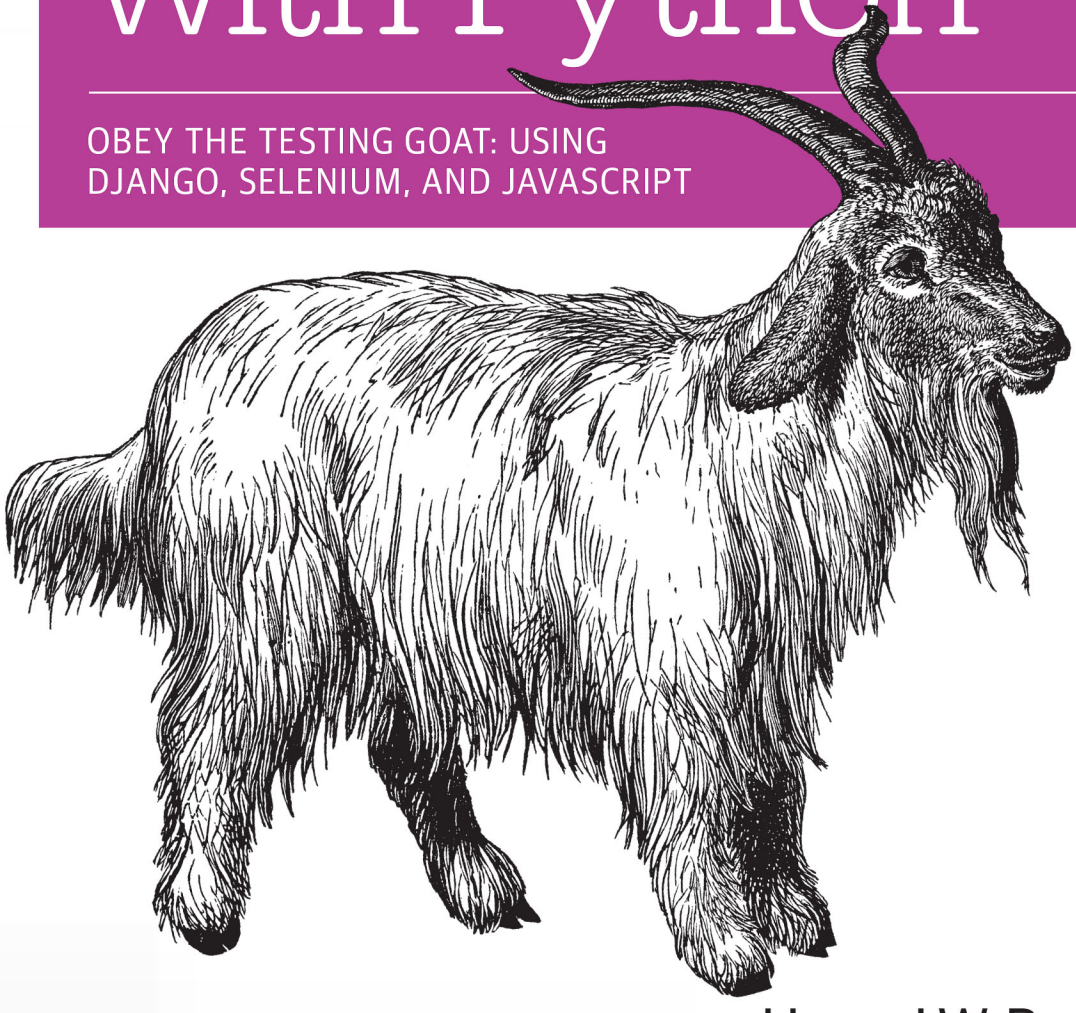
O'REILLY®

Test-Driven Development with Python

OBEY THE TESTING GOAT: USING DJANGO, SELENIUM, AND JAVASCRIPT



Harry J.W. Percival

Praise for *Test-Driven Development with Python*

"In this book, Harry takes us on an adventure of discovery with Python and testing. It's an excellent book, fun to read and full of vital information. It has my highest recommendations for anyone interested in testing with Python, learning Django or wanting to use Selenium.

Testing is essential for developer sanity and it's a notoriously difficult field, full of trade-offs. Harry does a fantastic job of holding our attention whilst exploring real world testing practices."

— Michael Foord
Python Core Developer and Maintainer of unittest

"This book is far more than an introduction to Test Driven Development—it's a complete best-practices crash course, from start to finish, into modern web application development with Python. Every web developer needs this book."

— Kenneth Reitz
Fellow at Python Software Foundation

"Harry's book is what we wish existed when we were learning Django. At a pace that's achievable and yet delightfully challenging, it provides excellent instruction for Django and various test practices. The material on Selenium alone makes the book worth purchasing, but there's so much more!"

— Daniel and Audrey Roy Greenfield
authors of *Two Scoops of Django* (Two Scoops Press)

Test-Driven Development with Python

Harry Percival

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

Test-Driven Development with Python

by Harry Percival

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Meghan Blanchette
Production Editor: Kara Ebrahim
Copyeditor: Charles Roumeliotis
Proofreader: Gillian McGarvey
Indexer: Wendy Catalano
Cover Designer: Randy Comer
Interior Designer: David Futato
Illustrator: Rebecca Demarest

June 2014: First Edition

Revision History for the First Edition:

2014-06-09: First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449364823* for release details.

ISBN: 978-1-449-36482-3

[LSI]

Table of Contents

Preface

This book is my attempt to share with the world the journey I've taken from "hacking" to "software engineering". It's mainly about testing, but there's a lot more to it, as you'll soon see.

I want to thank you for reading it.

If you bought a copy, then I'm very grateful. If you're reading the free online version, then I'm *still* grateful that you've decided it's worth spending some of your time on. Who knows, perhaps once you get to the end, you'll decide it's good enough to buy a real copy for yourself or for a friend.

If you have any comments, questions, or suggestions, I'd love to hear from you. You can reach me directly via *obeythetestinggoat@gmail.com*, or on Twitter @hjwp. You can also check out the website and my blog, and there's a mailing list.

I hope you'll enjoy reading this book as much as I enjoyed writing it.

Why I Wrote a Book About Test-Driven Development

"Who are you, why are you writing this book, and why should I read it?" I hear you ask.

I'm still quite early on in my programming career. They say that in any discipline, you go from apprentice, to journeyman, and eventually, sometimes, on to master. I'd say that I'm—at best—a journeyman programmer. But I was lucky enough, early on in my career, to fall in with a bunch of TDD fanatics, and it made such a big impact on my programming that I'm burning to share it with everyone. You might say I have the enthusiasm of a recent convert, and the learning experience is still a recent memory for me, so I hope I can still empathise with beginners.

When I first learned Python (from Mark Pilgrim's excellent *Dive Into Python*), I came across the concept of TDD, and thought "Yes. I can definitely see the sense in that". Perhaps you had a similar reaction when you first heard about TDD? It sounds like a

really sensible approach, a really good habit to get into—like regularly flossing your teeth or something.

Then came my first big project, and you can guess what happened—there was a client, there were deadlines, there was lots to do, and any good intentions about TDD went straight out of the window.

And, actually, it was fine. I was fine.

At first.

At first I knew I didn't really need TDD because it was a small website, and I could easily test whether things worked by just manually checking it out. Click this link *here*, choose that drop-down item *there*, and *this* should happen. Easy. This whole writing tests thing sounded like it would have taken *ages*, and besides, I fancied myself, from the full height of my three weeks of adult coding experience, as being a pretty good programmer. I could handle it. Easy.

Then came the fearful goddess Complexity. She soon showed me the limits of my experience.

The project grew. Parts of the system started to depend on other parts. I did my best to follow good principles like DRY (Don't Repeat Yourself), but that just led to some pretty dangerous territory. Soon I was playing with multiple inheritance. Class hierarchies 8 levels deep. `eval` statements.

I became scared of making changes to my code. I was no longer sure what depended on what, and what might happen if I changed this code *over here*, oh gosh, I think that bit over there inherits from it—no, it doesn't, it's overriden. Oh, but it depends on that class variable. Right, well, as long as I override the override it should be fine. I'll just check —but checking was getting much harder. There were lots of sections to the site now, and clicking through them all manually was starting to get impractical. Better to leave well enough alone, forget refactoring, just make do.

