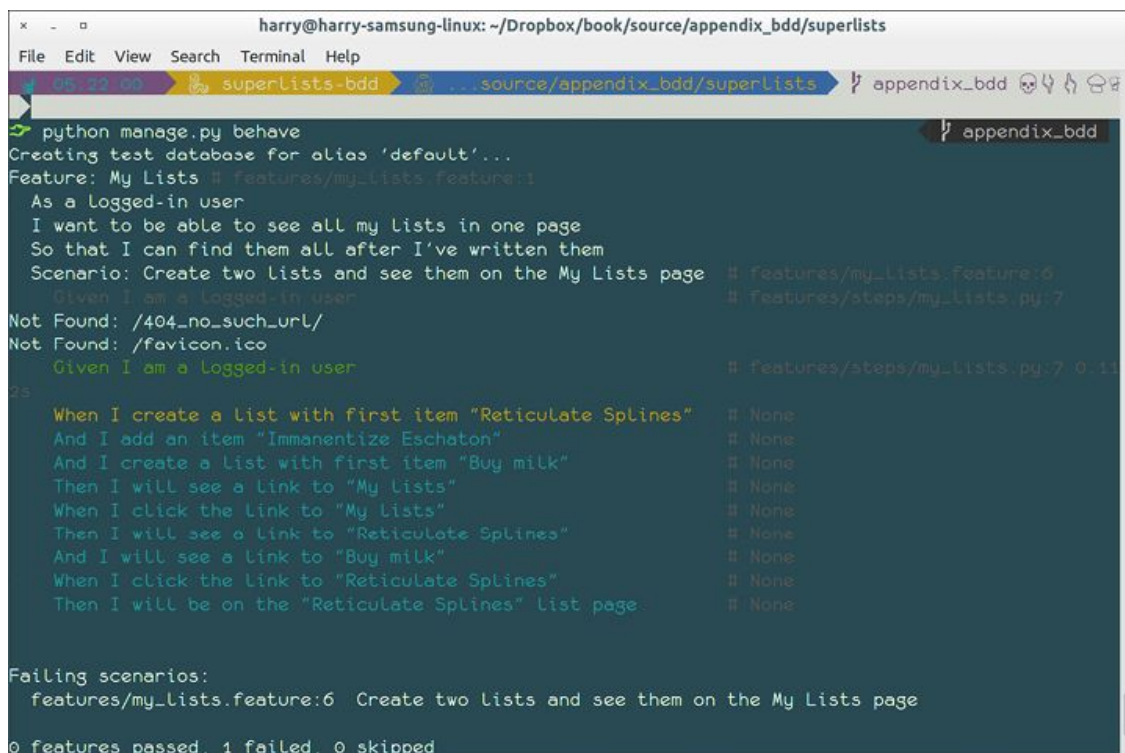
0 features passed, 1 failed, 0 skipped  
0 scenarios passed, 1 failed, 0 skipped  
0 steps passed, 0 failed, 0 skipped, 10 undefined  
Took 0m0.000s

You can implement step definitions for undefined steps with these snippets:

```
@given(u'I am a logged-in user')
def step_impl(context):
 raise NotImplementedError(u'STEP: Given I am a logged-in user')
```

```
@when(u'I create a list with first item "Reticulate Splines"")
def step_impl(context):
 [...]
```

Você perceberá que toda essa saída é elegantemente colorida, como vemos na Figura E.1.



```
harry@harry-samsung-linux: ~/Dropbox/book/source/appendix_bdd/superlists
File Edit View Search Terminal Help
05:22:00 superlists-bdd ...source/appendix_bdd/superLists appendix_bdd
python manage.py behave appendix_bdd
Creating test database for alias 'default'...
Feature: My Lists # Features/my_lists.feature:1
 As a logged-in user
 I want to be able to see all my lists in one page
 So that I can find them all after I've written them
 Scenario: Create two lists and see them on the My Lists page # Features/my_lists.feature:6
 Given I am a logged-in user # Features/steps/my_lists.py:7
 Not Found: /404_no_such_url/
 Not Found: /favicon.ico
 Given I am a logged-in user # Features/steps/my_lists.py:7 0-11
 2s
 When I create a list with first item "Reticulate Splines" # None
 And I add an item "Immanentize Eschaton" # None
 And I create a list with first item "Buy milk" # None
 Then I will see a link to "My lists" # None
 When I click the link to "My Lists" # None
 Then I will see a link to "Reticulate Splines" # None
 And I will see a link to "Buy milk" # None
 When I click the link to "Reticulate Splines" # None
 Then I will be on the "Reticulate Splines" list page # None
Failing scenarios:
 features/my_lists.feature:6 Create two lists and see them on the My Lists page
0 features passed, 1 failed, 0 skipped
```

Figura E.1 – O behave com saída colorida no console.



Ele está nos incentivando a copiar e colar esses trechos de código e usá-los como pontos de partida para construir nossos passos.

## Definição do primeiro passo

Eis uma primeira tentativa de criar um passo para “Given I am a logged-in user”. Comecei roubando o código de `self.create_pre_authenticated_session` de *functional\_tests/test\_my\_lists.py* e adaptando-o levemente (removendo a versão do lado do servidor, por exemplo, embora fosse fácil adicioná-la novamente depois).

### features/steps/my\_lists.py

```
from behave import given, when, then
from functional_tests.management.commands.create_session import \
 create_pre_authenticated_session
from django.conf import settings
```

```
@given('I am a logged-in user')
def given_i_am_logged_in(context):
 session_key =
 create_pre_authenticated_session(email='edith@example.com')
 ## para definir um cookie, precisamos antes acessar o domínio.
 ## as páginas 404 são as que carregam mais rapidamente!
 context.browser.get(context.get_url("/404_no_such_url/"))
 context.browser.add_cookie(dict(
 name=settings.SESSION_COOKIE_NAME,
 value=session_key,
 path='/',
))
```

A variável *context* exige uma pequena explicação – é um tipo de variável global, pois é passada para cada passo executado, e pode ser usada para armazenar informações que precisem ser compartilhadas entre os passos. Nesse caso, estamos supondo que armazenaremos aí um objeto de navegador e o `server_url`. Acabamos usando-a de modo muito semelhante ao que fizemos com `self` quando estávamos escrevendo FTs com `unittest`.

## Equivalentes a setUp e tearDown em environment.py

Os passos podem fazer alterações no estado em context, mas o local para fazer uma configuração preliminar, isto é, o equivalente a setUp, está em um arquivo chamado *environment.py*:

### features/environment.py

```
from selenium import webdriver

def before_all(context):
 context.browser = webdriver.Firefox()

def after_all(context):
 context.browser.quit()

def before_feature(context, feature):
 pass
```

## Outra execução

Como uma verificação de sanidade, podemos fazer outra execução para ver se o novo passo funciona e se realmente podemos iniciar um navegador:

```
$ python manage.py behave
[...]
1 step passed, 0 failed, 0 skipped, 9 undefined
```

Temos o volume de dados usual na saída, porém podemos ver que aparentemente passamos pelo primeiro passo; vamos definir os passos restantes.

## Capturando parâmetros nos passos

Veremos como o Behave permite capturar parâmetros das descrições dos passos. Nosso próximo passo diz o seguinte:

```
features/my_lists.feature
```

When I create a list with first item "Reticulate Splines"

A definição gerada automaticamente para o passo tinha o seguinte aspecto:

## features/steps/my\_lists.py

```
@given('I create a list with first item "Reticulate Splines"')
def step_impl(context):
 raise NotImplementedError(
 u'STEP: When I create a list with first item "Reticulate Splines"'
)
```

Queremos ser capazes de criar listas com primeiros itens arbitrários, portanto seria bom, de algum modo, capturar o que quer que esteja entre as aspas e passar esse dado como argumento para uma função mais genérica. Esse é um requisito comum em BDD, e o Behave tem uma sintaxe interessante para isso, que lembra o novo estilo de sintaxe de formatação de strings de Python:

## features/steps/my\_lists.py (ch35l006)

[...]

```
@when('I create a list with first item "{first_item_text}"')
def create_a_list(context, first_item_text):
 context.browser.get(context.get_url('/'))
 context.browser.find_element_by_id('id_text').send_keys(first_item_text)
 context.browser.find_element_by_id('id_text').send_keys(Keys.ENTER)
 wait_for_list_item(context, first_item_text)
```

Organizado, não é mesmo?



Capturar parâmetros para os passos é um dos recursos mais eficazes da sintaxe de BDD.

Como ocorre geralmente com testes que usam o Selenium, precisaremos de uma espera explícita. Vamos reutilizar o nosso decorador `@wait` de `base.py`:

## features/steps/my\_lists.py (ch35l007)

```
from functional_tests.base import wait
[...]
```

```
@wait
def wait_for_list_item(context, item_text):
 context.test.assertIn(
 item_text,
 context.browser.find_element_by_css_selector('#id_list_table').text
)
```

De modo semelhante, podemos fazer acréscimos em uma lista existente e ver ou clicar nos links:

## features/steps/my\_lists.py (ch35l008)

```
from selenium.webdriver.common.keys import Keys
[...]
```

```
@when('I add an item "{item_text}"')
def add_an_item(context, item_text):
 context.browser.find_element_by_id('id_text').send_keys(item_text)
 context.browser.find_element_by_id('id_text').send_keys(Keys.ENTER)
 wait_for_list_item(context, item_text)
```

```
@then('I will see a link to "{link_text}"')
@wait
def see_a_link(context, link_text):
 context.browser.find_element_by_link_text(link_text)
```

```
@when('I click the link to "{link_text}"')
def click_link(context, link_text):
 context.browser.find_element_by_link_text(link_text).click()
```

Observe que podemos até mesmo utilizar o nosso decorador `@wait` nos próprios passos.

Por fim, vejamos o passo um pouco mais complexo, que diz que estou na página de uma lista em particular:

## features/steps/my\_lists.py (ch35l009)

```
@then('I will be on the "{first_item_text}" list page')
@wait
def on_list_page(context, first_item_text):
 first_row = context.browser.find_element_by_css_selector(
 '#id_list_table tr:first-child'
)
 expected_row_text = '1: ' + first_item_text
 context.test.assertEqual(first_row.text, expected_row_text)
```

Agora podemos executar isso e ver a nossa primeira falha esperada:

```
$ python manage.py behave
```

```
Feature: My Lists # features/my_lists.feature:1
 As a logged-in user
 I want to be able to see all my lists in one page
 So that I can find them all after I've written them
 Scenario: Create two lists and see them on the My Lists page #
features/my_lists.feature:6
 Given I am a logged-in user #
features/steps/my_lists.py:19
 When I create a list with first item "Reticulate Splines" #
features/steps/my_lists.py:31
 And I add an item "Immanentize Eschaton" #
features/steps/my_lists.py:39
 And I create a list with first item "Buy milk" #
features/steps/my_lists.py:31
 Then I will see a link to "My lists" #
functional_tests/base.py:12
 Traceback (most recent call last):
[...]
 File "features/steps/my_lists.py", line 49, in see_a_link
 context.browser.find_element_by_link_text(link_text)
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable
to
locate element: My lists

[...]
```

Failing scenarios:

features/my\_lists.feature:6 Create two lists and see them on the My Lists page

0 features passed, 1 failed, 0 skipped

0 scenarios passed, 1 failed, 0 skipped

4 steps passed, 1 failed, 5 skipped, 0 undefined

Podemos ver como a saída realmente nos dá uma noção do ponto a que chegamos na “história” do teste: conseguimos criar nossas duas listas com sucesso, mas o link “My lists” não aparece.

## Comparação com o FT em estilo inline

Não descreverei a implementação da funcionalidade, mas você pode ver como o teste orientará o desenvolvimento, assim como o FT em estilo inline teria feito.

Vamos observá-la para fazer uma comparação:

### functional\_tests/test\_my\_lists.py

```
def test_logged_in_users_lists_are_saved_as_my_lists(self):
 # Edith é uma usuária logada
 self.create_pre_authenticated_session('edith@example.com')

 # Edith acessa a página inicial e começa uma lista
 self.browser.get(self.live_server_url)
 self.add_list_item('Reticulate splines')
 self.add_list_item('Immanentize eschaton')
 first_list_url = self.browser.current_url

 # Ela percebe o link para "My lists" pela primeira vez.
 self.browser.find_element_by_link_text('My lists').click()

 # Ela vê que sua lista está lá, nomeada de acordo com
 # o primeiro item da lista
 self.wait_for(
 lambda: self.browser.find_element_by_link_text('Reticulate splines')
)
 self.browser.find_element_by_link_text('Reticulate splines').click()
```

```

self.wait_for(
 lambda: self.assertEqual(self.browser.current_url, first_list_url)
)

Ela decide iniciar outra lista, somente para conferir
self.browser.get(self.live_server_url)
self.add_list_item('Click cows')
second_list_url = self.browser.current_url

Em "my lists", sua nova lista aparece
self.browser.find_element_by_link_text('My lists').click()
self.wait_for(
 lambda: self.browser.find_element_by_link_text('Click cows')
)
self.browser.find_element_by_link_text('Click cows').click()
self.wait_for(
 lambda: self.assertEqual(self.browser.current_url, second_list_url)
)

Ela faz logout. A opção "My lists" desaparece
self.browser.find_element_by_link_text('Log out').click()
self.wait_for(lambda: self.assertEqual(
 self.browser.find_elements_by_link_text('My lists'),
 []
))

```

Não é totalmente uma comparação de maçãs com maçãs, mas podemos ver o número de linhas de código na Tabela E.1.

*Tabela E.1 – Comparação entre o número de linhas de código*

BDD	FT padrão
Arquivo de feature: 20 (3 opcionais)	Corpo da função de teste: 45
Arquivo de passos : 56 linhas	Funções auxiliares: 23

A comparação não é perfeita, mas poderíamos dizer que o arquivo de feature e o corpo de uma função de “FT padrão” são equivalentes, pois apresentam a “história” principal de um teste, enquanto os passos e as funções auxiliares representam os detalhes de implementação “ocultos”. Se somá-los, os números totais são bem semelhantes, mas note que eles estão distribuídos

de modo diferente: os testes de BDD deixaram a história mais concisa e passaram mais tarefas para os detalhes de implementação ocultos.