Soon I had a hideous, ugly mess of code. New development became painful.

Not too long after this, I was lucky enough to get a job with a company called Resolver Systems (now PythonAnywhere), where Extreme Programming (XP) was the norm. They introduced me to rigorous TDD.

Although my previous experience had certainly opened my mind to the possible benefits of automated testing, I still dragged my feet at every stage. "I mean, testing in general might be a good idea, but *really*? All these tests? Some of them seem like a total waste of time … What? Functional tests as *well* as unit tests? Come on, that's overdoing it! And this TDD test/minimal-code-change/test cycle? This is just silly! We don't need all these baby steps! Come on, we can see what the right answer is, why don't we just skip to the end?"

Believe me, I second-guessed every rule, I suggested every shortcut, I demanded justifications for every seemingly pointless aspect of TDD, and I came out seeing the wisdom of it all. I've lost count of the number of times I've thought "Thanks, tests", as a functional test uncovers a regression we would never have predicted, or a unit test saves me from making a really silly logic error. Psychologically, it's made development a much less stressful process. It produces code that's a pleasure to work with.

So, let me tell you *all* about it!

Aims of This Book

My main aim is to impart a methodology—a way of doing web development, which I think makes for better web apps and happier developers. There's not much point in a book that just covers material you could find by googling, so this book isn't a guide to Python syntax, or a tutorial on web development *per se*. Instead, I hope to teach you how to use TDD to get more reliably to our shared, holy goal: *clean code that works.*

With that said: I will constantly refer to a real practical example, by building a web app from scratch using tools like Django, Selenium, jQuery, and Mock. I'm not assuming any prior knowledge of any of these, so you should come out of the other end of this book with a decent introduction to those tools, as well as the discipline of TDD.

In Extreme Programming we always pair-program, so I've imagined writing this book as if I was pairing with my previous self, having to explain how the tools work and answer questions about why we code in this particular way. So, if I ever take a bit of a patronising tone, it's because I'm not all that smart, and I have to be very patient with myself. And if I ever sound defensive, it's because I'm the kind of annoying person that systematically disagrees with whatever anyone else says, so sometimes it takes a lot of justifying to convince myself of anything.

Outline

I've split this book into three parts.

Part I (Chapters 1–6): The basics
: Dives straight into building a simple web app using TDD. We start by writing a functional test (with Selenium), then we go through the basics of Django—models, views, templates—with rigorous unit testing at every stage. I also introduce the Testing Goat.

Part II (Chapters 7–14): Web development essentials
: Covers some of the trickier but unavoidable aspects of web development, and shows how testing can help us with them: static files, deployment to production, form data validation, database migrations, and the dreaded JavaScript.

Part III (Chapters 15–20): More advanced topics
: Mocking, integrating a third-party authentication system, Ajax, test fixtures, Outside-In TDD, and Continuous Integration (CI).

On to a little housekeeping…

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

Occasionally I will use the symbol:

```
[...]
```

to signify that some of the content has been skipped, to shorten long bits of output, or to skip down to a relevant bit.

This element signifies a tip or suggestion.

This element signifies a general note or aside.

This element indicates a warning or caution.

Using Code Examples

Code examples are available at *https://github.com/hjwp/book-example/*; you'll find branches for each chapter there (eg, *https://github.com/hjwp/book-example/tree/chapter_03*). You'll also find some suggestions on ways of working with this repository at the end of each chapter.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Test-Driven Development with Python* by Harry Percival (O'Reilly). Copyright 2014 Harry Percival, 978-1-449-36482-3."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

Contacting O'Reilly

If you'd like to get in touch with my beloved publisher with any questions about this book, contact details follow:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

You can also send email to *bookquestions@oreilly.com*.

You can find errata, examples, and additional information at *http://bit.ly/test-driven-python*.

For more information about books, courses, conferences, and news, see O'Reilly's website at *http://www.oreilly.com*.

Facebook: *http://facebook.com/oreilly*

Twitter: *http://twitter.com/oreillymedia*

YouTube: *http://www.youtube.com/oreillymedia*

Prerequisites and Assumptions

Here's an outline of what I'm assuming about you and what you already know, as well as what software you'll need ready and installed on your computer.

Python 3 and Programming

I've written the book with beginners in mind, but if you're new to programming, I'm assuming that you've already learned the basics of Python. So if you haven't already, do run through a Python beginner's tutorial or get an introductory book like *Dive Into Python* or *Learn Python the Hard Way*, or, just for fun, *Invent Your Own Computer Games with Python*, all of which are excellent introductions.

If you're an experienced programmer but new to Python, you should get along just fine. Python is joyously simple to understand.

I'm using *Python 3* for this book. When I wrote it in 2013–14, Python 3 had been around for several years, and the world was just about on the tipping point at which it was the preferred choice. You should be able to follow this book on Mac, Windows, or Linux. Detailed installation instructions for each OS follow.

This book was tested against Python 3.3 and Python 3.4. If you're on 3.2 for any reason, you may find minor differences, so you're best off upgrading if you can.

I wouldn't recommend trying to use Python 2, as the differences are more substantial. You'll still be able to carry across all the lessons you learn in this book if your next project happens to be in Python 2. But spending time figuring out whether the reason your program output looks different from mine is because of Python 2, or because you made an actual mistake, won't be time spent productively.

If you are thinking of using PythonAnywhere (the PaaS startup I work for), rather than a locally installed Python, you should go and take a quick look at Appendix A before you get started.

In any case, I expect you to have access to Python, to know how to launch it from a command line (usually with the command **`python3`**), and to know how to edit a Python file and run it. Again, have a look at the three books I recommended previously if you're in any doubt.

If you already have Python 2 installed, and you're worried that installing Python 3 will break it in some way, don't! Python 3 and 2 can coexist peacefully on the same system, and they each store their packages in totally different locations. You just need to make sure that you have one command to launch Python 3 (`python3`), and another to launch Python 2 (usually, just `python`). Similarly, when we install pip for Python 3, we just make sure that its command (usually `pip3`) is identifiably different from the Python 2 pip.

How HTML Works

I'm also assuming you have a basic grasp of how the web works—what HTML is, what a POST request is, etc. If you're not sure about those, you'll need to find a basic HTML tutorial—there are a few at *http://www.webplatform.org/*. If you can figure out how to create an HTML page on your PC and look at it in your browser, and understand what a form is and how it might work, then you're probably OK.

JavaScript

There's a little bit of JavaScript in the second half of the book. If you don't know JavaScript, don't worry about it until then, and if you find yourself a little confused, I'll recommend a couple of guides at that point.

Required Software Installations

Aside from Python, you'll need:

The Firefox web browser
: A quick Google search will get you an installer for whichever platform you're on. Selenium can actually drive any of the major browsers, but Firefox is the easiest to use as an example because it's reliably cross-platform and, as a bonus, is less sold out to corporate interests.

The Git version control system
: This is available for any platform, at *http://git-scm.com/*.

The pip Python package management tool

This comes bundled with Python 3.4 (it didn't always used to, this is a big hooray). To make sure we're using the Python3 version of pip, I'll always use `pip3` as the executable in my command-line examples. Depending on your platform, it may be `pip-3.4` or `pip-3.3`. Have a look at the detailed notes for each operating system for more info.

Windows Notes

Windows users can sometimes feel a little neglected, since OS X and Linux make it easy to forget there's a world outside the Unix paradigm. Backslashes as directory separators? Drive letters? What? Still, it is absolutely possible to follow along with this book on Windows. Here are a few tips:

1. When you install Git for Windows, make sure you choose *"Run Git and included Unix tools from the Windows command prompt"*. You'll then get access to a program called "Git Bash". Use this as your main command prompt and you'll get all the useful GNU command-line tools like `ls`, `touch`, and `grep`, plus forward-slash directory separators.
2. When you install Python 3, make sure you tick the option that says "add python.exe to Path" as in Figure P-1, to make sure you can run Python from the command line.

Figure P-1. Add python to the system path from the installer

3. On Windows, Python 3's executable is called `python.exe`, which is exactly the same as Python 2. To avoid any confusion, create a symlink in the Git Bash binaries folder, like this:

   ```
   ln -s /c/Python34/python.exe /bin/python3.exe
   ```

   You may need to right-click Git-Bash and choose "Run as an administrator" for that command to work. Note also that the symlink will only work in Git Bash, not in the regular DOS command prompt.

4. Python 3.4 comes with pip, the package management tool. You can check it's installed by doing a `which pip3` from a command line, and it should show you `/c/Python34/Scripts/pip3`.

   If, for whatever reason, you're stuck with Python 3.3 and you don't have `pip3`, check *http://www.pip-installer.org/* for installation instructions. At the time of writing, this involved downloading a file and then executing it with `python3 get-pip.py`. *Make sure you use `python3` when you run the setup script.*

The test for all this is that you should be able to go to a Git-Bash command prompt and just run `python3` or `pip3` from any folder.

MacOS Notes

MacOS is a bit more sane than Windows, although getting `pip3` installed was still fairly challenging up until recently. With the arrival of 3.4, things are now quite straightforward:

- Python 3.4 should install without a fuss from its downloadable installer. It will automatically install pip, too.
- Git's installer should also "just work".

Similarly to Windows, the test for all this is that you should be able to open a terminal and just run `git`, `python3`, or `pip3` from anywhere. If you run into any trouble, the search terms "system path" and "command not found" should provide good troubleshooting resources.

You might also want to check out Homebrew. It used to be the only reliable way of installing lots of Unixy tools (including Python 3) on a Mac. Although the Python installer is now fine, you may find it useful in future. It does require you to download all 1.1 GB of Xcode, but that also gives you a C compiler, which is a useful side effect.

Git's Default Editor, and Other Basic Git Config

I'll provide step-by-step instructions for Git, but it may be a good idea to get a bit of configuration done now. For example, when you do your first commit, by default *vi* will pop up, at which point you may have no idea what to do with it. Well, much as vi has two modes, you then have two choices. One is to learn some minimal vi commands *(press the i key to go into insert mode, type your text, press `<Esc>` to go back to normal mode, then write the file and quit with `:wq<Enter>`)*. You'll then have joined the great fraternity of people who know this ancient, revered text editor.

Or you can point-blank refuse to be involved in such a ridiculous throwback to the 1970s, and configure Git to use an editor of your choice. Quit vi using `<Esc>` followed by `:q!`, then change your Git default editor. See the Git documentation on basic Git configuration.

Required Python Modules

Once you have *pip* installed, it's trivial to install new Python modules. We'll install some as we go, but there are a couple we'll need right from the beginning, so you should install them right away:

- *Django 1.7*, **`sudo pip3 install django==1.7`** (omit the `sudo` on Windows). This is our web framework. You should make sure you have version 1.7 installed and that you can access the `django-admin.py` executable from a command line. The Django documentation has some installation instructions if you need help.

As of May 2014, Django 1.7 was still in beta. If the above command doesn't work, use **`sudo pip3 install https://github.com/django/django/archive/stable/1.7.x.zip`**

- *Selenium*, **`sudo pip3 install --upgrade selenium`** (omit the `sudo` on Windows), a browser automation tool that we'll use to drive what are called functional tests.

Make sure you have the absolute latest version installed. Selenium is engaged in a permanent arms race with the major browsers, trying to keep up with the latest features. If you ever find Selenium misbehaving for some reason, the answer is often that it's a new version of Firefox and you need to upgrade to the latest Selenium …

Unless you're absolutely sure you know what you're doing, *don't* use a `virtualenv`. We'll start using one later in the book, in Chapter 8.

A Note on IDEs

If you've come from the world of Java or .NET, you may be keen to use an IDE for your Python coding. They have all sorts of useful tools, including VCS integration, and there are some excellent ones out there for Python. I used one myself when I was starting out, and I found it very useful for my first couple of projects.

Can I suggest (and it's only a suggestion) that you *don't* use an IDE, at least for the duration of this tutorial? IDEs are much less necessary in the Python world, and I've written this whole book with the assumption that you're just using a basic text editor and a command line. Sometimes, that's all you have—when you're working on a server for example—so it's always worth learning how to use the basic tools first and understanding how they work. It'll be something you always have, even if you decide to go back to your IDE and all its helpful tools, after you've finished this book.

Did these instructions not work for you? Or have you got better ones? Get in touch: *obeythetestinggoat@gmail.com*!

Acknowledgments

Lots of people to thank, without whom this book would never have happened, and/or would have been even worse than it is.

Thanks first to "Greg" at $OTHER_PUBLISHER, who was the first person to encourage me to believe it really could be done. Even though your employers turned out to have overly regressive views on copyright, I'm forever grateful that you believed in me.

Thanks to Michael Foord, another ex-employee of Resolver Systems, for providing the original inspiration by writing a book himself, and thanks for his ongoing support for the project. Thanks also to my boss Giles Thomas, for foolishly allowing another one of his employees to write a book (although I believe he's now changed the standard employment contract to say "no books"). Thanks also for your ongoing wisdom and for setting me off on the testing path.

Thanks to my other colleagues, Glenn Jones and Hansel Dunlop, for being invaluable sounding boards, and your patience with my one-track record conversation over the last year.

Thanks to my wife Clementine, and to both my families, without whose support and patience I would never have made it. I apologise for all the time spent with nose in computer on what should have been memorable family occasions. I had no idea when I set out what the book would do to my life ("write it in my spare time you say? That sounds reasonable…"). I couldn't have done it without you.

Thanks to my tech reviewers, Jonathan Hartley, Nicholas Tollervey, and Emily Bache, for your encouragements and invaluable feedback. Especially Emily, who actually conscientiously read every single chapter. Partial credit to Nick and Jon, but that should still be read as eternal gratitude. Having y'all around made the whole thing less of a lonely endeavour. Without all of you the book would have been little more than the nonsensical ramblings of an idiot.

Thanks to everyone else who's given up some of their time to give some feedback on the book, out of nothing more than the goodness of their heart: Gary Bernhardt, Mark

Lavin, Matt O'Donnell, Michael Foord, Hynek Schlawack, Russell Keith-Magee, Andrew Godwin, and Kenneth Reitz. Thanks for being much smarter than I am, and for preventing me from saying several stupid things. Naturally, there are still plenty of stupid things left in the book, for which y'all can absolutely not be held responsible.

Thanks to my editor Meghan Blanchette, for being a very friendly and likeable slave driver, for keeping the book on track, both in terms of timescales and by restraining my sillier ideas. Thanks to all the others at O'Reilly for your help, including Sarah Schneider, Kara Ebrahim, and Dan Fauxsmith for letting me keep British English. Thanks to Charles Roumeliotis for your help with style and grammar. We may never see eye-to-eye on the merits of Chichago School quotation/punctuation rules, but I sure am glad you were around. And thanks to the design department for giving us a goat for the cover!

And thanks most especially to all my Early Release readers, for all your help picking out typos, for your feedback and suggestions, for all the ways in which you helped to smooth out the learning curve in the book, and most of all for your kind words of encouragement and support that kept me going. Thank you 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, Sorcha 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, and many, many more. If I've missed your name, you have an absolute right to be aggrieved; I am incredibly grateful to you too, so write to me and I will try and make it up to you in any way I can.

And finally thanks to you, the latest reader, for deciding to check out the book! I hope you enjoy it.

PART I
The Basics of TDD and Django

In this first part, I'm going to introduce the basics of *Test-Driven Development* (TDD). We'll build a real web application from scratch, writing tests first at every stage.

We'll cover functional testing with Selenium, as well as unit testing, and see the difference between the two. I'll introduce the TDD workflow, what I call the unit-test/code cycle. We'll also do some refactoring, and see how that fits with TDD. Since it's absolutely essential to serious software engineering, I'll also be using a version control system (Git). We'll discuss how and when to do commits and integrate them with the TDD and web development workflow.

We'll be using Django, the Python world's most popular web framework (probably). I've tried to introduce the Django concepts slowly and one at a time, and provide lots of links to further reading. If you're a total beginner to Django, I thoroughly recommend taking the time to read them. If you find yourself feeling a bit lost, take a couple of hours to go through the official Django tutorial, and then come back to the book.