## **BDD incentiva código de teste estruturado**

Para mim, este é o verdadeiro atrativo: a ferramenta de BDD nos *forçou* a estruturar o nosso código de teste. No FT em estilo inline, somos livres para usar quantas linhas quisermos para implementar um passo, conforme descrito em sua linha de comentário. É muito difícil resistir ao impulso de simplesmente copiar e colar um código de outro lugar, ou um código anterior de teste. Podemos ver que, a essa altura do livro, eu implementei apenas poucas funções auxiliares (como `get_item_input_box`).

Em contrapartida, a sintaxe de BDD imediatamente me forçou a ter uma função separada para cada passo, de modo que já implementei um pouco de código altamente reutilizável para:

- iniciar uma nova lista;
- adicionar um item em uma lista existente;
- clicar em um link com um texto em particular;
- garantir que estou olhando para a página de uma lista em particular.

O BDD de fato incentiva você a escrever um código de teste que pareça corresponder bem ao domínio do negócio e a usar uma camada de abstração entre a história de seu FT e a sua implementação no código.

Para expressar isso de forma definitiva, podemos dizer que, teoricamente, se você quisesse mudar as linguagens de programação, seria possível manter todas as suas features na sintaxe Gherkin, exatamente como estão, jogar fora os passos em Python e substituí-los por passos implementados em outra linguagem.



## Padrão Page como uma alternativa

No Capítulo 25 do livro, apresentei um exemplo do “padrão Page”, que é uma abordagem orientada a objetos para estruturar seus testes com o Selenium. Eis um lembrete de como era a sua aparência:

### functional\_tests/test\_sharing.py

```
from .my_lists_page import MyListsPage
[...]
```

```
class SharingTest(FunctionalTest):

 def test_can_share_a_list_with_another_user(self):
 # [...]
 self.browser.get(self.live_server_url)
 list_page = ListPage(self).add_list_item('Get help')

 # Ela percebe que há uma opção "Share this list" (Compartilhar essa lista)
 share_box = list_page.get_share_box()
 self.assertEqual(
 share_box.get_attribute('placeholder'),
 'your-friend@example.com'
)

 # Ela compartilha sua lista.
 # A página é atualizada para informar que a lista foi compartilhada
 # com Oniciferous:
 list_page.share_list_with('oniciferous@example.com')
```

Eis a aparência da classe Page:

### functional\_tests/lists\_pages.py

```
class ListPage(object):

 def __init__(self, test):
 self.test = test

 def get_table_rows(self):
```

```
return self.test.browser.find_elements_by_css_selector('#id_list_table tr')
```

@wait

```
def wait_for_row_in_list_table(self, item_text, item_number):
 row_text = '{}: {}'.format(item_number, item_text)
 rows = self.get_table_rows()
 self.test.assertIn(row_text, [row.text for row in rows])
```

```
def get_item_input_box(self):
 return self.test.browser.find_element_by_id('id_text')
```

Portanto, definitivamente, é possível implementar uma camada de abstração semelhante e uma espécie de DSL nos FTs em estilo inline, seja usando o padrão Page ou qualquer estrutura que você preferir – mas é uma questão de autodisciplina, e não de ter um framework que empurre você nessa direção.



Com efeito, você pode, na verdade, usar o padrão Page também com o BDD, como um recurso a ser utilizado pelos seus passos ao navegar pelas páginas de seu site.

## O BDD pode ser menos expressivo que os comentários inline

Por outro lado, também posso ver o potencial da sintaxe Gherkin de parecer, de certo modo, mais restritiva. Compare o quão expressivos e legíveis são os comentários em estilo inline, em relação ao recurso um pouco mais inconveniente do BDD:

### functional\_tests/test\_my\_lists.py

```
Edith é uma usuária logada
Edith acessa a página inicial e começa uma lista
Ela percebe o link para "My lists" pela primeira vez.
Ela vê que sua lista está lá, nomeada de acordo com
o primeiro item da lista
Ela decide iniciar outra lista, somente para conferir
```

```
Em "my lists", sua nova lista aparece
Ela faz logout. A opção "My lists" desaparece
[...]
```

Isso é muito mais legível e natural do que os nossos encantamentos levemente forçados com Given/Then/When e, de certo modo, pode incentivar um raciocínio mais centrado no usuário. (Há uma sintaxe no Gherkin para incluir “comentários” em um arquivo de feature que, até certo ponto, atenuaria esse problema, mas entendo que ele não é amplamente usado.)

## **Pessoas que não são programadoras escreverão testes?**

Não discuti uma das promessas originais do BDD, que é o fato de pessoas que não são programadoras – talvez representantes do negócio ou do cliente – poderem, na verdade, escrever na sintaxe Gherkin. Sou bem cético sobre o fato de isso realmente funcionar no mundo real, mas não acho que diminua as outras vantagens em potencial do BDD.

## **Algumas tentativas de conclusões**

Mergulhei apenas a ponta dos dedos dos pés no mundo do BDD, portanto hesito em tirar qualquer conclusão sólida. Acho, no entanto, que a estruturação “forçada” dos FTs em passos é muito atraente – parece que ela tem o potencial para incentivar muita reutilização em seu código de FT e separa, de forma organizada, as responsabilidades entre descrever a história e implementá-la; ela nos força a pensar na situação em termos do domínio do negócio, em vez de pensar “no que precisamos fazer com o Selenium”.

Contudo, nada vem de graça. A sintaxe Gherkin é restritiva, quando comparada com a total liberdade oferecida pelos comentários de FT inline.

Também gostaria de ver como o BDD escala, depois que você tiver não só uma ou duas features, e quatro ou cinco passos, mas várias

dezenas de features e centenas de linhas de código para os passos. De modo geral, eu diria que, sem dúvida, vale a pena investigar, e provavelmente eu usarei o BDD em meu próximo projeto pessoal. Agradeço a Daniel Pope, Rachel Willmer e Jared Contrascere por seu feedback para este capítulo.

## **Conclusões sobre o BDD**

***Incentiva um código de teste estruturado e reutilizável.***

Ao separar as responsabilidades, dividir seus FTs entre o arquivo de “feature” (funcionalidade) com a sintaxe Gherkin, legível aos seres humanos, e uma implementação separada das funções de passos (steps), o BDD tem o potencial para incentivar um código de teste mais reutilizável e administrável.

### *Pode vir à custa da legibilidade*

A sintaxe Gherkin, apesar de toda a sua tentativa de ser legível aos seres humanos, em última instância, é uma restrição à linguagem humana, portanto talvez não capture as nuances e intenções tão bem quanto os comentários inline o fazem.

### *Experimente usá-lo! Eu vou*

Como continuo dizendo, não usei o BDD em um projeto de verdade, portanto você deve considerar minhas palavras com uma boa dose de cautela, mas eu gostaria de endossá-lo calorosamente. Experimentarei usar o BDD no próximo projeto que puder, e incentivaria você a fazer o mesmo também.





## APÊNDICE F

# Construindo uma API REST: JSON, Ajax e simulação com JavaScript

REST (Representational State Transfer, ou Transferência de Estado Representativo) é uma abordagem para design de um web service que permite a um usuário obter e atualizar informação sobre “recursos”. Tornou-se a abordagem predominante no design de APIs para uso na web.

Construímos uma aplicação web funcional sem a necessidade de uma API até agora. Por que iríamos querer uma? Um motivo poderia ser melhorar a experiência do usuário deixando o site mais dinâmico. Em vez de esperar que a página se atualize após cada acréscimo em uma lista, podemos usar JavaScript para disparar essas requisições assincronamente para a nossa API e dar uma impressão mais interativa ao usuário.

Depois da construção da API, o mais interessante talvez seja a possibilidade de interagir com a nossa aplicação backend por meio de outros mecanismos além do navegador. Um aplicativo móvel pode ser um novo candidato como uma aplicação cliente, outro poderia ser algum tipo de aplicação de linha de comando, ou outros desenvolvedores poderiam construir bibliotecas e ferramentas em torno de seu backend.

Neste capítulo, veremos como construir uma API “manualmente”. No próximo, apresentarei uma visão geral de como usar uma ferramenta popular do ecossistema do Django, chamada Django-

Rest-Framework.

## Nossa abordagem neste apêndice

Não converterei totalmente a aplicação por enquanto; começaremos supondo que temos uma lista existente. O REST define um relacionamento entre os URLs e os métodos HTTP (GET e POST, mas também os mais incomuns, como PUT e DELETE), o que nos orientará em nosso design.

A entrada na Wikipédia sobre REST (<http://bit.ly/2u6qeYw>) apresenta uma boa visão geral. Em suma:

- nossa nova estrutura de URL será `/api/lists/{id}/`;
- GET dará os detalhes de uma lista (incluindo todos os seus itens) em formato JSON;
- POST permitirá adicionar um item.

Partiremos do código no estado em que estava no final do Capítulo 25.

## Escolhendo a nossa abordagem de teste

Se estivéssemos construindo uma API totalmente agnóstica sobre seus clientes, poderíamos pensar nos níveis em que a testaríamos. O equivalente aos testes funcionais poderia ser iniciar um servidor real (talvez usando `LiveServerTestCase`) e interagir com ele usando a biblioteca `requests`. Teríamos que pensar cuidadosamente sobre como configurar as fixtures (se usarmos a própria API, isso introduzirá muitas dependências entre os testes) e quais camadas adicionais de testes de baixo nível/de unidade seriam mais eficazes para nós. Ou poderíamos decidir que uma única camada de testes usando o Django Test Client seria suficiente.

Em nosso caso, estamos construindo uma API no contexto de um lado cliente baseado em navegador. Queremos começar usando-a em nosso site de produção e fazer a aplicação continuar oferecendo

a mesma funcionalidade proporcionada antes. Desse modo, nossos testes funcionais continuarão a desempenhar o papel de testes de mais alto nível e a verificar a integração entre o nosso JavaScript e a API.

Isso deixa o Django Test Client como um lugar natural para deixar nossos testes de mais baixo nível. Vamos começar por aí.

## Estrutura básica

Começamos com um teste de unidade que simplesmente verifica se a nossa nova estrutura de URL devolve uma resposta 200 a requisições GET, e se o formato JSON está sendo usado (em vez de HTML):

### lists/tests/test\_api.py

```
import json
from django.test import TestCase

from lists.models import List, Item

class ListAPITest(TestCase):
 base_url = '/api/lists/{}' ❶

 def test_get_returns_json_200(self):
 list_ = List.objects.create()
 response = self.client.get(self.base_url.format(list_.id))
 self.assertEqual(response.status_code, 200)
 self.assertEqual(response['content-type'], 'application/json')
```

- ❶ Usar uma constante no nível de classe para o URL em teste é um novo padrão que introduziremos neste apêndice. Isso nos ajudará a remover duplicações de URLs fixos no código. Você poderia até mesmo usar uma chamada a `reverse` para reduzir mais ainda a duplicação.

Inicialmente vamos preparar alguns arquivos de *urls*:

## superlists/urls.py

```
from django.conf.urls import include, url
from accounts import urls as accounts_urls
from lists import views as list_views
from lists import api_urls
from lists import urls as list_urls

urlpatterns = [
 url(r'^$', list_views.home_page, name='home'),
 url(r'^lists/', include(list_urls)),
 url(r'^accounts/', include(accounts_urls)),
 url(r'^api/', include(api_urls)),
]
```

e:

## lists/api\_urls.py

```
from django.conf.urls import url
from lists import api

urlpatterns = [
 url(r'^lists/(\d+)/$', api.list, name='api_list'),
]
```

O núcleo propriamente dito de nossa API pode ficar em um arquivo chamado *api.py*. Apenas três linhas devem ser suficientes:

## lists/api.py

```
from django.http import HttpResponse

def list(request, list_id):
 return HttpResponse(content_type='application/json')
```

Os testes devem passar, e temos a estrutura básica pronta:

```
$ python manage.py test lists
```

```
[...]
```

```

Ran 50 tests in 0.177s
```

```
OK
```

## Respondendo realmente com algum dado

Nosso próximo passo é fazer a nossa API de fato responder com algum conteúdo – especificamente, com uma representação JSON dos itens de nossa lista:

### lists/tests/test\_api.py (ch36l002)

```
def test_get_returns_items_for_correct_list(self):
 other_list = List.objects.create()
 Item.objects.create(list=other_list, text='item 1')
 our_list = List.objects.create()
 item1 = Item.objects.create(list=our_list, text='item 1')
 item2 = Item.objects.create(list=our_list, text='item 2')
 response = self.client.get(self.base_url.format(our_list.id))
 self.assertEqual(
 json.loads(response.content.decode('utf8')), ❶
 [
 {'id': item1.id, 'text': item1.text},
 {'id': item2.id, 'text': item2.text},
]
)
```

❶ Esse é o ponto principal a ser observado nesse teste. Esperamos que nossa resposta esteja em formato JSON; usamos `json.loads()` porque testar objetos Python é mais fácil do que ficar lidando com strings JSON puras.

De modo recíproco, a implementação utiliza `json.dumps()`:

### lists/api.py

```
import json
from django.http import HttpResponse
from lists.models import List, Item
```

```
def list(request, list_id):
 list_ = List.objects.get(id=list_id)
 item_dicts = [
 {'id': item.id, 'text': item.text}
 for item in list_.item_set.all()
]
```

```
]
return HttpResponse(
 json.dumps(item_dicts),
 content_type='application/json'
)
```

É uma boa oportunidade para usar uma compreensão de lista (list comprehension)!