You'll also get to meet the Testing Goat…

Be Careful with Copy and Paste

If you're working from a digital version of the book, it's natural to want to copy and paste code listings from the book as you're working through it. It's much better if you don't: typing things in by hand gets them into your muscle memory, and just feels much more real. You also inevitably make the occasional typo, and debugging them is an important thing to learn.

Quite apart from that, you'll find that the quirks of the PDF format mean that weird stuff often happens when you try and copy/paste from it…

CHAPTER 1

Getting Django Set Up Using a Functional Test

TDD isn't something that comes naturally. It's a discipline, like a martial art, and just like in a Kung-Fu movie, you need a bad-tempered and unreasonable master to force you to learn the discipline. Ours is the Testing Goat.

Obey the Testing Goat! Do Nothing Until You Have a Test

The Testing Goat is the unofficial mascot of TDD in the Python testing community. It probably means different things to different people, but, to me, the Testing Goat is a voice inside my head that keeps me on the True Path of Testing—like one of those little angels or demons that pop up above your shoulder in the cartoons, but with a very niche set of concerns. I hope, with this book, to install the Testing Goat inside your head too.

We've decided to build a website, even if we're not quite sure what it's going to do yet. Normally the first step in web development is getting your web framework installed and configured. *Download this, install that, configure the other, run the script …* but TDD requires a different mindset. When you're doing TDD, you always have the Testing Goat inside you — single-minded as goats are—bleating "Test first, test first!"

In TDD the first step is always the same: *write a test*.

First we write the test, *then* we run it and check that it fails as expected. *Only then* do we go ahead and build some of our app. Repeat that to yourself in a goat-like voice. I know I do.

Another thing about goats is that they take one step at a time. That's why they seldom fall off mountains, see, no matter how steep they are. As you can see in Figure 1-1.

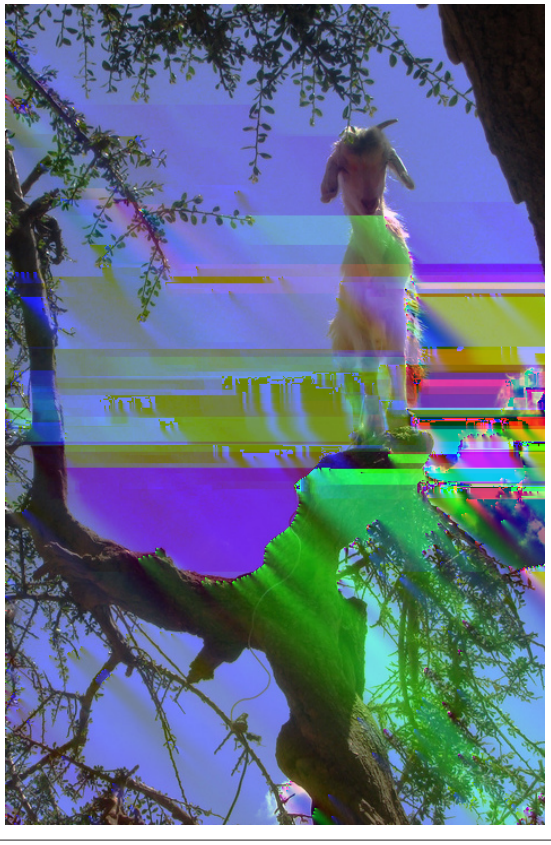


Figure 1-1. Goats are more agile than you think (source: Caitlin Stewart, on Flickr)

We'll proceed with nice small steps; we're going to use *Django*, which is a popular Python web framework, to build our app.

The first thing we want to do is check that we've got Django installed, and that it's ready for us to work with. The *way* we'll check is by confirming that we can spin up Django's development server and actually see it serving up a web page, in our web browser, on our local PC. We'll use the *Selenium* browser automation tool for this.

Create a new Python file called *functional_tests.py*, wherever you want to keep the code for your project, and enter the following code. If you feel like making a few little goat noises as you do it, it may help:

functional_tests.py.

```
from selenium import webdriver

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

assert 'Django' in browser.title
```

Adieu to Roman Numerals!

So many introductions to TDD use Roman numerals as an example that it's a running joke—I even started writing one myself. If you're curious, you can find it on my GitHub page.

Roman numerals, as an example, are both good and bad. It's a nice "toy" problem, reasonably limited in scope, and you can explain TDD quite well with it.

The problem is that it can be hard to relate to the real world. That's why I've decided to use building a real web app, starting from nothing, as my example. Although it's a simple web app, my hope is that it will be easier for you to carry across to your next real project.

That's our first *functional test* (FT); I'll talk more about what I mean by functional tests, and how they contrast with unit tests. For now, it's enough to assure ourselves that we understand what it's doing:

- Starting a Selenium *webdriver* to pop up a real Firefox browser window
- Using it to open up a web page which we're expecting to be served from the local PC
- Checking (making a test assertion) that the page has the word "Django" in its title

Let's try running it:

```
$ python3 functional_tests.py
Traceback (most recent call last):
  File "functional_tests.py", line 6, in <module>
    assert 'Django' in browser.title
AssertionError
```

You should see a browser window pop up and try and open *localhost:8000*, and then the Python error message should appear. And then, you will probably be irritated at the fact that it left a Firefox window lying around your desktop for you to tidy up. We'll fix that later!

If, instead, you see an error trying to import Selenium, you might need to go back and have another look at the Prerequisites and Assumptions chapter.

For now though, we have a *failing test*, so that means we're allowed to start building our app.

Getting Django Up and Running

Since you've definitely read Prerequisites and Assumptions by now, you've already got Django installed. The first step in getting Django up and running is to create a *project*, which will be the main container for our site. Django provides a little command-line tool for this:

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

That will create a folder called *superlists*, and a set of files and subfolders inside it:

```
.
├── functional_tests.py
└── superlists
    ├── manage.py
    └── superlists
        ├── __init__.py
        ├── settings.py
        ├── urls.py
        └── wsgi.py
```

Yes, there's a folder called *superlists* inside a folder called *superlists*. It's a bit confusing, but it's just one of those things; there are good reasons when you look back at the history of Django. For now, the important thing to know is that the *superlists/superlists* folder is for stuff that applies to the whole project—like *settings.py* for example, which is used to store global configuration information for the site.

You'll also have noticed *manage.py*. That's Django's Swiss Army knife, and one of the things it can do is run a development server. Let's try that now. Do a **`cd superlists`** to go into the top-level *superlists* folder (we'll work from this folder a lot) and then run:

```
$ python3 manage.py runserver
Validating models...

0 errors found
Django version 1.7, using settings 'superlists.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Leave that running, and open another command shell. In that, we can try running our test again (from the folder we started in):