## Adicionando o POST

O segundo recurso de que precisamos em nossa API é a capacidade de adicionar novos itens em nossa lista usando uma requisição POST. Começaremos com o “caminho feliz”:

### lists/tests/test\_api.py (ch36l004)

```
def test_POSTing_a_new_item(self):
 list_ = List.objects.create()
 response = self.client.post(
 self.base_url.format(list_.id),
 {'text': 'new item'},
)
 self.assertEqual(response.status_code, 201)
 new_item = list_.item_set.get()
 self.assertEqual(new_item.text, 'new item')
```

A implementação é igualmente simples – basicamente é o mesmo que fazemos em nossa view usual, mas devolvemos um 201 em vez de um redirecionamento:

### lists/api.py (ch36l005)

```
def list(request, list_id):
 list_ = List.objects.get(id=list_id)
 if request.method == 'POST':
 Item.objects.create(list=list_, text=request.POST['text'])
 return HttpResponse(status=201)
 item_dicts = [
 ...]
```

Com isso, nosso trabalho está iniciado:

```
$ python manage.py test lists
```

```
[...]
```

```
Ran 52 tests in 0.177s
```

```
OK
```



Um dos pontos divertidos sobre construir uma API REST é que você passa a usar um pouco mais do intervalo completo de códigos de status HTTP ([https://en.wikipedia.org/wiki/List\\_of\\_HTTP\\_status\\_codes](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes)).

## Testando o Ajax do lado cliente com Sinon.js

Nem *pense* em fazer testes de Ajax sem uma biblioteca de simulação. Frameworks de testes e ferramentas diferentes têm a sua própria biblioteca; o *Sinon* é genérico. Ele também oferece mocks JavaScript, como veremos...

Comece fazendo o download do Sinon em <http://sinonjs.org/> e colocando-o em nossa pasta `lists/static/tests/`.

Então podemos escrever o nosso primeiro teste de Ajax:

### lists/static/tests/tests.html (ch36l007)

```
<div id="qunit-fixture">
 <form>
 <input name="text" />
 <div class="has-error">Error text</div>
 </form>
 <table id="id_list_table"> ❶
 </table>
</div>
<script src="../../jquery-3.1.1.min.js"></script>
<script src="../../list.js"></script>
<script src="qunit-2.0.1.js"></script>
<script src="sinon-1.17.6.js"></script> ❷

<script>
/* sinon global */
```

```

var server;
QUnit.testStart(function () {
 server = sinon.fakeServer.create(); ❸
});
QUnit.testDone(function () {
 server.restore(); ❹
});

QUnit.test("errors should be hidden on keypress", function (assert) {
[...]

QUnit.test("should get items by ajax on initialize", function (assert) {
 var url = '/getitems/';
 window.Superlists.initialize(url);

 assert.equal(server.requests.length, 1); ❺
 var request = server.requests[0];
 assert.equal(request.url, url);
 assert.equal(request.method, 'GET');
});

</script>

```

- ❶ Adicionamos um novo item na div de fixture para representar a nossa tabela com a lista.
- ❷ Importamos *sinon.js* (você deverá fazer o seu download e colocá-lo na pasta correta).
- ❸ `testStart` e `testDone` são os equivalentes da QUnit a `setUp` e `tearDown`. São usados para dizer ao Sinon que inicie suas ferramenta de testes de Ajax, o `fakeServer`, e deixá-lo disponível por meio de uma variável de escopo global chamada `server`.
- ❹ Isso nos permite fazer asserções sobre qualquer requisição Ajax que tenha sido feita pelo nosso código. Nesse caso, testamos para qual URL a requisição foi enviada e qual foi o método HTTP usado.

Para realmente fazer a nossa requisição Ajax, usaremos os



auxiliares de Ajax da jQuery (<https://api.jquery.com/jquery.get/>), que é  *muito*  mais fácil do que tentar usar os objetos padrões XMLHttpRequest de baixo nível do navegador.

## lists/static/list.js

```
@@ -1,6 +1,10 @@
window.Superlists = {};
-window.Superlists.initialize = function () {
+window.Superlists.initialize = function (url) {
 $('input[name="text"]').on('keypress', function () {
 $('.has-error').hide();
 });
+
+ $.get(url);
+
};
+
```

Com isso, nossos testes devem passar:

- 5 assertions of 5 passed, 0 failed.
- 1. errors should be hidden on keypress (1)
- 2. errors aren't hidden if there is no keypress (1)
- 3. should get items by ajax on initialize (3)

Bem, podemos enviar uma requisição GET ao servidor, mas o que dizer sobre realmente *fazer* algo? Como podemos testar a parte “assíncrona”, em que lidamos com a (futura) resposta?

## Sinon e os testes da parte assíncrona do Ajax

Esse é um motivo importante para amar o Sinon. `server.respond()` nos permite controlar exatamente o fluxo do código assíncrono.

## lists/static/tests/tests.html (ch36I009)

```
QUnit.test("should fill in lists table from ajax response", function (assert) {
 var url = '/getitems/';
 var responseData = [
 {id: 101, 'text': 'item 1 text'},
 {id: 102, 'text': 'item 2 text'},
];
});
```

```
server.respondWith('GET', url, [
 200, {"Content-Type": "application/json"}, JSON.stringify(responseData) ❶
]);
window.Superlists.initialize(url); ❷
```

```
server.respond(); ❸
```

```
var rows = $('#id_list_table tr'); ❹
assert.equal(rows.length, 2);
var row1 = $('#id_list_table tr:first-child td');
assert.equal(row1.text(), '1: item 1 text');
var row2 = $('#id_list_table tr:last-child td');
assert.equal(row2.text(), '2: item 2 text');
});
```

- ❶ Configuramos alguns dados de resposta para o Sinon usar, dizendo-lhe qual é o código de status, os cabeçalhos e, mais importante ainda, qual é o tipo de resposta JSON que queremos simular, proveniente do servidor.
- ❷ Então chamamos a função em teste.
- ❸ Aqui está a mágica. *Então* chamamos `server.respond()`, quando quisermos; isso disparará toda a parte assíncrona do laço Ajax – isto é, qualquer callback que tenhamos designado para lidar com a resposta.
- ❹ Agora podemos verificar discretamente se a nossa callback Ajax de fato preencheu nossa tabela com as novas linhas da lista...

A implementação pode ter o seguinte aspecto:

## lists/static/list.js (ch36|010)

```
if (url) {
 $.get(url).done(function (response) { ❶
 var rows = "";
 for (var i=0; i<response.length; i++) { ❷
 var item = response[i];
 rows += '\n<tr><td>' + (i+1) + ': ' + item.text + '</td></tr>';
 }
 $('#id_list_table').html(rows);
 });
}
```

```
}
```



Temos sorte por causa do modo como a jQuery registra suas callbacks para o Ajax quando usamos a função `.done()`. Se quiser passar para a callback `.then()` padrão de JavaScript Promise, teremos mais um “nível” de execução assíncrona. A QUnit tem uma forma de lidar com isso. Consulte a documentação para a função de execução assíncrona (<http://api.qunitjs.com/async/>). Outros frameworks de teste têm recursos semelhantes.

## Associando tudo no template para ver se realmente funciona

Vamos inicialmente provocar uma falha removendo o laço na tabela de lista `{% for %}` do template `lists.html`:

### lists/templates/list.html

```
@@ -6,9 +6,6 @@
```

```
{% block table %}
 <table id="id_list_table" class="table">
- {% for item in list.item_set.all %}
- <tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>
- {% endfor %}
 </table>

 {% if list.owner %}
```



Isso fará um dos testes de unidade falhar. Não há problemas em apagar esse teste neste ponto.

## Degradação elegante e melhorias progressivas

Ao remover a versão não Ajax da página de lista, eliminei a opção de depreciação elegante

([https://www.w3.org/wiki/Graceful\\_degradation\\_versus\\_progressive\\_enhancement](https://www.w3.org/wiki/Graceful_degradation_versus_progressive_enhancement))- – isto é, manter uma versão do site que continuará funcionando sem

JavaScript.

Isso costumava ser um problema para a acessibilidade: navegadores que “liam a tela” para pessoas com deficiência visual costumavam não ter JavaScript, portanto depender totalmente dele excluiria esses usuários. Esse não é mais um grande problema, conforme o meu entendimento. Alguns usuários, porém, bloquearão o JavaScript por questões de segurança.

Outro problema comum são os diferentes níveis de suporte a JavaScript em navegadores distintos. Será um problema em particular se você começar a se aventurar em direção ao desenvolvimento de frontends “modernos” e ao ES2015.

Em suma, é sempre bom ter um “backup” sem JavaScript. Particularmente, se você construir um site que funcione bem sem ele, não jogue fora prematuramente a sua “boa e velha” versão HTML funcional. Só estou fazendo isso porque é conveniente para o que quero demonstrar.

Isso faz o nosso FT básico falhar:

```
$ python manage.py test functional_tests.test_simple_list_creation
[...]
FAIL: test_can_start_a_list_for_one_user
[...]
File ".../superlists/functional_tests/test_simple_list_creation.py", line
32, in test_can_start_a_list_for_one_user
 self.wait_for_row_in_list_table('1: Buy peacock feathers')
[...]
AssertionError: '1: Buy peacock feathers' not found in []
[...]
FAIL: test_multiple_users_can_start_lists_at_different_urls

FAILED (failures=2)
```

Vamos adicionar um bloco chamado `{% scripts %}` no template-base, que poderá ser seletivamente sobrescrito depois em nossa página de listas:

### lists/templates/base.html

```
<script src="/static/list.js"></script>
```

```
{% block scripts %}
```

```
<script>
$(document).ready(function () {
 window.Superlists.initialize();
});
</script>
{% endblock scripts %}

</body>
```

Agora, em *list.html*, adicionamos uma chamada um pouco diferente para initialize, com o URL correto:

## lists/templates/list.html (ch36l016)

```
{% block scripts %}
<script>
$(document).ready(function () {
 var url = "{% url 'api_list' list.id %}";
 window.Superlists.initialize(url);
});
</script>
{% endblock scripts %}
```

E adivinhe só? O teste passa!

```
$ python manage.py test functional_tests.test_simple_list_creation
[...]
Ran 2 test in 11.730s
```

OK

É um ótimo começo!

Se você executar todos os FTs agora, verá que teremos algumas falhas em outros FTs, portanto precisaremos lidar com elas. Além do mais, estamos usando um POST antigo do formulário, com atualização de página, desse modo ainda não temos nosso aplicativo single-page (de página única) super da moda. Mas chegaremos lá!

## Implementando POST com Ajax, incluindo o token de CSRF

Inicialmente daremos ao nosso formulário de lista um id para que possamos identificá-lo facilmente em nosso JavaScript:

## lists/templates/base.html

```
<h1>{% block header_text %}{% endblock %}</h1>
{% block list_form %}
 <form id="id_item_form" method="POST"
 action="{% block form_action %}{% endblock %}">
 {{ form.text }}
 [...]
```

Em seguida, ajuste a fixture em nosso teste de JavaScript para que reflita esse ID, bem como o token de CSRF que atualmente está na página:

## lists/static/tests/tests.html

```
@@ -9,9 +9,14 @@
<body>
 <div id="qunit"></div>
 <div id="qunit-fixture">
- <form>
+ <form id="id_item_form">
 <input name="text" />
- <div class="has-error">Error text</div>
+ <input type="hidden" name="csrfmiddlewaretoken" value="tokey" />
+ <div class="has-error">
+ <div class="help-block">
+ Error text
+ </div>
+ </div>
 </form>
```

Eis o nosso teste:

## lists/static/tests/tests.html (ch36l019)

```
QUnit.test("should intercept form submit and do ajax post", function (assert) {
 var url = '/listitemsapi/';
 window.Superlists.initialize(url);

 $('#id_item_form input[name="text"]').val('user input'); ❶
```

```
$('#id_item_form input[name="csrfmiddlewaretoken"]').val('tokeney'); ❶
$('#id_item_form').submit(); ❶
```

```
assert.equal(server.requests.length, 2); ❷
var request = server.requests[1];
assert.equal(request.url, url);
assert.equal(request.method, "POST");
assert.equal(
 request.requestBody,
 'text=user+input&csrfmiddlewaretoken=tokeney' ❸
);
});
```

- ❶ Simularemos o usuário preenchendo o formulário e pressionando Submit.
- ❷ Esperamos que agora deva haver uma segunda requisição Ajax (a primeira é o GET para a tabela com itens da lista).
- ❸ Verificamos o requestBody de nosso POST. Como podemos ver, ele está codificado com URL, o que não é o valor mais fácil para ser testado, mas ainda é razoavelmente legível.

Eis o modo como fizemos a implementação:

## lists/static/list.js

```
[...]
 $('#id_list_table').html(rows);
});

var form = $('#id_item_form');
form.on('submit', function(event) {
 event.preventDefault();
 $.post(url, {
 'text': form.find('input[name="text"]').val(),
 'csrfmiddlewaretoken': form.find('input[name="csrfmiddlewaretoken"]').val(),
 });
});
```

Isso faz com que nossos testes de JavaScript passem, mas causa falha em nossos FTs, pois, embora estejamos fazendo corretamente o nosso POST, não estamos atualizando a página depois dele para

mostrar o novo item da lista:

```
$ python manage.py test functional_tests.test_simple_list_creation
[...]
AssertionError: '2: Use peacock feathers to make a fly' not found in ['1: Buy
peacock feathers']
```

## Simulação em JavaScript

Queremos que o nosso lado cliente atualize a tabela de itens depois que o POST Ajax for concluído. Essencialmente ele fará o mesmo trabalho que nós assim que a página carregar, obtendo a lista atual de itens do servidor e preenchendo a tabela de itens.

Parece que uma função auxiliar se faz necessária!