```
$ python3 functional_tests.py
$
```

Not much action on the command line, but you should notice two things: firstly, there was no ugly `AssertionError` and secondly, the Firefox window that Selenium popped up had a different-looking page on it.

Well, it may not look like much, but that was our first ever passing test! Hooray!

If it all feels a bit too much like magic, like it wasn't quite real, why not go and take a look at the dev server manually, by opening a web browser yourself and visiting *http://localhost:8000*? You should see something like Figure 1-2.

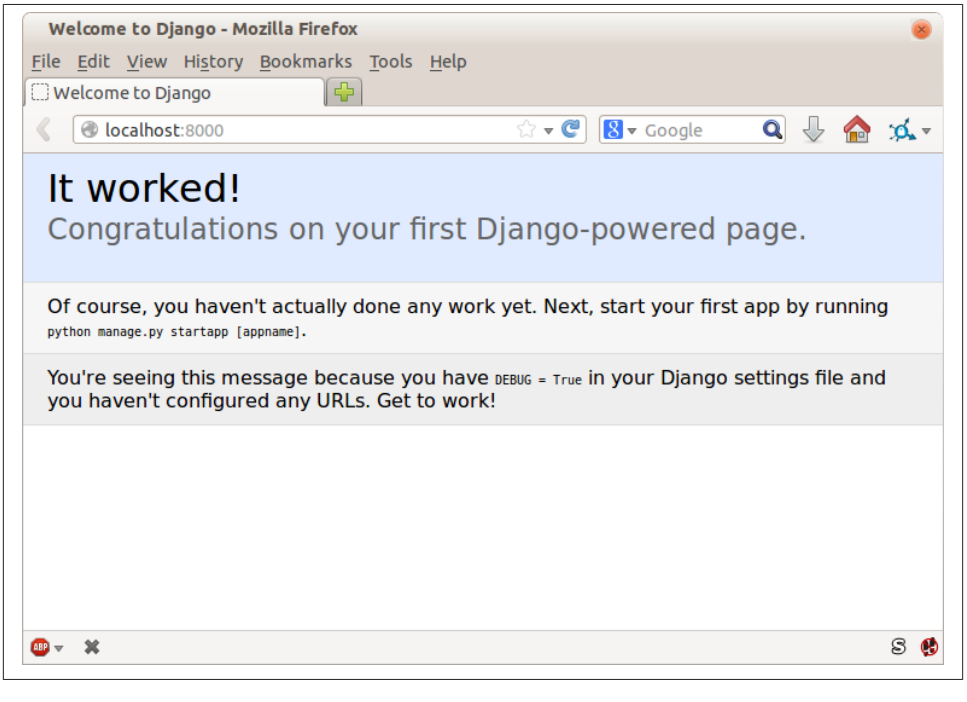


Figure 1-2. It worked!

You can quit the development server now if you like, back in the original shell, using Ctrl-C.

Starting a Git Repository

There's one last thing to do before we finish the chapter: start to commit our work to a *version control system* (VCS). If you're an experienced programmer you don't need to hear me preaching about version control, but if you're new to it please believe me when I say that VCS is a must-have. As soon as your project gets to be more than a few weeks old and a few lines of code, having a tool available to look back over old versions of code, revert changes, explore new ideas safely, even just as a backup … boy. TDD goes hand in hand with version control, so I want to make sure I impart how it fits into the workflow.

So, our first commit! If anything it's a bit late, shame on us. We're using *Git* as our VCS, 'cos it's the best.

Let's start by moving *functional_tests.py* into the *superlists* folder, and doing the `git init` to start the repository:

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

From this point onwards, the top-level *superlists* folder will be our working directory. Whenever I show a command to type in, it will assume we're in this directory. Similarly, if I mention a path to a file, it will be relative to this top-level directory. So *superlists/settings.py* means the *settings.py* inside the second-level *superlists*. Clear as mud? If in doubt, look for *manage.py*; you want to be in the same directory as *manage.py*.

Now let's add the files we want to commit—which is everything really!

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

`db.sqlite3` is a database file. We don't want to have that in version control, so we add it to a special file called *.gitignore* which, um, tells Git what to ignore:

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

Next we can add the rest of the contents of the current folder, ".":

```
$ 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-34.pyc
        new file:   superlists/__pycache__/settings.cpython-34.pyc
        new file:   superlists/__pycache__/urls.cpython-34.pyc
        new file:   superlists/__pycache__/wsgi.cpython-34.pyc
        new file:   superlists/settings.py
        new file:   superlists/urls.py
        new file:   superlists/wsgi.py
```

Darn! We've got a bunch of *.pyc* files in there; it's pointless to commit those. Let's remove them from Git and add them to *.gitignore* too:

```
$ git rm -r --cached superlists/__pycache__
rm 'superlists/__pycache__/__init__.cpython-34.pyc'
rm 'superlists/__pycache__/settings.cpython-34.pyc'
rm 'superlists/__pycache__/urls.cpython-34.pyc'
rm 'superlists/__pycache__/wsgi.cpython-34.pyc'
$ echo "__pycache__" >> .gitignore
$ echo "*.pyc" >> .gitignore
```

Now let's see where we are … (You'll see I'm using `git status` a lot—so much so that I often alias it to `git st` … I'm not telling you how to do that though; I leave you to discover the secrets of Git aliases on your own!):

```
$ 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
```

Looking good, we're ready to do our first commit!

```
$ git commit
```

When you type `git commit`, it will pop up an editor window for you to write your commit message in. Mine looked like Figure 1-3.[1]

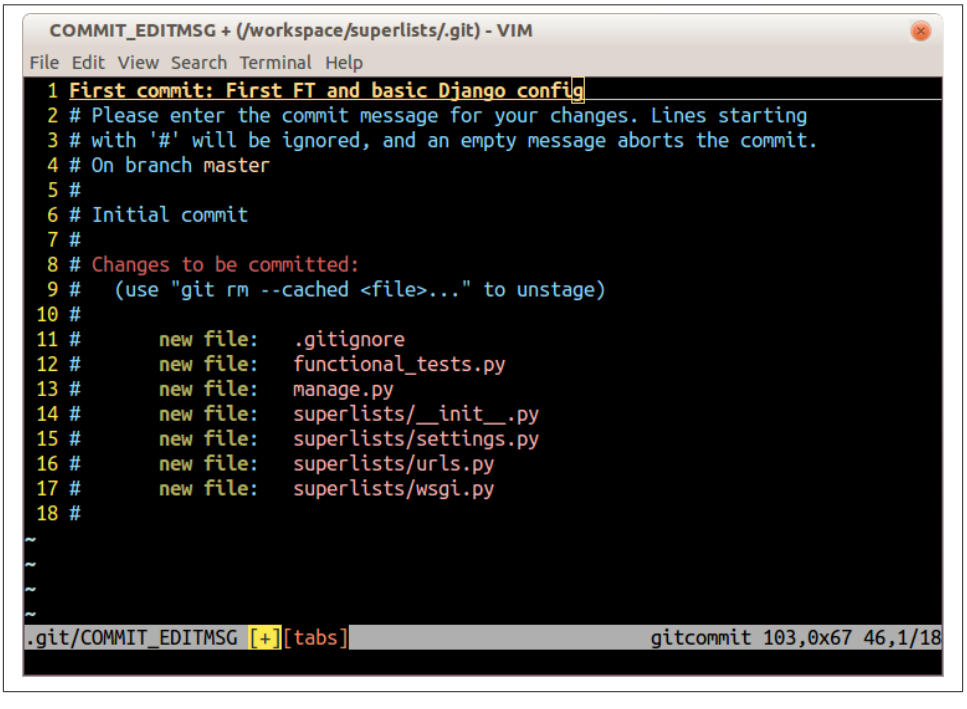


Figure 1-3. First Git commit

If you want to really go to town on Git, this is the time to also learn about how to push your work to a cloud-based VCS hosting service, like GitHub or BitBucket. They'll be useful if you think you want to follow along with this book on different PCs. I leave it to you to find out how they work; they have excellent documentation. Alternatively, you can wait until Chapter 8 when we'll be using one for deployment.

1. Did vi pop up and you had no idea what to do? Or did you see a message about account identity and `git config --global user.username`? Go and take another look at Prerequisites and Assumptions; there are some brief instructions.

That's it for the VCS lecture. Congratulations! You've written a functional test using Selenium, and you've gotten Django installed and running, in a certifiable, test-first, goat-approved TDD way. Give yourself a well-deserved pat on the back before moving on to Chapter 2.

CHAPTER 2

Extending Our Functional Test Using the unittest Module

Let's adapt our test, which currently checks for the default Django "it worked" page, and check instead for some of the things we want to see on the real front page of our site.

Time to reveal what kind of web app we're building: a to-do lists site! In doing so we're very much following fashion: a few years ago all web tutorials were about building a blog. Then it was forums and polls; nowadays it's all to-do lists.

The reason is that a to-do list is a really nice example. At its most basic it is very simple indeed—just a list of text strings—so it's easy to get a "minimum viable" list app up and running. But it can be extended in all sorts of ways—different persistence models, adding deadlines, reminders, sharing with other users, and improving the client-side UI. There's no reason to be limited to just "to-do" lists either; they could be any kind of lists. But the point is that it should allow me to demonstrate all of the main aspects of web programming, and how you apply TDD to them.

Using a Functional Test to Scope Out a Minimum Viable App

Tests that use Selenium let us drive a real web browser, so they really let us see how the application *functions* from the user's point of view. That's why they're called *functional tests*.

This means that an FT can be a sort of specification for your application. It tends to track what you might call a *User Story*, and follows how the user might work with a particular feature and how the app should respond to them.

Terminology: Functional Test == Acceptance Test == End-to-End Test

What I call functional tests, some people prefer to call *acceptance tests*, or *end-to-end tests*. The main point is that these kinds of tests look at how the whole application functions, from the outside. Another term is *black box test*, because the test doesn't know anything about the internals of the system under test.

FTs should have a human-readable story that we can follow. We make it explicit using comments that accompany the test code. When creating a new FT, we can write the comments first, to capture the key points of the User Story. Being human-readable, you could even share them with nonprogrammers, as a way of discussing the requirements and features of your app.

TDD and agile software development methodologies often go together, and one of the things we often talk about is the minimum viable app; what is the simplest thing we can build that is still useful? Let's start by building that, so that we can test the water as quickly as possible.

A minimum viable to-do list really only needs to let the user enter some to-do items, and remember them for their next visit.

Open up *functional_tests.py* and write a story a bit like this one:

functional_tests.py.