### lists/static/list.js

```
window.Superlists = {};

window.Superlists.updateItems = function (url) {
 $.get(url).done(function (response) {
 var rows = "";
 for (var i=0; i<response.length; i++) {
 var item = response[i];
 rows += "\n<tr><td> + (i+1) + ': ' + item.text + '</td></tr>';
 }
 $('#id_list_table').html(rows);
 });
};

window.Superlists.initialize = function (url) {
 $('input[name="text"]').on('keypress', function () {
 $('.has-error').hide();
 });

 if (url) {
 window.Superlists.updateItems(url);

 var form = $('#id_item_form');
 [...]
```



Essa foi apenas uma refatoração; verificamos agora se todos os testes de JavaScript continuam passando:

12 assertions of 12 passed, 0 failed.

1. errors should be hidden on keypress (1)
2. errors aren't hidden if there is no keypress (1)
3. should get items by ajax on initialize (3)
4. should fill in lists table from ajax response (3)
5. should intercept form submit and do ajax post (4)

Como podemos testar se o nosso POST Ajax chama `updateItems` quando o POST é bem-sucedido? Não queremos duplicar estupidamente o código que simula uma resposta do servidor e verifica a tabela de itens de modo manual... Que tal um mock?

Inicialmente configuramos um recurso chamado “sandbox”. Ele manterá o controle de todos os mocks que criarmos e garantirá que removerá o monkeypatch de tudo que tiver sido simulado, após cada teste:

## lists/static/tests/tests.html (ch36I023)

```
var server, sandbox;
QUnit.testStart(function () {
 server = sinon.fakeServer.create();
 sandbox = sinon.sandbox.create();
});
QUnit.testDone(function () {
 server.restore();
 sandbox.restore(); ❶
});
```

- ❶ Esse `.restore()` é a parte importante; ele desfaz todas as simulações que realizamos em cada teste.

## lists/static/tests/tests.html (ch36I024)

```
QUnit.test("should call updateItems after successful post", function (assert) {
 var url = '/listitemsapi/';
 window.Superlists.initialize(url); ❶
 var response = [
 201,
 {"Content-Type": "application/json"},
```

```

 JSON.stringify({}),
];
 server.respondWith('POST', url, response); ❶
 $('#id_item_form input[name="text"]').val('user input');
 $('#id_item_form input[name="csrfmiddlewaretoken"]').val('tokeney');
 $('#id_item_form').submit();

 sandbox.spy(window.Superlists, 'updateItems'); ❷
 server.respond(); ❸

 assert.equal(
 window.Superlists.updateItems.lastCall.args, ❹
 url
);
});

```

- ❶ Primeiro ponto a ser observado: só configuramos a resposta de nosso servidor *depois* de termos feito a inicialização. Queremos que essa seja a resposta à requisição POST que ocorre na submissão do formulário, e não a resposta à requisição GET inicial. (Você se lembra de nossa lição no Capítulo 16? Um dos aspectos mais desafiadores nos testes de JavaScript é controlar a ordem da execução.)
- ❷ De modo semelhante, só começamos a simular a nossa função auxiliar *depois* que soubermos que a primeira chamada para o GET inicial já ocorreu. A chamada a `sandbox.spy` é responsável pelo trabalho feito por `patch` nos testes de Python. Ela substitui o dado objeto por uma versão simulada.
- ❸ Nossa função `updateItems` agora adquiriu alguns atributos extras de simulação, como `lastCall` e `lastCall.args`, que são como o `call_args` dos mocks de Python.

Para fazer os testes passarem, inicialmente causamos um erro proposital a fim de verificar se eles de fato verificam o que achamos que devem verificar:

## lists/static/list.js

```
$.post(url, {
```

```
'text': form.find('input[name="text"]').val(),
'csrfmiddlewaretoken': form.find('input[name="csrfmiddlewaretoken"]').val(),
}).done(function () {
 window.Superlists.updateItems();
});
```

Sim, estamos quase lá, mas não exatamente:

12 assertions of 13 passed, 1 failed.

[...]

6. should call updateItems after successful post (1, 0, 1)

1. failed

Expected: "/listitemsapi/"

Result: []

Diff: "/listitemsapi/"[]

Source: file:///.../superlists/lists/static/tests/tests.html:124:15

Fazemos a correção assim:

## lists/static/list.js

```
}).done(function () {
 window.Superlists.updateItems(url);
});
```

E o nosso FT passa! Ou, pelo menos, um deles. Os outros têm problemas, e retornaremos a eles em breve.

## Terminando a refatoração: fazendo os testes corresponderem ao código

Em primeiro lugar, não estarei satisfeito enquanto não terminar essa refatoração e fazer nossos testes de unidade terem uma correspondência um pouco melhor com o código:

## lists/static/tests/tests.html

```
@@ -50,9 +50,19 @@ QUnit.testDone(function () {
});
```

```
-QUnit.test("should get items by ajax on initialize", function (assert) {
+QUnit.test("should call updateItems on initialize", function (assert) {
 var url = '/getitems/';
```

```

+ sandbox.spy(window.Superlists, 'updateItems');
 window.Superlists.initialize(url);
+ assert.equal(
+ window.Superlists.updateItems.lastCall.args,
+ url
+);
+ });
+
+QUnit.test("updateItems should get correct url by ajax", function (assert) {
+ var url = '/getitems/';
+ window.Superlists.updateItems(url);

 assert.equal(server.requests.length, 1);
 var request = server.requests[0];
@@ -60,7 +70,7 @@ QUnit.test("should get items by ajax on initialize", function
(assert) {
 assert.equal(request.method, 'GET');
});

-QUnit.test("should fill in lists table from ajax response", function (assert) {
+QUnit.test("updateItems should fill in lists table from ajax response", function
(assert) {
 var url = '/getitems/';
 var responseData = [
 {'id': 101, 'text': 'item 1 text'},
@@ -69,7 +79,7 @@ QUnit.test("should fill in lists table from ajax response",
function [...]
 server.respondWith('GET', url, [
 200, {"Content-Type": "application/json"}, JSON.stringify(responseData)
]);
- window.Superlists.initialize(url);
+ window.Superlists.updateItems(url);

 server.respond();

```

Com isso, devemos ter uma execução de testes com a aparência a seguir:

- 14 assertions of 14 passed, 0 failed.
1. errors should be hidden on keypress (1)
  2. errors aren't hidden if there is no keypress (1)
  3. should call updateItems on initialize (1)

4. updateItems should get correct url by ajax (3)
5. updateItems should fill in lists table from ajax response (3)
6. should intercept form submit and do ajax post (4)
7. should call updateItems after successful post (1)

## Validação de dados: um exercício para o leitor?

Se você fizer uma execução completa dos testes, deverá ver que dois dos FTs de validação estão com falha:

```
$ python manage.py test
[...]
ERROR: test_cannot_add_duplicate_items
(functional_tests.test_list_item_validation.ItemValidationTest)
[...]
ERROR: test_error_messages_are_cleared_on_input
(functional_tests.test_list_item_validation.ItemValidationTest)
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate
element: .has-error
```

Não os descreverei em detalhes, mas, pelo menos, aqui estão os testes de unidade de que você precisará:

### lists/tests/test\_api.py (ch36l027)

```
from lists.forms import DUPLICATE_ITEM_ERROR, EMPTY_ITEM_ERROR
[...]
def post_empty_input(self):
 list_ = List.objects.create()
 return self.client.post(
 self.base_url.format(list_.id),
 data={'text': ''}
)

def test_for_invalid_input_nothing_saved_to_db(self):
 self.post_empty_input()
 self.assertEqual(Item.objects.count(), 0)
```

```

def test_for_invalid_input_returns_error_code(self):
 response = self.post_empty_input()
 self.assertEqual(response.status_code, 400)
 self.assertEqual(
 json.loads(response.content.decode('utf8')),
 {'error': EMPTY_ITEM_ERROR}
)

def test_duplicate_items_error(self):
 list_ = List.objects.create()
 self.client.post(
 self.base_url.format(list_.id), data={'text': 'thing'}
)
 response = self.client.post(
 self.base_url.format(list_.id), data={'text': 'thing'}
)
 self.assertEqual(response.status_code, 400)
 self.assertEqual(
 json.loads(response.content.decode('utf8')),
 {'error': DUPLICATE_ITEM_ERROR}
)

```

Do lado do JavaScript, temos:

## lists/static/tests/tests.html (ch36l029-2)

```

QUnit.test("should display errors on post failure", function (assert) {
 var url = '/listitemsapi/';
 window.Superlists.initialize(url);
 server.respondWith('POST', url, [
 400,
 {"Content-Type": "application/json"},
 JSON.stringify({'error': 'something is amiss'})
]);
 $('#has-error').hide();

 $('#id_item_form').submit();
 server.respond(); // post

 assert.equal($('#has-error').is(':visible'), true);
 assert.equal($('#has-error .help-block').text(), 'something is amiss');

```

```
});
```

```
QUnit.test("should hide errors on post success", function (assert) {
 [...]
```

Você também vai querer fazer algumas modificações em *base.html* a fim de deixá-lo compatível para exibir tanto os erros de Django (que, por enquanto, a página inicial ainda utiliza) quanto os erros de JavaScript:

## lists/templates/base.html (ch36l031)

```
@@ -51,17 +51,21 @@
 <div class="col-md-6 col-md-offset-3 jumbotron">
 <div class="text-center">
 <h1>{% block header_text %}{% endblock %}</h1>
+
 {% block list_form %}
 <form id="id_item_form" method="POST" action="{% block [...] %}"
 {{ form.text }}
 {% csrf_token %}
- {% if form.errors %}
- <div class="form-group has-error">
- <div class="help-block">{{ form.text.errors }}</div>
+ <div class="form-group has-error">
+ <div class="help-block">
+ {% if form.errors %}
+ {{ form.text.errors }}
+ {% endif %}
 </div>
- {% endif %}
+ </div>
 </form>
 {% endblock %}
+
 </div>
 </div>
</div>
```

No final, você deverá chegar a uma execução de testes de JavaScript como esta:

20 assertions of 20 passed, 0 failed.

1. errors should be hidden on keypress (1)
2. errors aren't hidden if there is no keypress (1)
3. should call updateItems on initialize (1)
4. updateItems should get correct url by ajax (3)
5. updateItems should fill in lists table from ajax response (3)
6. should intercept form submit and do ajax post (4)
7. should call updateItems after successful post (1)
8. should not intercept form submit if no api url passed in (1)
9. should display errors on post failure (2)
10. should hide errors on post success (1)
11. should display generic error if no error json (2)

Uma execução completa dos testes deve passar, incluindo todos os FTs:

```
$ python manage.py test
[...]
Ran 81 tests in 62.029s
OK
```

Adorável.

E eis que temos a nossa API REST com Django, desenvolvida manualmente. Se você precisar de uma pista para terminá-la por conta própria, consulte o repositório ([https://github.com/hjwp/book-example/tree/appendix\\_rest\\_api](https://github.com/hjwp/book-example/tree/appendix_rest_api)).

Contudo, eu jamais sugeriria construir uma API REST no Django sem, no mínimo, dar uma olhada no *Django-Rest-Framework*. Esse será o assunto de nosso próximo apêndice! Continue lendo, Macduff.

## Dicas sobre APIs REST



## *Remova a duplicação de URLs*

De certo modo, os URLs são mais importantes para uma API do que o são para uma aplicação voltada a um navegador. Procure reduzir o número de vezes que você os deixa fixos no código em seus testes.

*Não trabalhe com strings JSON puras*

json.loads e json.dumps são seus amigos.

*Sempre utilize uma biblioteca de simulação de Ajax para seus testes de JavaScript*

O Sinon é bom. O Jasmine tem a própria biblioteca, assim como o Angular.

*Tenha a degradação elegante e as melhorias progressivas em mente*

Particularmente, se você estiver passando de um site estático para um site mais orientado a JavaScript, considere manter pelo menos o núcleo dos recursos de seu site funcionando sem JavaScript.



## APÊNDICE G

# Django-Rest-Framework

Depois de ter “desenvolvido a nossa própria” API REST no último apêndice, é hora de dar uma olhada no Django-Rest-Framework (<http://www.django-rest-framework.org/>) – uma opção disponível para muitos desenvolvedores de Python/Django que estejam construindo APIs. Assim como o Django tem como objetivo oferecer a você todas as ferramentas básicas necessárias para construir um site orientado a banco de dados (um ORM, templates e assim por diante), o DRF visa a lhe dar todas as ferramentas de que você precisará para construir uma API e, desse modo, evitar que você tenha que escrever código boilerplate repetidamente.

Ao escrever este apêndice, um dos pontos principais com os quais tive que lutar foi obter exatamente a mesma API que acabei de implementar de modo manual, replicada pelo DRF. Ter o mesmo layout de URLs e as mesmas estruturas de dados JSON que defini provou ser um desafio e tanto, e tive a sensação de estar lutando contra o framework.

Esse é sempre um sinal de advertência. As pessoas que desenvolveram o Django-Rest-Framework são muito mais espertas do que eu, e viram muito mais APIs REST do que eu; se elas têm opiniões sobre o modo como as soluções “devam” parecer, talvez meu tempo seria mais bem gasto se eu tentasse me adaptar e trabalhar com suas visões de mundo, em vez de impor minhas próprias ideias preconcebidas.

“Não lute contra o framework” foi um dos melhores conselhos que já ouvi. Siga o fluxo ou, quem sabe, reavalie se você realmente quer usar um framework.

Trabalharemos a partir da API que tínhamos no final do último apêndice, e veremos se é possível reescrevê-la de modo a usar o DRF.

## Instalação

Um rápido `pip install` nos disponibiliza o DRF. Estou usando a versão mais recente, que era a 3.5.4 quando escrevi este livro:

```
$ pip install djangorestframework
```

Além disso, adicionamos `rest_framework` em `INSTALLED_APPS` no arquivo `settings.py`:

### superlists/settings.py

```
INSTALLED_APPS = [
 #'django.contrib.admin',
 'django.contrib.auth',
 'django.contrib.contenttypes',
 'django.contrib.sessions',
 'django.contrib.messages',
 'django.contrib.staticfiles',
 'lists',
 'accounts',
 'functional_tests',
 'rest_framework',
]
```

## Serializadores (bem, na verdade, ModelSerializers)

O tutorial do Django-Rest-Framework (<http://bit.ly/2t6T6eX>) é um recurso muito bom para conhecer o DRF. O primeiro recurso com o qual você vai deparar são os serializadores – em nosso caso especificamente, os “ModelSerializers”. Eles são o modo do DRF de converter modelos de banco de dados do Django para JSON (ou possivelmente para outros formatos), que poderá ser enviado para transmissão:

## lists/api.py (ch37I003)

```
from lists.models import List, Item
[...]
from rest_framework import routers, serializers, viewsets
```

```
class ItemSerializer(serializers.ModelSerializer):
```

```
 class Meta:
 model = Item
 fields = ('id', 'text')
```

```
class ListSerializer(serializers.ModelSerializer):
 items = ItemSerializer(many=True, source='item_set')
```

```
 class Meta:
 model = List
 fields = ('id', 'items',)
```

## Viewsets (bem, na verdade, ModelViewsets) e roteadores

Um `ModelViewSet` é o modo do DRF de definir todas as maneiras diferentes com as quais podemos interagir com os objetos de um modelo em particular por meio de sua API. Depois de lhe informar em quais modelos você está interessado (usando o atributo `queryset`) e como serializá-los (`serializer_class`), ele então fará o restante – construindo as views automaticamente para você, permitindo listar, obter, atualizar e até mesmo apagar objetos.

Eis tudo que precisamos fazer para um `ViewSet` ser capaz de obter itens de uma lista em particular:

## lists/api.py (ch37I004)

```
class ListViewSet(viewsets.ModelViewSet):
 queryset = List.objects.all()
 serializer_class = ListSerializer
```



```
router = routers.SimpleRouter()
router.register(r'lists', ListViewSet)
```

Um *roteador* (router) é o modo como o DRF constrói a configuração de URLs automaticamente, mapeando-os para as funcionalidades oferecidas pelo ViewSet.

Nesse ponto, podemos começar a apontar o nosso *urls.py* para o novo roteador, ignorando o antigo código da API, e ver como os nossos testes lidam com o novo código:

## superlists/urls.py (ch371005)

```
[...]
from lists.api import urls as api_urls
from lists.api import router

urlpatterns = [
 url(r'^$', list_views.home_page, name='home'),
 url(r'^lists/', include(list_urls)),
 url(r'^accounts/', include(accounts_urls)),
 # url(r'^api/', include(api_urls)),
 url(r'^api/', include(router.urls)),
]
```

Isso faz com que vários de nossos testes falhem:

```
$ python manage.py test lists
```

```
[...]
django.urls.exceptions.NoReverseMatch: Reverse for 'api_list' not found.
'api_list' is not a valid view function or pattern name.
[...]
AssertionError: 405 != 400
[...]
AssertionError: {'id': 2, 'items': [{'id': 2, 'text': 'item 1'}, {'id': 3,
'text': 'item 2'}]} != [{'id': 2, 'text': 'item 1'}, {'id': 3, 'text': 'item
2'}]
```

```

Ran 54 tests in 0.243s
```

```
FAILED (failures=4, errors=10)
```

Vamos dar uma olhada nesses dez erros antes, todos informando que não podem inverter `api_list`. Isso ocorre porque o roteador do DRF utiliza uma convenção de nomenclatura diferente para os URLs, em comparação com aquela que usamos quando fizemos a implementação manualmente. Com base nos tracebacks, você verá que eles estão ocorrendo quando renderizamos um template. É em *list.html*. Podemos corrigir isso em um só local; `api_list` passa a ser `list-detail`:

## lists/templates/list.html (ch371006)

```
<script>
$(document).ready(function () {
 var url = "{% url 'list-detail' list.id %}";
});
</script>
```

Com isso, ficamos reduzidos a apenas quatro falhas:

```
$ python manage.py test lists
[...]
FAIL: test_POSTing_a_new_item (lists.tests.test_api.ListAPITest)
[...]
FAIL: test_duplicate_items_error (lists.tests.test_api.ListAPITest)
[...]
FAIL: test_for_invalid_input_returns_error_code
(lists.tests.test_api.ListAPITest)
[...]
FAIL: test_get_returns_items_for_correct_list
(lists.tests.test_api.ListAPITest)
[...]
FAILED (failures=4)
```

Vamos negar todos os testes de validação (inserindo `DONT`) por enquanto e deixar essa complexidade para mais tarde:

## lists/tests/test\_api.py (ch371007)

```
[...]
def DONTtest_for_invalid_input_nothing_saved_to_db(self):
 [...]
```

```

def DONTtest_for_invalid_input_returns_error_code(self):
 [...]
def DONTtest_duplicate_items_error(self):
 [...]

```

Agora temos somente duas falhas:

```

FAIL: test_POSTing_a_new_item (lists.tests.test_api.ListAPITest)
[...]
self.assertEqual(response.status_code, 201)
AssertionError: 405 != 201
[...]
FAIL: test_get_returns_items_for_correct_list
(lists.tests.test_api.ListAPITest)
[...]
AssertionError: {'id': 2, 'items': [{'id': 2, 'text': 'item 1'}, {'id': 3,
'text': 'item 2'}]} != [{'id': 2, 'text': 'item 1'}, {'id': 3, 'text': 'item
2'}]
[...]
FAILED (failures=2)

```

Vamos ver a última falha primeiro.

A configuração default do DRF disponibiliza uma estrutura de dados um pouco diferente daquela que implementamos manualmente – fazer um GET para uma lista nos dá o seu ID e, então, os itens de lista estão em uma chave chamada “items”. Isso implica uma pequena modificação em nosso teste de unidade, antes que ele volte a passar:

## lists/tests/test\_api.py (ch371008)

```

@@ -23,10 +23,10 @@ class ListAPITest(TestCase):
 response = self.client.get(self.base_url.format(our_list.id))
 self.assertEqual(
 json.loads(response.content.decode('utf8')),
- [
+ {'id': our_list.id, 'items': [
 {'id': item1.id, 'text': item1.text},
 {'id': item2.id, 'text': item2.text},
-]
+]}
)

```

Esse é o GET para obter os itens de lista ordenados (e, como veremos mais adiante, teremos várias outras informações gratuitamente também). Que tal adicionar itens novos usando POST?

## Um URL diferente para POST de itens

Esse é o ponto em que desisti de lutar contra o framework e simplesmente vi a direção para a qual o DRF queria me levar. Embora possível, é bem complicado fazer um POST para a ViewSet de “listas” a fim de adicionar um item em uma lista.

Em vez disso, o modo mais simples é fazer um post para uma view de itens, e não para uma view de lista:

### lists/api.py (ch371009)

```
class ItemViewSet(viewsets.ModelViewSet):
 serializer_class = ItemSerializer
 queryset = Item.objects.all()
```

```
[...]
router.register(r'items', ItemViewSet)
```

Portanto, isso implica mudar um pouco o teste, retirando todos os testes de POST de ListAPITest e passando-os para uma nova classe de teste, ItemsAPITest:

### lists/tests/test\_api.py (ch371010)

```
@@ -1,3 +1,4 @@
import json
+from django.core.urlresolvers import reverse
from django.test import TestCase
from lists.models import List, Item
@@ -31,9 +32,13 @@ class ListAPITest(TestCase):

+
+class ItemsAPITest(TestCase):
```

```

+ base_url = reverse('item-list')
+
 def test_POSTing_a_new_item(self):
 list_ = List.objects.create()
 response = self.client.post(
- self.base_url.format(list_.id),
- {'text': 'new item'},
+ self.base_url,
+ {'list': list_.id, 'text': 'new item'},
)
 self.assertEqual(response.status_code, 201)

```

Eis o resultado:

```
django.db.utils.IntegrityError: NOT NULL constraint failed: lists_item.list_id
```

A falha permanecerá até adicionarmos o ID de lista à nossa serialização de itens; caso contrário, não saberemos em que lista estamos atuando:

## lists/api.py (ch37I011)

```

class ItemSerializer(serializers.ModelSerializer):

 class Meta:
 model = Item
 fields = ('id', 'list', 'text')

```

Com isso, temos outra pequena mudança associada ao teste:

## lists/tests/test\_api.py (ch37I012)

```

@@ -25,8 +25,8 @@ class ListAPITest(TestCase):
 self.assertEqual(
 json.loads(response.content.decode('utf8')),
 {'id': our_list.id, 'items': [
- {'id': item1.id, 'text': item1.text},
- {'id': item2.id, 'text': item2.text},
+ {'id': item1.id, 'list': our_list.id, 'text': item1.text},
+ {'id': item2.id, 'list': our_list.id, 'text': item2.text},
]}
)

```

## Adaptando o lado do cliente

Nossa API não devolve mais um array linear dos itens de uma lista. Ela devolve um objeto com um atributo `.items` representando os itens. Isso implica um pequeno ajuste em nossa função `updateItems`:

### lists/static/list.js (ch37|013)

```
@@ -3,8 +3,8 @@ window.Superlists = {};
window.Superlists.updateItems = function (url) {
 $.get(url).done(function (response) {
 var rows = "";
- for (var i=0; i<response.length; i++) {
- var item = response[i];
+ for (var i=0; i<response.items.length; i++) {
+ var item = response.items[i];
 rows += '\n<tr><td>' + (i+1) + ': ' + item.text + '</td></tr>';
 }
 $('#id_list_table').html(rows);
}
```

Como estamos usando URLs diferentes para GET em listas e POST em itens, ajustamos levemente a função `initialize` também. Em vez de usar vários argumentos, passaremos a utilizar um objeto `params` contendo a configuração necessária:

### lists/static/list.js

```
@@ -11,23 +11,24 @@ window.Superlists.updateItems = function (url) {
 });
};

-window.Superlists.initialize = function (url) {
+window.Superlists.initialize = function (params) {
 $('input[name="text"]').on('keypress', function () {
 $('.has-error').hide();
 });

- if (url) {
- window.Superlists.updateItems(url);
+ if (params) {
+ window.Superlists.updateItems(params.listApiUrl);
}
```

```

 var form = $('#id_item_form');
 form.on('submit', function(event) {
 event.preventDefault();
- $.post(url, {
+ $.post(params.itemsApiUrl, {
+ 'list': params.listId,
 'text': form.find('input[name="text"]').val(),
 'csrfmiddlewaretoken': form.find('input[name="csrfmiddlewaretoken"]').val(),
 }).done(function () {
 $('.has-error').hide();
- window.Superlists.updateItems(url);
+ window.Superlists.updateItems(params.listApiUrl);
 }).fail(function (xhr) {
 $('.has-error').show();
 if (xhr.responseJSON && xhr.responseJSON.error) {

```

Isso deve se refletir em *list.html*:

## lists/templates/list.html (ch37I014)

```

$(document).ready(function () {
 window.Superlists.initialize({
 listApiUrl: "{% url 'list-detail' list.id %}",
 itemsApiUrl: "{% url 'item-list' %}",
 listId: {{ list.id }},
 });
});

```

Na verdade, essa mudança deve bastar para que o FT básico volte a funcionar:

```

$ python manage.py test functional_tests.test_simple_list_creation
[...]
Ran 2 tests in 15.635s

```

OK

Há mais algumas alterações a serem feitas para tratamento de erros, que poderão ser exploradas no repositório associado a este apêndice ([https://github.com/hjwp/book-example/blob/appendix\\_DjangoRestFramework/lists/api.py](https://github.com/hjwp/book-example/blob/appendix_DjangoRestFramework/lists/api.py)), caso você esteja curioso.

## O que o Django-Rest-Framework oferece a você

Talvez você esteja se perguntando qual foi o sentido de usar esse framework.

### Configuração em vez de código

Bem, a primeira vantagem é que transformei minha antiga função de view procedural em uma sintaxe mais declarativa:

#### lists/api.py

```
def list(request, list_id):
 list_ = List.objects.get(id=list_id)
 if request.method == 'POST':
 form = ExistingListItemForm(for_list=list_, data=request.POST)
 if form.is_valid():
 form.save()
 return HttpResponse(status=201)
 else:
 return HttpResponse(
 json.dumps({'error': form.errors['text'][0]}),
 content_type='application/json',
 status=400
)
 item_dicts = [
 {'id': item.id, 'text': item.text}
 for item in list_.item_set.all()
]
 return HttpResponse(
 json.dumps(item_dicts),
 content_type='application/json'
)
```

Se comparar esse código com a versão final com DRF, você perceberá que estamos totalmente configurados agora:

#### lists/api.py

```
class ItemSerializer(serializers.ModelSerializer):
 text = serializers.CharField(
 allow_blank=False, error_messages={'blank': EMPTY_ITEM_ERROR}
```



```
)
```

```
class Meta:
 model = Item
 fields = ('id', 'list', 'text')
 validators = [
 UniqueTogetherValidator(
 queryset=Item.objects.all(),
 fields=('list', 'text'),
 message=DUPLICATE_ITEM_ERROR
)
]
```

```
class ListSerializer(serializers.ModelSerializer):
 items = ItemSerializer(many=True, source='item_set')
```

```
class Meta:
 model = List
 fields = ('id', 'items',)
```

```
class ListViewSet(viewsets.ModelViewSet):
 queryset = List.objects.all()
 serializer_class = ListSerializer
```

```
class ItemViewSet(viewsets.ModelViewSet):
 serializer_class = ItemSerializer
 queryset = Item.objects.all()
```

```
router = routers.SimpleRouter()
router.register(r'lists', ListViewSet)
router.register(r'items', ItemViewSet)
```

## Funcionalidades gratuitas

A segunda vantagem é que, ao usar ModelSerializer, ViewSet e roteadores do DRF, na verdade acabei com uma API muito mais

abrangente do que aquela que desenvolvi manualmente.