```
from selenium import webdriver

browser = webdriver.Firefox()

# Edith has heard about a cool new online to-do app. She goes
# to check out its homepage
browser.get('http://localhost:8000')

# She notices the page title and header mention to-do lists
assert 'To-Do' in browser.title

# She is invited to enter a to-do item straight away

# She types "Buy peacock feathers" into a text box (Edith's hobby
# is tying fly-fishing lures)

# When she hits enter, the page updates, and now the page lists
# "1: Buy peacock feathers" as an item in a to-do list

# There is still a text box inviting her to add another item. She
# enters "Use peacock feathers to make a fly" (Edith is very methodical)

# The page updates again, and now shows both items on her list
```

```
    # Edith wonders whether the site will remember her list. Then she sees
    # that the site has generated a unique URL for her -- there is some
    # explanatory text to that effect.

    # She visits that URL - her to-do list is still there.

    # Satisfied, she goes back to sleep

    browser.quit()
```

We Have a Word for Comments…

When I first started at Resolver, I used to virtuously pepper my code with nice descriptive comments. My colleagues said to me: "Harry, we have a word for comments. We call them lies." I was shocked! But I learned in school that comments are good practice?

They were exaggerating for effect. There is definitely a place for comments that add context and intention. But their point was that it's pointless to write a comment that just repeats what you're doing with the code:

```
    # increment wibble by 1
    wibble += 1
```

Not only is it pointless, there's a danger that you forget to update the comments when you update the code, and they end up being misleading. The ideal is to strive to make your code so readable, to use such good variable names and function names, and to structure it so well that you no longer need any comments to explain *what* the code is doing. Just a few here and there to explain *why*.

There are other places where comments are very useful. We'll see that Django uses them a lot in the files it generates for us to use as a way of suggesting helpful bits of its API. And, of course, we use comments to explain the User Story in our functional tests—by forcing us to make a coherent story out of the test, it makes sure we're always testing from the point of view of the user.

There is more fun to be had in this area, things like *Behaviour Driven Development* and testing DSLs, but they're topics for other books.

You'll notice that, apart from writing the test out as comments, I've updated the `assert` to look for the word "To-Do" instead of "Django". That means we expect the test to fail now. Let's try running it

First, start up the server:

```
$ python3 manage.py runserver
```

And then, in another shell, run the tests:

```
$ python3 functional_tests.py
Traceback (most recent call last):
```

```
  File "functional_tests.py", line 10, in <module>
    assert 'To-Do' in browser.title
AssertionError
```

That's what we call an *expected fail*, which is actually good news - not quite as good as a test that passes, but at least it's failing for the right reason; we can have some confidence we've written the test correctly.

The Python Standard Library's unittest Module

There are a couple of little annoyances we should probably deal with. Firstly, the message "AssertionError" isn't very helpful—it would be nice if the test told us what it actually found as the browser title. Also, it's left a Firefox window hanging around the desktop, it would be nice if this would clear up for us automatically.

One option would be to use the second parameter to the `assert` keyword, something like:

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

And we could also use a `try/finally` to clean up the old Firefox window. But these sorts of problems are quite common in testing, and there are some ready-made solutions for us in the standard library's `unittest` module. Let's use that! In *functional_tests.py*:

functional_tests.py.

```
from selenium import webdriver
import unittest

class NewVisitorTest(unittest.TestCase):  #❶

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

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

    def test_can_start_a_list_and_retrieve_it_later(self):  #❹
        # Edith has heard about a cool new online to-do app. She goes
        # to check out its homepage
        self.browser.get('http://localhost:8000')

        # She notices the page title and header mention to-do lists
        self.assertIn('To-Do', self.browser.title)  #❺
        self.fail('Finish the test!')  #❻

        # She is invited to enter a to-do item straight away
        [...rest of comments as before]

if __name__ == '__main__':  #❼
    unittest.main(warnings='ignore')  #❽
```

You'll probably notice a few things here:

❶ Tests are organised into classes, which inherit from `unittest.TestCase`.

❹ The main body of the test is in a method called `test_can_start_a_list_and_retrieve_it_later`. Any method whose name starts with `test_` is a test method, and will be run by the test runner. You can have more than one `test_` method per class. Nice descriptive names for our test methods are a good idea too.

❷ ❸ `setUp` and `tearDown` are special methods which get run before and after each test. I'm using them to start and stop our browser—note that they're a bit like a `try/except`, in that `tearDown` will run even if there's an error during the test itself.[1] No more Firefox windows left lying around!

❺ We use `self.assertIn` instead of just `assert` to make our test assertions. `unittest` provides lots of helper functions like this to make test assertions, like `assertEqual`, `assertTrue`, `assertFalse`, and so on. You can find more in the `unittest` documentation.

❻ `self.fail` just fails no matter what, producing the error message given. I'm using it as a reminder to finish the test.

❼ Finally, we have the `if __name__ == '__main__'` clause (if you've not seen it before, that's how a Python script checks if it's been executed from the command line, rather than just imported by another script). We call `unittest.main()`, which launches the `unittest` test runner, which will automatically find test classes and methods in the file and run them.

❽ `warnings='ignore'` suppresses a superfluous `ResourceWarning` which was being emitted at the time of writing. It may have disappeared by the time you read this; feel free to try removing it!

If you've read the Django testing documentation, you might have seen something called `LiveServerTestCase`, and are wondering whether we should use it now. Full points to you for reading the friendly manual! `LiveServerTestCase` is a bit too complicated for now, but I promise I'll use it in a later chapter…

Let's try it!

```
$ python3 functional_tests.py
F
======================================================================
FAIL: test_can_start_a_list_and_retrieve_it_later (__main__.NewVisitorTest)
 ----------------------------------------------------------------------
```

1. The only exception is if you have an exception inside `setUp`, then `tearDown` doesn't run.

```
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)
```

That's a bit nicer isn't it? It tidied up our Firefox window, it gives us a nicely formatted report of how many tests were run and how many failed, and the `assertIn` has given us a helpful error message with useful debugging info. Bonzer!

Implicit waits

There's one more thing to do at this stage: add an `implicitly_wait` in the `setUp`:

functional_tests.py.

```
[...]
def setUp(self):
    self.browser = webdriver.Firefox()
    self.browser.implicitly_wait(3)

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

This is a standard trope in Selenium tests. Selenium is reasonably good at waiting for pages to complete loading before it tries to do anything, but it's not perfect. The `implicitly_wait` tells it to wait a few seconds if it needs to. When asked to find something on the page, Selenium will now wait up to three seconds for it to appear.

Don't rely on `implicitly_wait`; it won't work for every use case. It will do its job while our app is still simple, but as we'll see in Part III (eg, in Chapter 15 and Chapter 20), you'll want to build more sophisticated, *explicit* wait algorithms into your tests once your app gets beyond a certain level of complexity.

Commit

This is a good point to do a commit; it's a nicely self-contained change. We've expanded our functional test to include comments that describe the task we're setting ourselves, our minimum viable to-do list. We've also rewritten it to use the Python `unittest` module and its various testing helper functions.