- Todos os métodos HTTP – GET, POST, PUT, PATCH, DELETE e OPTIONS – agora funcionam prontamente para todos os URLs de listas e de itens.
- Uma versão da API com documentação própria, que permite navegação, está disponível em <http://localhost:8000/api/lists/> e em <http://localhost:8000/api/items/>. (Figura G1.; teste isso!)

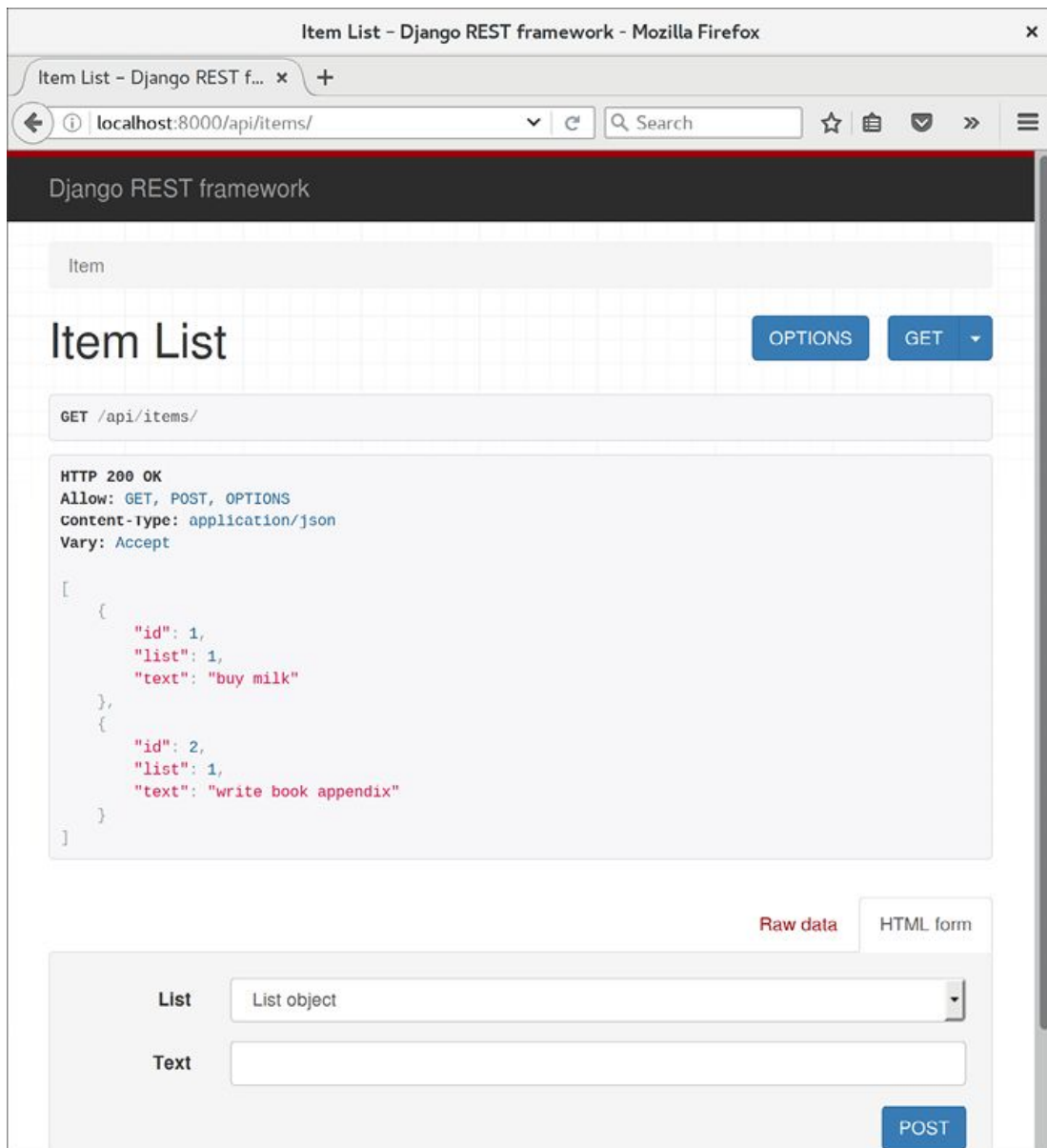


Figure G.1 – Uma API navegável gratuita para seus usuários.

Há mais informações na documentação do DRF (<http://www.django-rest-framework.org/topics/documenting-your-api/#self-describing-apis>), mas esses dois recursos são extremamente convenientes para serem oferecidos aos usuários finais de sua API.

Em suma, o DRF é uma ótima maneira de gerar APIs quase automaticamente, com base em sua estrutura de modelos existente. Se estiver usando o Django, dê uma olhada nele sem falta antes de começar a desenvolver manualmente o seu próprio código de API.

## **Dicas sobre o Django-Rest-Framework**

## *Não lute contra o framework*

Seguir o fluxo geralmente é a melhor maneira de permanecer produtivo. Isso ou, talvez, não usar o framework. Ou use-o em um nível mais baixo.

## *Roteadores e ViewSets para o princípio do mínimo de surpresas*

Uma das vantagens do DRF é o fato de suas ferramentas genéricas como roteadores e ViewSets lhe darem uma API bem previsível, com defaults sensatos para seus endpoints, uma estrutura de URL e respostas para diferentes métodos HTTP.

### *Dê uma olhada na versão navegável, com documentação própria*

Dê uma olhada nos endpoints de sua API em um navegador. O DRF responde de modo diferente quando detecta que sua API está sendo acessada por um navegador web “normal”, e exibe uma versão de si mesma com uma documentação própria muito elegante, que você poderá compartilhar com seus usuários.



# APÊNDICE H

## Folha de cola

Atendendo a pedidos, essa “folha de cola” está baseada, de modo geral, nas pequenas caixas de revisão/resumo do final de cada capítulo. A ideia é oferecer alguns lembretes e referências para os capítulos nos quais você poderá encontrar mais informações para reativar a sua memória. Espero que ache útil!

### Configuração inicial do projeto

- Comece com uma *História de Usuário* e mapeie-a para um primeiro *teste funcional*.
- Escolha um framework de testes – o unittest é conveniente, e opções como py.test, nose ou Green também podem oferecer algumas vantagens.
- Execute o teste funcional e veja a sua primeira *falha esperada*.
- Escolha um framework web, como o Django, e descubra como executar *testes de unidade* aí.
- Crie o seu primeiro *teste de unidade* para lidar com a falha atual de FT, e veja-o falhar.
- Faça o seu *primeiro commit* em um VCS como o *Git*.

Capítulos relevantes: Capítulo 1, Capítulo 2, Capítulo 3

### Fluxo de trabalho básico do TDD

- TDD com laço duplo (Figura H.1)
- Vermelho, Verde, Refatorar



- Triangulação
- A folha de rascunho
- “Três acertos e refatorar”
- ”De estado funcional para estado funcional”
- “YAGNI” (You Ain’t Gonna Need It, ou Você Não Vai Precisar Disso).

Eis um fluxograma (Figura H.1) que mostra os testes funcionais como o ciclo geral, e os testes de unidade ajudando na implementação:

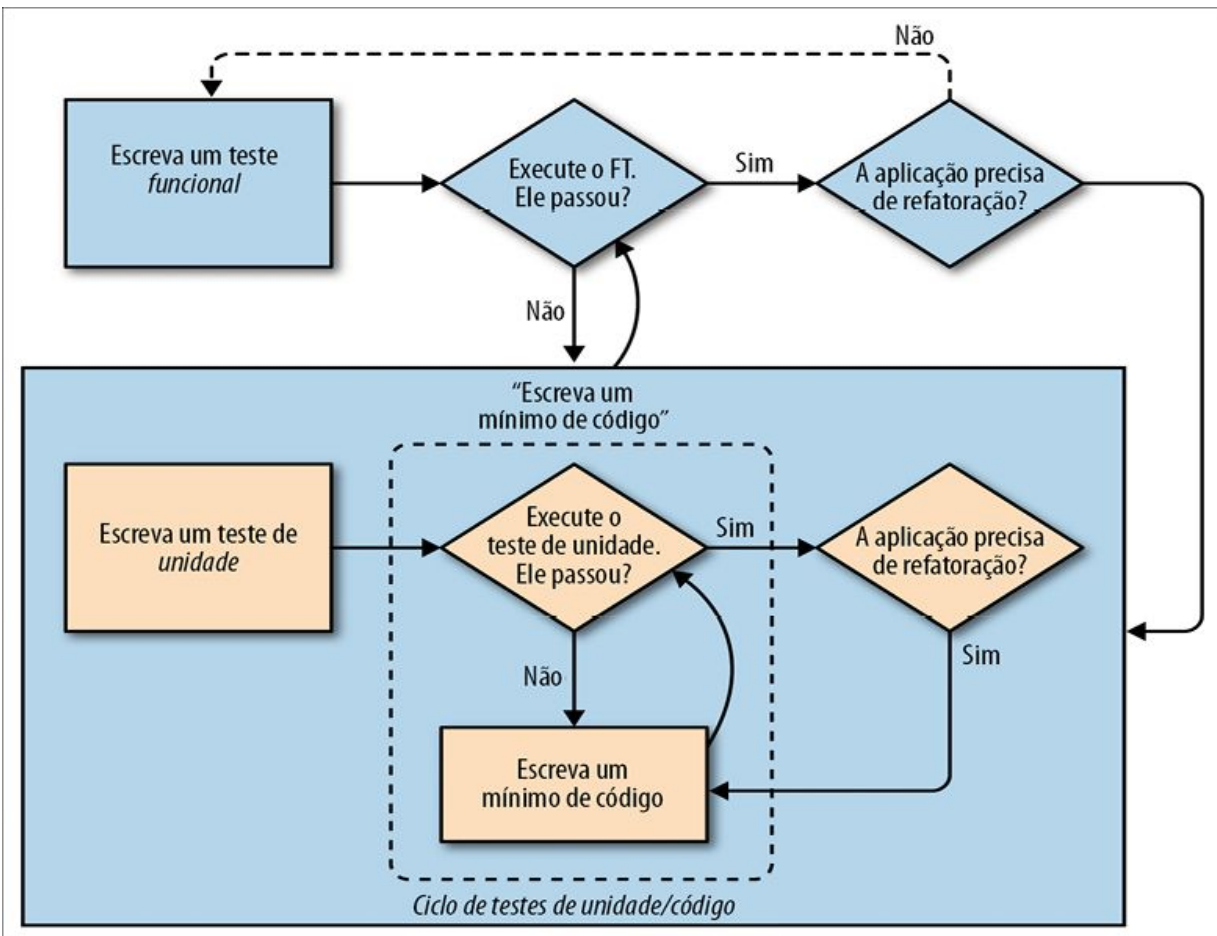


Figura H.1 – O processo de TDD com testes funcionais e testes de unidade.

Capítulos relevantes: Capítulo 4, Capítulo 5, Capítulo 7

## **Para além dos testes somente de desenvolvimento**

- Inicie os testes de sistema cedo. Certifique-se de que seus componentes funcionam em conjunto: servidor web, conteúdo estático, banco de dados.
- Construa um ambiente de staging correspondente ao seu ambiente de produção; execute aí a sua suíte de FT.
- Automatize seus ambientes de staging e de produção:
  - PaaS *versus* VPS
  - Fabric
  - Gerenciamento de configuração (Chef, Puppet, Salt, Ansible)
  - Vagrant
- Pense nos pontos complicados da implantação: o banco de dados, arquivos estáticos, dependências, como personalizar configurações, e assim por diante.
- Construa um servidor de CI o mais rápido possível, de modo que você não precise depender de sua autodisciplina para ver os testes executarem.

Capítulos relevantes: Capítulo 9, Capítulo 11, Capítulo 24, Apêndice C

## **Melhores práticas de teste em geral**

- Cada teste deve verificar um só aspecto.
- Tenha um arquivo de teste por arquivo de código-fonte da aplicação.
- Considere pelo menos um teste placeholder para cada função e classe, independentemente do quão simples elas sejam.
- “Não teste constantes”.
- Procure testar comportamentos, em vez de testar a

implementação.

- Tente pensar além do caminho feliz no código e pense nos casos extremos e nos casos de erro.

Capítulos relevantes: Capítulo 4, Capítulo 13, Capítulo 14

## **Melhores práticas para testes funcionais/Selenium**

- Utilize esperas explícitas em vez de implícitas, e o padrão de interação/espera.
- Evite duplicação de código de testes – métodos auxiliares em uma classe-base e o padrão Page são possíveis soluções.
- Evite testes duplicados de funcionalidades. Se você tiver um teste que inclua um processo que consuma tempo (por exemplo, login), considere formas de não o executar em outros testes (entretanto, esteja ciente de interações inesperadas entre partes aparentemente não relacionadas das funcionalidades).
- Dê uma olhada em ferramentas de BDD como outra maneira de estruturar seus FTs.

Capítulos relevantes: Capítulo 21, Capítulo 24, Capítulo 25

## **Outside-In, isolamento de testes versus testes integrados e simulação**

Lembre-se dos motivos pelos quais escrevemos testes, antes de tudo:

- para garantir que a aplicação esteja correta e evitar regressões;
- para nos ajudar a escrever um código limpo, possível de manter;
- para possibilitar um fluxo de trabalho rápido e produtivo.

Com esses objetivos em mente, pense nos diferentes tipos de testes e nas negociações de custo-benefício entre eles:

### *Testes funcionais*

- Oferecem a melhor garantia de que a sua aplicação realmente funciona corretamente, do ponto de vista do usuário.
- No entanto: tem um ciclo de feedback mais lento.
- E não necessariamente ajudam você a escrever um código mais limpo.

### *Testes integrados (dependentes, por exemplo, de ORM ou do Django Test Client)*

- São rápidos para serem escritos.
- São fáceis de entender.
- Avisarão você acerca de qualquer problema de integração.
- No entanto: nem sempre levam a um bom design (isso cabe a você!).
- E geralmente são mais lentos que os testes isolados.

### *Testes isolados (“com simulação”)*

- Envolvem o trabalho mais árduo.
- Podem ser mais difíceis de ler e de entender.
- No entanto: são os melhores para orientar você em direção a um design melhor.
- E são os mais rápidos para executar.

Se você se vir escrevendo testes com muitos mocks e eles estiverem complicados, lembre-se de “ouvir seus testes” – testes feios, com simulação, talvez estejam tentando lhe dizer que o seu código poderia ser simplificado.

Capítulos relevantes: Capítulo 22, Capítulo 23, Capítulo 26



# APÊNDICE I

## O que fazer em seguida

Neste apêndice, ofereço algumas sugestões de tópicos a serem investigados a seguir para desenvolver suas habilidades com testes e aplicá-las em algumas das novas tecnologias interessantes em desenvolvimento web (quando escrevi este livro!).

Espero, no mínimo, transformar cada um desses itens em uma espécie de postagem de blog, senão em um futuro apêndice no livro. Espero também, com o passar do tempo, gerar códigos de exemplo para todos eles. Portanto, dê uma olhada em <http://www.obeythetestinggoat.com> e veja se há alguma atualização.

Ou por que você não tenta me superar e escreve a própria postagem de blog narrando a sua tentativa associada a qualquer um desses tópicos?

Ficarei muito feliz em responder às perguntas e oferecer dicas e orientações sobre todos esses assunto, portanto, se você se vir tentando e não sabendo como continuar, por favor, não hesite em entrar em contato comigo em [obeythetestinggoat@gmail.com](mailto:obeythetestinggoat@gmail.com)!

### **Notificações – tanto no site quanto por email**

Seria bom se os usuários fossem notificados quando alguém compartilhasse uma lista com eles.

Você pode usar `django-notifications` para exibir uma mensagem aos usuários na próxima vez que eles atualizarem a tela. Para isso você precisará de dois navegadores em seu FT.

E/ou você poderia enviar notificações por email. Investigue os recursos do Django para testes com emails. Então decida se isso é

tão importante a ponto de você precisar de testes reais com emails reais. Utilize a biblioteca IMAPClient para buscar emails de verdade de uma conta de webmail para testes.

## **Mude para o Postgres**

O SQLite é um banco de dados pequeno incrível, mas não terá um bom desempenho quando você tiver mais de um processo worker para web atendendo às requisições de seu site. O Postgres atualmente é o banco de dados favorito de todos, portanto descubra como instalá-lo e configurá-lo.

Você precisará definir um local para armazenar seus nomes de usuário e senhas para os servidores local, de staging, de produção e do Postgres. Como, por questões de segurança, provavelmente você não vai querê-los em seu repositório de código, procure maneiras de modificar os seus scripts de implantação a fim de passá-los na linha de comando. Variáveis de ambiente são uma solução popular como um local para mantê-los...

Faça experimentos em que você mantenha seus testes de unidade executando com o SQLite e compare a sua velocidade com a velocidade da execução no Postgres. Realize a configuração de modo que a sua máquina local utilize SQLite para testes, mas o seu servidor de CI utilize o Postgres.

## **Execute seus testes com navegadores diferentes**

O Selenium aceita todo tipo de navegadores diferentes, incluindo o Chrome e o Internet Explorer. Teste ambos e veja se a sua suíte de FT se comporta de algum modo diferente.

Você também deve testar um navegador “headless”, como o PhantomJS.

Em minha experiência, mudar de navegador tende a expor todo tipo de condições de concorrência (race conditions) em testes com o Selenium; provavelmente você precisará usar muito mais o padrão

de interação/espera (em particular, para o PhantomJS).

## Testes de 404 e 500

Um site profissional precisa ter páginas de erro com uma boa aparência. Testar uma página 404 é fácil, mas provavelmente você precisará de uma view personalizada para “gerar uma exceção de propósito” a fim de testar a página 500.

## Site de administração do Django

Pense em uma história em que um usuário envie um email a você querendo “reivindicar” uma lista anônima. Suponha que implementemos uma solução manual para isso envolvendo o administrador do site alterando manualmente o registro usando o site de administração do Django.

Descubra como ativar o site de administração e experimente usá-lo. Escreva um FT que mostre um usuário comum, não logado, criando uma lista; em seguida, faça um usuário administrador realizar login, acessar o site de administração e atribuir a lista ao usuário. O usuário poderá, então, vê-la em sua página “My Lists”.

## Escreva alguns testes de segurança

Expanda os testes de login, minhas listas e de compartilhamento – o que você precisa escrever para se certificar de que seus usuários possam fazer somente o que estão autorizados a fazer?

## Teste para saber se a degradação é elegante

O que aconteceria se o Persona parasse de funcionar? Poderíamos, pelo menos, exibir uma mensagem de erro pedindo desculpas aos nossos usuários?

- **Dica:** uma forma de simular o Persona parando de funcionar é com um hack em seu arquivo de hosts (em `/etc/hosts` ou em



`c:\Windows\System32\drivers\etc`). Lembre-se de restaurar os dados no `tearDown` do teste!

- Pense no lado do servidor, bem como no lado do cliente.

## Testes de caching e de desempenho

Descubra como instalar e configurar o `memcached`. Descubra como usar o `ab` do Apache para executar um teste de desempenho. Como é o desempenho com e sem caching? Você é capaz de escrever um teste automatizado que falhará caso o caching não esteja ativado? E o que dizer do terrível problema de invalidação de cache? Os testes podem ajudar você a garantir que a sua lógica de invalidação de cache é robusta?

## Frameworks MVC para JavaScript

As bibliotecas JavaScript que permitem implementar um padrão Model-View-Controller (Modelo-Visão-Controlador) do lado cliente estão extremamente em alta nos dias atuais. Listas de tarefas são umas das aplicações demo favoritas para eles, portanto deve ser bem fácil converter o site para que seja um site single-page, em que todos os acréscimos em listas ocorram em JavaScript.

Escolha um framework – talvez o `Backbone.js` ou o `Angular.js` – e crie um spike com uma implementação. Cada framework tem as próprias preferências acerca de como escrever testes de unidade, portanto aprenda aquela apropriada ao respectivo framework e veja se você gosta dela.

## Execução assíncrona e websockets

Suponha que dois usuários estejam trabalhando na mesma lista ao mesmo tempo. Não seria interessante ver atualizações em tempo real, de modo que, se a outra pessoa acrescentar um item na lista, você o verá imediatamente? Uma conexão persistente entre o cliente e o servidor usando websockets é a forma de fazer isso

funcionar.

Dê uma olhada em um dos servidores web assíncronos para Python – Tornado, gevent, Twisted – e veja se consegue usá-lo para implementar notificações dinâmicas.

Para testar isso, será necessário ter duas instâncias de navegador (como fizemos nos testes de compartilhamento de listas), e verificar se as notificações das ações de uma pessoa aparecem para a outra, sem a necessidade de atualizar a página...

## Mude para `py.test`

O `py.test` permite escrever testes de unidade com menos boilerplate. Tente converter alguns de seus testes de unidade para que usem `py.test`. Talvez você precise utilizar um plugin para que ele funcione bem com o Django.

## Dê uma olhada no `coverage.py`

O `coverage.py` de Ned Batchelder informará qual é a *abrangência de seus testes* – qual é o percentual de seu código que está coberto pelos testes. Teoricamente, como estamos usando um TDD rigoroso, devemos ter sempre uma cobertura de 100%. Porém, é bom saber com certeza, e é também uma ferramenta muito útil para trabalhar com projetos que não tiveram testes desde o princípio.

## Criptografia do lado do cliente

Eis um cenário divertido: o que aconteceria se nossos usuários fossem paranoicos em relação à NSA e decidissem que não querem mais confiar suas listas na Nuvem? Você seria capaz de construir um sistema de criptografia JavaScript, em que o usuário pudesse fornecer uma senha para criptografar o texto do item de sua lista antes que esse fosse enviado para o servidor?

Uma forma de testar isso seria com um usuário “administrador” que acessasse a view de administração do Django a fim de inspecionar

as listas dos usuários e verificasse se elas estão armazenadas de forma criptografada no banco de dados.

## **Suas sugestões aqui**

O que você acha que eu deveria colocar aqui? Dê sugestões, por favor!



# APÊNDICE J

## Código-fonte dos exemplos

Todos os códigos de exemplo que usei no livro estão disponíveis em meu repositório (<https://github.com/hjwp/book-example/>) no GitHub. Portanto, se algum dia você quiser comparar o seu código com o meu, poderá dar uma olhada lá.

Cada capítulo tem o próprio branch nomeado com base nele, da seguinte maneira:

*Capítulo 1* – [https://github.com/hjwp/book-example/tree/chapter\\_01](https://github.com/hjwp/book-example/tree/chapter_01)

Saiba que cada branch contém todos os commits daquele capítulo, portanto o seu estado representa o código no *final* do capítulo.

### Lista completa dos links para cada capítulo

Capítulo 1 – [https://github.com/hjwp/book-example/tree/chapter\\_01](https://github.com/hjwp/book-example/tree/chapter_01)

Capítulo 2 – [https://github.com/hjwp/book-example/tree/chapter\\_02\\_unittest](https://github.com/hjwp/book-example/tree/chapter_02_unittest)

Capítulo 3 – [https://github.com/hjwp/book-example/tree/chapter\\_unit\\_test\\_first\\_view](https://github.com/hjwp/book-example/tree/chapter_unit_test_first_view)

Capítulo 4 – [https://github.com/hjwp/book-example/tree/chapter\\_philosophy\\_and\\_refactoring](https://github.com/hjwp/book-example/tree/chapter_philosophy_and_refactoring)

Capítulo 5 – [https://github.com/hjwp/book-example/tree/chapter\\_post\\_and\\_database](https://github.com/hjwp/book-example/tree/chapter_post_and_database)

Capítulo 6 – [https://github.com/hjwp/book-example/tree/chapter\\_explicit\\_waits\\_1](https://github.com/hjwp/book-example/tree/chapter_explicit_waits_1)

Capítulo 7 – <https://github.com/hjwp/book->

*example/tree/chapter\_working\_incrementally*

Capítulo 8 – [https://github.com/hjwp/book-example/tree/chapter\\_prettification](https://github.com/hjwp/book-example/tree/chapter_prettification)

Capítulo 9 – [https://github.com/hjwp/book-example/tree/chapter\\_manual\\_deployment](https://github.com/hjwp/book-example/tree/chapter_manual_deployment)

Capítulo 10 – [https://github.com/hjwp/book-example/tree/chapter\\_making\\_deployment\\_production\\_ready](https://github.com/hjwp/book-example/tree/chapter_making_deployment_production_ready)

Capítulo 11 – [https://github.com/hjwp/book-example/tree/chapter\\_automate\\_deployment\\_with\\_fabric](https://github.com/hjwp/book-example/tree/chapter_automate_deployment_with_fabric)

Capítulo 12 – [https://github.com/hjwp/book-example/tree/chapter\\_organising\\_test\\_files](https://github.com/hjwp/book-example/tree/chapter_organising_test_files)

Capítulo 13 – [https://github.com/hjwp/book-example/tree/chapter\\_database\\_layer\\_validation](https://github.com/hjwp/book-example/tree/chapter_database_layer_validation)

Capítulo 14 – [https://github.com/hjwp/book-example/tree/chapter\\_simple\\_form](https://github.com/hjwp/book-example/tree/chapter_simple_form)

Capítulo 15 – [https://github.com/hjwp/book-example/tree/chapter\\_advanced\\_forms](https://github.com/hjwp/book-example/tree/chapter_advanced_forms)

Capítulo 16 – [https://github.com/hjwp/book-example/tree/chapter\\_javascript](https://github.com/hjwp/book-example/tree/chapter_javascript)

Capítulo 17 – [https://github.com/hjwp/book-example/tree/chapter\\_deploying\\_validation](https://github.com/hjwp/book-example/tree/chapter_deploying_validation)

Capítulo 18 – [https://github.com/hjwp/book-example/tree/chapter\\_spiking\\_custom\\_auth](https://github.com/hjwp/book-example/tree/chapter_spiking_custom_auth)

Capítulo 19 – [https://github.com/hjwp/book-example/tree/chapter\\_mocking](https://github.com/hjwp/book-example/tree/chapter_mocking)

Capítulo 20 – [https://github.com/hjwp/book-example/tree/chapter\\_fixtures\\_and\\_wait\\_decorator](https://github.com/hjwp/book-example/tree/chapter_fixtures_and_wait_decorator)

Capítulo 21 – [https://github.com/hjwp/book-example/tree/chapter\\_server\\_side\\_debugging](https://github.com/hjwp/book-example/tree/chapter_server_side_debugging)

Capítulo 22 – <https://github.com/hjwp/book->

	<i>example/tree/chapter_outside_in</i>		
Capítulo	23	–	<a href="https://github.com/hjwp/book-example/tree/chapter_purist_unit_tests">https://github.com/hjwp/book-example/tree/chapter_purist_unit_tests</a>
Capítulo	24	–	<a href="https://github.com/hjwp/book-example/tree/chapter_CI">https://github.com/hjwp/book-example/tree/chapter_CI</a>
Capítulo	25	–	<a href="https://github.com/hjwp/book-example/tree/chapter_page_pattern">https://github.com/hjwp/book-example/tree/chapter_page_pattern</a>
Apêndice	B	–	<a href="https://github.com/hjwp/book-example/tree/appendix_Django_Class-Based_Views">https://github.com/hjwp/book-example/tree/appendix_Django_Class-Based_Views</a>
Apêndice	E	–	<a href="https://github.com/hjwp/book-example/tree/appendix_bdd">https://github.com/hjwp/book-example/tree/appendix_bdd</a>
Apêndice	F	–	<a href="https://github.com/hjwp/book-example/tree/appendix_rest_api">https://github.com/hjwp/book-example/tree/appendix_rest_api</a>
Apêndice	G	–	<a href="https://github.com/hjwp/book-example/tree/appendix_DjangoRestFramework">https://github.com/hjwp/book-example/tree/appendix_DjangoRestFramework</a>

## Usando o Git para verificar o seu progresso

Se você achar que deve desenvolver melhor suas habilidades com o Git, poderá adicionar o meu repositório como um *remoto* (remote):

```
git remote add harry https://github.com/hjwp/book-example.git
git fetch harry
```

Então, para verificar suas diferenças com o *final* do Capítulo 4, execute:

```
git diff harry/chapter_philosophy_and_refactoring
```

O Git é capaz de lidar com vários remotos, de modo que você ainda poderá realizar isso, mesmo que já esteja fazendo push de seu código para o GitHub ou o Bitbucket.