Do a **`git status`**—that should assure you that the only file that has changed is *functional_tests.py*. Then do a `git diff`, which shows you the difference between the last

commit and what's currently on disk. That should tell you that *functional_tests.py* has changed quite substantially:

```
$ 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()
+        self.browser.implicitly_wait(3)
+
+    def tearDown(self):
+        self.browser.quit()
[...]
```

Now let's do a:

```
$ git commit -a
```

The `-a` means "automatically add any changes to tracked files" (ie, any files that we've committed before). It won't add any brand new files (you have to explicitly `git add` them yourself), but often, as in this case, there aren't any new files, so it's a useful shortcut.

When the editor pops up, add a descriptive commit message, like "First FT specced out in comments, and now uses unittest".

Now we're in an excellent position to start writing some real code for our lists app. Read on!

Useful TDD Concepts

User Story
: A description of how the application will work from the point of view of the user. Used to structure a functional test.

Expected failure
: When a test fails in the way that we expected it to.

CHAPTER 3

Testing a Simple Home Page with Unit Tests

We finished the last chapter with a functional test failing, telling us that it wanted the home page for our site to have "To-Do" in its title. It's time to start working on our application.

Warning: Things Are About to Get Real

The first two chapters were intentionally nice and light. From now on, we get into some more meaty coding. Here's a prediction: at some point, things are going to go wrong. You're going to see different results from what I say you should see. This is a Good Thing, because it will be a genuine character-building Learning Experience™.

One possibility is that I've given some ambiguous explanations, and you've done something different from what I intended. Step back and have a think about what we're trying to achieve at this point in the book. Which file are we editing, what do we want the user to be able to do, what are we testing and why? It may be that you've edited the wrong file or function, or are running the wrong tests. I reckon you'll learn more about TDD from these stop and think moments than you do from all the bits where the following instructions and copy-pasting goes smoothly.

Or it may be a real bug. Be tenacious, read the error message carefully (see my aside on reading tracebacks a little later on in the chapter), and you'll get to the bottom of it. It's probably just a missing comma, or trailing-slash, or maybe a missing "s" in one of the Selenium find methods. But, as Zed Shaw put it so well, this kind of debugging is also an absolutely vital part of learning, so do stick it out!

You can always drop me an email (or try the Google Group) if you get really stuck. Happy debugging!

Our First Django App, and Our First Unit Test

Django encourages you to structure your code into *apps*: the theory is that one project can have many apps, you can use third-party apps developed by other people, and you might even reuse one of your own apps in a different project … although I admit I've never actually managed it myself! Still, apps are a good way to keep your code organised.

Let's start an app for our to-do lists:

```
$ python3 manage.py startapp lists
```

That will create a folder at *superlists/lists*, next to *superlists/superlists*, and within it a number of placeholder files for things like models, views, and, of immediate interest to us, tests:

```
superlists/
├── db.sqlite3
├── functional_tests.py
├── lists
│   ├── admin.py
│   ├── __init__.py
│   ├── migrations
│   │   └── __init__.py
│   ├── models.py
│   ├── tests.py
│   └── views.py
├── manage.py
└── superlists
    ├── __init__.py
    ├── __pycache__
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

Unit Tests, and How They Differ from Functional Tests

As with so many of the labels we put on things, the line between unit tests and functional tests can become a little blurry at times. The basic distinction, though, is that functional tests test the application from the outside, from the point of view of the user. Unit tests test the application from the inside, from the point of view of the programmer.

The TDD approach I'm following wants our application to be covered by both types of test. Our workflow will look a bit like this:

1. We start by writing a *functional test*, describing the new functionality from the user's point of view.
2. Once we have a functional test that fails, we start to think about how to write code that can get it to pass (or at least to get past its current failure). We now use one or

more *unit tests* to define how we want our code to behave—the idea is that each line of production code we write should be tested by (at least) one of our unit tests.

3. Once we have a failing unit test, we write the smallest amount of *application code* we can, just enough to get the unit test to pass. We may iterate between steps 2 and 3 a few times, until we think the functional test will get a little further.
4. Now we can rerun our functional tests and see if they pass, or get a little further. That may prompt us to write some new unit tests, and some new code, and so on.

You can see that, all the way through, the functional tests are driving what development we do from a high level, while the unit tests drive what we do at a low level.

Does that seem slightly redundant? Sometimes it can feel that way, but functional tests and unit tests do really have very different objectives, and they will usually end up looking quite different.

Functional tests should help you build an application with the right functionality, and guarantee you never accidentally break it. Unit tests should help you to write code that's clean and bug free.

Enough theory for now, let's see how it looks in practice.

Unit Testing in Django

Let's see how to write a unit test for our home page view. Open up the new file at *lists/tests.py*, and you'll see something like this:

lists/tests.py.

```
from django.test import TestCase

# Create your tests here.
```

Django has helpfully suggested we use a special version of `TestCase`, which it provides. It's an augmented version of the standard `unittest.TestCase`, with some additional Django-specific features, which we'll discover over the next few chapters.

You've already seen that the TDD cycle involves starting with a test that fails, then writing code to get it to pass. Well, before we can even get that far, we want to know that the unit test we're writing will definitely be run by our automated test runner, whatever it is. In the case of *functional_tests.py*, we're running it directly, but this file made by Django is a bit more like magic. So, just to make sure, let's make a deliberately silly failing test:

lists/tests.py.

```
from django.test import TestCase

class SmokeTest(TestCase):

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

Now let's invoke this mysterious Django test runner. As usual, it's a *manage.py* command:

```
$ python3 manage.py test
Creating test database for alias 'default'...
F
======================================================================
FAIL: test_bad_maths (lists.tests.SmokeTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/workspace/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)
Destroying test database for alias 'default'...
```

Excellent. The machinery seems to be working. This is a good point for a commit:

```
$ git status  # should show you lists/ is untracked
$ git add lists
$ git diff --staged  # will show you the diff that you're about to commit
$ git commit -m"Add app for lists, with deliberately failing unit test"
```

As you've no doubt guessed, the `-m` flag lets you pass in a commit message at the command line, so you don't need to use an editor. It's up to you to pick the way you like to use the Git command line, I'll just show you the main ones I've seen used. The main rule is: *make sure you always review what you're about to commit before you do it*.

Django's MVC, URLs, and View Functions

Django is broadly structured along a classic *Model-View-Controller* (MVC) pattern. Well, *broadly*. It definitely does have models, but its views are more like a controller, and it's the templates that are actually the view part, but the general idea is there. If you're interested, you can look up the finer points of the discussion in the Django FAQs.

Irrespective of any of that, like any web server, Django's main job is to decide what to do when a user asks for a particular URL on our site. Django's workflow goes something like this:

1. An HTTP *request* comes in for a particular *URL*.
2. Django uses some rules to decide which *view* function should deal with the request (this is referred to as *resolving* the URL).
3. The view function processes the request and returns an HTTP *response*.

So we want to test two things:

- Can we resolve the URL for the root of the site ("/") to a particular view function we've made?
- Can we make this view function return some HTML which will get the functional test to pass?

Let's start with the first. Open up *lists/tests.py*, and change our silly test to something like this:

lists/tests.py.

```
from django.core.urlresolvers import resolve
from django.test import TestCase
from lists.views import home_page #❶

class HomePageTest(TestCase):

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

What's going on here?

❷ ❸ `resolve` is the function Django uses internally to resolve URLs, and find what view function they should map to. We're checking that `resolve`, when called with "/", the root of the site, finds a function called `home_page`.

❶ What function is that? It's the view function we're going to write next, which will actually return the HTML we want. You can see from the `import` that we're planning to store it in *lists/views.py*.

So, what do you think will happen when we run the tests?

```
$ python3 manage.py test
ImportError: cannot import name 'home_page'
```

It's a very predictable and uninteresting error: we tried to import something we haven't even written yet. But it's still good news—for the purposes of TDD, an exception which was predicted counts as an expected failure. Since we have both a failing functional test and a failing unit test, we have the Testing Goat's full blessing to code away.

At Last! We Actually Write Some Application Code!

It is exciting isn't it? Be warned, TDD means that long periods of anticipation are only defused very gradually, and by tiny increments. Especially since we're learning and only just starting out, we only allow ourselves to change (or add) one line of code at a time —and each time, we make just the minimal change required to address the current test failure.

I'm being deliberately extreme here, but what's our current test failure? We can't import `home_page` from `lists.views`? OK, let's fix that—and only that. In *lists/views.py*:

lists/views.py.