Esteja ciente de que a ordem exata, digamos, dos métodos em uma classe pode ser diferente entre a sua versão e a minha. Isso talvez dificulte a leitura dos diffs.

## Fazendo download de um arquivo ZIP para um capítulo

Se, por qualquer que seja o motivo, você quiser “começar do zero” em um capítulo, ou pular adiante<sup>1</sup>, e/ou se simplesmente não se sentir à vontade com o Git, poderá fazer o download de uma versão de meu código na forma de um arquivo ZIP, a partir de URLs que seguem o padrão a seguir:

- [https://github.com/hjwp/book-example/archive/chapter\\_01.zip](https://github.com/hjwp/book-example/archive/chapter_01.zip)
- [https://github.com/hjwp/book-example/archive/chapter\\_philosophy\\_and\\_refactoring.zip](https://github.com/hjwp/book-example/archive/chapter_philosophy_and_refactoring.zip)

## Não deixe que vire muletas!

Tente não espiar as respostas, a menos que você realmente não saiba o que fazer. Como eu disse no início do último capítulo, depurar erros por conta própria é muito importante e, na vida real, não haverá nenhum “repositório de harry” com o qual conferir e encontrar todas as respostas.

---

<sup>1</sup> Não recomendo pular partes do texto. Não projetei os capítulos para que fossem independentes; cada um deles depende dos capítulos anteriores, portanto isso pode ser mais confuso do que tudo o mais...



# Bibliografia

- [dip] Mark Pilgrim, *Dive Into Python*: <http://www.diveintopython.net/><sup>1</sup>
- [lpthw] Zed A. Shaw, *Learn Python the Hard Way*:  
<http://learnpythonthehardway.org/>
- [iwp] Al Sweigart, *Invent Your Own Computer Games with Python*:  
<http://inventwithpython.com>
- [tddbe] Kent Beck, *Test Driven Development: By Example*, Addison-Wesley<sup>2</sup>
- [refactoring] Martin Fowler, *Refactoring*, Addison-Wesley<sup>3</sup>
- [seceng] Ross Anderson, *Security Engineering, segunda edição*, Addison-Wesley: <http://www.cl.cam.ac.uk/~rja14/book.html>
- [jsgoodparts] Douglas Crockford, *JavaScript: The Good Parts*, O'Reilly<sup>4</sup>
- [twoscoops] Daniel Greenfeld e Audrey Roy, *Two Scoops of Django*,  
<http://twoscoopspress.com/products/two-scoops-of-django-1-6>
- [mockfakestub] Emily Bache, *Mocks, Fakes and Stubs*,  
<https://leanpub.com/mocks-fakes-stubs>
- [GOOSGBT] Steve Freeman e Nat Pryce, *Growing Object-Oriented Software Guided by Tests*, Addison-Wesley<sup>5</sup>

---

<sup>1</sup> N.T.: Edição em português: *Mergulhando no Python* (Altabooks).

<sup>2</sup> N.T.: Edição em português: *TDD: desenvolvimento guiado por testes* (Bookman).

<sup>3</sup> N.T.: Edição em português: *Refatoração* (Bookman).

<sup>4</sup> N.T.: Edição em português: *O melhor do JavaScript* (Altabooks).

<sup>5</sup> N.T.: Edição em português: *Desenvolvimento de software orientado a objetos, guiado por testes* (Altabooks).

# Sobre o autor

Após uma infância idílica brincando com BASIC em computadores franceses de 8 bits como o Thomson T-07, cujas teclas faziam “blip” quando pressionadas, Harry passou alguns anos profundamente infeliz dando consultoria em economia e administração. Logo redescobriu sua verdadeira natureza geek e teve sorte o bastante para acabar com um grupo de fanáticos por XP, trabalhando na pioneira, porém infelizmente extinta, planilha Resolver One. Atualmente trabalha na PythonAnywhere LLP e divulga a boa-nova sobre TDD pelo mundo todo por meio de palestras, workshops e conferências, com toda paixão e entusiasmo de um recém-convertido.

# Colofão

O animal na capa de *TDD com Python* é uma cabra-da-caxemira. Embora todos os caprinos possam produzir uma pelagem interna para caxemira (ou casimira), somente aqueles seletivamente criados para produzir a lã de caxemira em quantidades viáveis do ponto de vista comercial são em geral considerados como “cabras-da-caxemira”. Desse modo, as cabras-da-caxemira pertencem à espécie doméstica de caprinos *Capra hircus*.

O pelo excepcionalmente fino e macio da pelagem interna de uma cabra-da-caxemira cresce junto com uma pelagem externa mais grossa, como parte da lã dupla do caprino. A pelagem interna da lã de caxemira aparece no inverno para suplementar a proteção oferecida pela pelagem externa, chamada de *pelo de guarda*. A qualidade crespada do pelo para a lã de caxemira na parte interna é responsável tanto por suas características de leveza, como de eficácia, quanto ao isolamento.

O nome “caxemira” é derivado da região do Vale de Kashmir no subcontinente indiano, em que o tecido vem sendo fabricado há centenas de anos. Uma população cada vez menor de cabras-da-caxemira no Kashmir moderno levou à interrupção das exportações das fibras de lã de caxemira da região. A maior parte da lã de caxemira atualmente é originária do Afeganistão, do Irã, da Mongólia Exterior, da Índia e, predominantemente, da China.

As cabras-da-caxemira apresentam pelos de cores variadas e com diferentes combinações. Tanto os machos quanto as fêmeas têm chifres, que servem para manter os animais frescos no verão, além de servirem como apoio para um manejo eficiente pelos proprietários dos caprinos nas atividades do campo.

Muitos dos animais nas capas dos livros da O’Reilly estão

ameaçados; todos eles são importantes para o mundo. Para saber mais sobre como ajudar, acesse [animals.oreilly.com](http://animals.oreilly.com).

A imagem da capa foi extraída do livro *Animate Creation* de Wood.

O'REILLY

# Python para Análise de Dados

TRATAMENTO DE DADOS COM  
PANDAS, NUMPY E IPYTHON



novatec

Wes McKinney

# Python para análise de dados

McKinney, Wes

9788575227510

616 páginas

[Compre agora e leia](#)

Obtenha instruções completas para manipular, processar, limpar e extrair informações de conjuntos de dados em Python. Atualizada para Python 3.6, este guia prático está repleto de casos de estudo práticos que mostram como resolver um amplo conjunto de problemas de análise de dados de forma eficiente. Você conhecerá as versões mais recentes do pandas, da NumPy, do IPython e do Jupyter no processo. Escrito por Wes McKinney, criador do projeto Python pandas, este livro contém uma introdução prática e moderna às ferramentas de ciência de dados em Python. É ideal para analistas, para quem Python é uma novidade, e para programadores Python iniciantes nas áreas de ciência de dados e processamento científico. Os arquivos de dados e os materiais relacionados ao livro estão disponíveis no GitHub.

- utilize o shell IPython e o Jupyter Notebook para processamentos exploratórios;
- conheça os recursos básicos e avançados da NumPy (Numerical Python);
- comece a trabalhar com ferramentas de análise de dados da biblioteca pandas;
- utilize ferramentas flexíveis para carregar, limpar, transformar, combinar e reformatar dados;
- crie visualizações informativas com a matplotlib;
- aplique o recurso

groupby do pandas para processar e sintetizar conjuntos de dados; •  
analise e manipule dados de séries temporais regulares e  
irregulares; • aprenda a resolver problemas de análise de dados do  
mundo real com exemplos completos e detalhados.

[Compre agora e leia](#)



O'REILLY®

# Padrões para Kubernetes

Elementos reutilizáveis no design de aplicações  
nativas de nuvem



novatec

Bilgin Ibryam  
Roland Huß

# Padrões para Kubernetes

Ibryam, Bilgin

9788575228159

272 páginas

[Compre agora e leia](#)

O modo como os desenvolvedores projetam, desenvolvem e executam software mudou significativamente com a evolução dos microsserviços e dos contêineres. Essas arquiteturas modernas oferecem novas primitivas distribuídas que exigem um conjunto diferente de práticas, distinto daquele com o qual muitos desenvolvedores, líderes técnicos e arquitetos estão acostumados. Este guia apresenta padrões comuns e reutilizáveis, além de princípios para o design e a implementação de aplicações nativas de nuvem no Kubernetes. Cada padrão inclui uma descrição do problema e uma solução específica no Kubernetes. Todos os padrões acompanham e são demonstrados por exemplos concretos de código. Este livro é ideal para desenvolvedores e arquitetos que já tenham familiaridade com os conceitos básicos do Kubernetes, e que queiram aprender a solucionar desafios comuns no ambiente nativo de nuvem, usando padrões de projeto de uso comprovado. Você conhecerá as seguintes classes de padrões:

- Padrões básicos, que incluem princípios e práticas essenciais para desenvolver aplicações nativas de nuvem com base em contêineres.
- Padrões comportamentais, que exploram conceitos mais

específicos para administrar contêineres e interações com a plataforma. • Padrões estruturais, que ajudam você a organizar contêineres em um Pod para tratar casos de uso específicos. • Padrões de configuração, que oferecem insights sobre como tratar as configurações das aplicações no Kubernetes. • Padrões avançados, que incluem assuntos mais complexos, como operadores e escalabilidade automática (autoscaling).

[Compre agora e leia](#)

# CANDLESTICK

Um método para ampliar lucros na Bolsa de Valores



novatec

Carlos Alberto Debastiani

# Candlestick

Debastiani, Carlos Alberto

9788575225943

200 páginas

[Compre agora e leia](#)

A análise dos gráficos de Candlestick é uma técnica amplamente utilizada pelos operadores de bolsas de valores no mundo inteiro. De origem japonesa, este refinado método avalia o comportamento do mercado, sendo muito eficaz na previsão de mudanças em tendências, o que permite desvendar fatores psicológicos por trás dos gráficos, incrementando a lucratividade dos investimentos.

Candlestick – Um método para ampliar lucros na Bolsa de Valores é uma obra bem estruturada e totalmente ilustrada. A preocupação do autor em utilizar uma linguagem clara e acessível a torna leve e de fácil assimilação, mesmo para leigos. Cada padrão de análise abordado possui um modelo com sua figura clássica, facilitando a identificação. Depois das características, das peculiaridades e dos fatores psicológicos do padrão, é apresentado o gráfico de um caso real aplicado a uma ação negociada na Bovespa. Este livro possui, ainda, um índice resumido dos padrões para pesquisa rápida na utilização cotidiana.

[Compre agora e leia](#)



# AVALIANDO EMPRESAS

# INVESTINDO EM AÇÕES

A APLICAÇÃO PRÁTICA DA  
ANÁLISE FUNDAMENTALISTA NA  
AVALIAÇÃO DE EMPRESAS

novatec

CARLOS ALBERTO DEBASTIANI  
FELIPE AUGUSTO RUSSO

# Avaliando Empresas, Investindo em Ações

Debastiani, Carlos Alberto

9788575225974

224 páginas

[Compre agora e leia](#)

Avaliando Empresas, Investindo em Ações é um livro destinado a investidores que desejam conhecer, em detalhes, os métodos de análise que integram a linha de trabalho da escola fundamentalista, trazendo ao leitor, em linguagem clara e acessível, o conhecimento profundo dos elementos necessários a uma análise criteriosa da saúde financeira das empresas, envolvendo indicadores de balanço e de mercado, análise de liquidez e dos riscos pertinentes a fatores setoriais e conjunturas econômicas nacional e internacional. Por meio de exemplos práticos e ilustrações, os autores exercitam os conceitos teóricos abordados, desde os fundamentos básicos da economia até a formulação de estratégias para investimentos de longo prazo.

[Compre agora e leia](#)

Marcos Abe

MANUAL DE  
**ANÁLISE  
TÉCNICA**

ESSÊNCIA E ESTRATÉGIAS AVANÇADAS

TUDO O QUE UM INVESTIDOR PRECISA SABER PARA  
PROSPERAR NA BOLSA DE VALORES ATÉ EM TEMPOS DE CRISE

novatec



# Manual de Análise Técnica

Abe, Marcos

9788575227022

256 páginas

[Compre agora e leia](#)

Este livro aborda o tema Investimento em Ações de maneira inédita e tem o objetivo de ensinar os investidores a lucrarem nas mais diversas condições do mercado, inclusive em tempos de crise. Ensinará ao leitor que, para ganhar dinheiro, não importa se o mercado está em alta ou em baixa, mas sim saber como operar em cada situação. Com o Manual de Análise Técnica o leitor aprenderá:

- os conceitos clássicos da Análise Técnica de forma diferenciada, de maneira que assimile não só os princípios, mas que desenvolva o raciocínio necessário para utilizar os gráficos como meio de interpretar os movimentos da massa de investidores do mercado;
- identificar oportunidades para lucrar na bolsa de valores, a longo e curto prazo, até mesmo em mercados baixistas; um sistema de investimentos completo com estratégias para abrir, conduzir e fechar operações, de forma que seja possível maximizar lucros e minimizar prejuízos;
- estruturar e proteger operações por meio do gerenciamento de capital. Destina-se a iniciantes na bolsa de valores e investidores que ainda não desenvolveram uma metodologia própria para operar lucrativamente.

[Compre agora e leia](#)