```
from django.shortcuts import render

# Create your views here.
home_page = None
```

"*You must be joking!*" I can hear you say.

I can hear you because it's what I used to say (with feeling) when my colleagues first demonstrated TDD to me. Well, bear with me, we'll talk about whether or not this is all taking it too far in a little while. For now, let yourself follow along, even if it's with some exasperation, and see where it takes us.

Let's run the tests again:

```
$ python3 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 "/workspace/superlists/lists/tests.py", line 8, in
test_root_url_resolves_to_home_page_view
    found = resolve('/')
  File "/usr/local/lib/python3.4/dist-packages/django/core/urlresolvers.py",
line 485, in resolve
    return get_resolver(urlconf).resolve(path)
  File "/usr/local/lib/python3.4/dist-packages/django/core/urlresolvers.py",
line 353, in resolve
    raise Resolver404({'tried': tried, 'path': new_path})
django.core.urlresolvers.Resolver404: {'tried': [[<RegexURLResolver
<RegexURLPattern list> (admin:admin) ^admin/>]], 'path': ''}

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

FAILED (errors=1)
Destroying test database for alias 'default'...
```

Reading Tracebacks

Let's spend a moment talking about how to read tracebacks, since it's something we have to do a lot in TDD. You soon learn to scan through them and pick up relevant clues:

```
======================================================================
ERROR: test_root_url_resolves_to_home_page_view (lists.tests.HomePageTest)❶
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/workspace/superlists/lists/tests.py", line 8, in
test_root_url_resolves_to_home_page_view
    found = resolve('/')❷
  File "/usr/local/lib/python3.4/dist-packages/django/core/urlresolvers.py",
line 485, in resolve
    return get_resolver(urlconf).resolve(path)
  File "/usr/local/lib/python3.4/dist-packages/django/core/urlresolvers.py",
line 353, in resolve
    raise Resolver404({'tried': tried, 'path': new_path})
django.core.urlresolvers.Resolver404: {'tried': [[<RegexURLResolver❸
<RegexURLPattern list> (admin:admin) ^admin/>]], 'path': ''}❹
 ----------------------------------------------------------------------
[...]
```

❸ ❹ The first place you look is usually *the error itself*. Sometimes that's all you need to see, and it will let you identify the problem immediately. But sometimes, like in this case, it's not quite self-evident.

❶ The next thing to double-check is: *which test is failing?* Is it definitely the one we expected, ie, the one we just wrote? In this case, the answer is yes.

❷ Then we look for the place in *our test code* that kicked off the failure. We work our way down from the top of the traceback, looking for the filename of the tests file, to check which test function, and what line of code, the failure is coming from. In this case it's the line where we call the `resolve` function for the "/" URL.

There is ordinarily a fourth step, where we look further down for any of *our own application code* which was involved with the problem. In this case it's all Django code, but we'll see plenty of examples of this fourth step later in the book.

Pulling it all together, we interpret the traceback as telling us that, when trying to resolve "/", Django raised a 404 error—in other words, Django can't find a URL mapping for "/". Let's help it out.

urls.py

Django uses a file called *urls.py* to define how URLs map to view functions. There's a main *urls.py* for the whole site in the *superlists/superlists* folder. Let's go take a look:

superlists/urls.py.

```
from django.conf.urls import patterns, include, url
from django.contrib import admin

urlpatterns = patterns('',
    # Examples:
    # url(r'^$', 'superlists.views.home', name='home'),
    # url(r'^blog/', include('blog.urls')),

    url(r'^admin/', include(admin.site.urls)),
)
```

As usual, lots of helpful comments and default suggestions from Django.

A `url` entry starts with a regular expression that defines which URLs it applies to, and goes on to say where it should send those requests—either to a dot-notation encoded function like `superlists.views.home`, or maybe to another *urls.py* file somewhere else using `include`.

You can see there's one entry in there by default there for the admin site. We're not using that yet, so let's comment it out for now:

superlists/urls.py.

```
from django.conf.urls import patterns, include, url
from django.contrib import admin

urlpatterns = patterns('',
    # Examples:
    # url(r'^$', 'superlists.views.home', name='home'),
    # url(r'^blog/', include('blog.urls')),

    # url(r'^admin/', include(admin.site.urls)),
)
```

The first entry in `urlpatterns` has the regular expression `^$`, which means an empty string—could this be the same as the root of our site, which we've been testing with "/"? Let's find out—what happens if we uncomment that line?

If you've never come across regular expressions, you can get away with just taking my word for it, for now—but you should make a mental note to go learn about them.

superlists/urls.py.

```
urlpatterns = patterns('',
    # Examples:
    url(r'^$', 'superlists.views.home', name='home'),
    # url(r'^blog/', include('blog.urls')),

    # url(r'^admin/', include(admin.site.urls)),
)
```

Run the unit tests again, with **`python3 manage.py test`**:

```
ImportError: No module named 'superlists.views'
[...]
django.core.exceptions.ViewDoesNotExist: Could not import
superlists.views.home. Parent module superlists.views does not exist.
```

That's progress! We're no longer getting a 404; instead Django is complaining that the dot-notation `superlists.views.home` doesn't point to a real view. Let's fix that, by pointing it towards our placeholder `home_page` object, which is inside *lists*, not *superlists*:

superlists/urls.py.

```
urlpatterns = patterns('',
    # Examples:
    url(r'^$', 'lists.views.home_page', name='home'),
```

And run the tests again:

```
django.core.exceptions.ViewDoesNotExist: Could not import
lists.views.home_page. View is not callable.
```

The unit tests have made the link between the URL / and the `home_page = None` in *lists/views.py*, and are now complaining that `home_page` isn't a callable; ie, it's not a function. Now we've got a justification for changing it from being `None` to being an actual function. Every single code change is driven by the tests. Back in *lists/views.py*:

lists/views.py.

```
from django.shortcuts import render

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

And now?

```
$ python3 manage.py test
Creating test database for alias 'default'...
.
 ----------------------------------------------------------------------
Ran 1 test in 0.003s

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

Hooray! Our first ever unit test pass! That's so momentous that I think it's worthy of a commit:

```
$ git diff  # should show changes to urls.py, tests.py, and views.py
$ git commit -am"First unit test and url mapping, dummy view"
```

That was the last variation on `git commit` I'll show, the `a` and `m` flags together, which adds all changes to tracked files and uses the commit message from the command line.

`git commit -am` is the quickest formulation, but also gives you the least feedback about what's being committed, so make sure you've done a `git status` and a `git diff` beforehand, and are clear on what changes are about to go in.

Unit Testing a View

On to writing a test for our view, so that it can be something more than a do-nothing function, and instead be a function that returns a real response with HTML to the browser. Open up *lists/tests.py*, and add a new *test method*. I'll explain each bit:

lists/tests.py.

```
from django.core.urlresolvers 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
        self.assertTrue(response.content.startswith(b'<html>'))  #❸
        self.assertIn(b'<title>To-Do lists</title>', response.content)  #❹
        self.assertTrue(response.content.endswith(b'</html>'))  #❺
```

What's going on in this new test?

❶ We create an `HttpRequest` object, which is what Django will see when a user's browser asks for a page.

❷ We pass it to our `home_page` view, which gives us a response. You won't be surprised to hear that this object is an instance of a class called `HttpResponse`. Then, we assert that the `.content` of the response—which is the HTML that we send to the user—has certain properties.

❸ ❺ We want it to start with an `<html>` tag which gets closed at the end. Notice that `response.content` is raw bytes, not a Python string, so we have to use the `b''` syntax to compare them. More info is available in Django's Porting to Python 3 docs.

❹ And we want a `<title>` tag somewhere in the middle, with the word "To-Do" in it—because that's what we specified in our functional test.

Once again, the unit test is driven by the functional test, but it's also much closer to the actual code—we're thinking like programmers now.

Let's run the unit tests now and see how we get on:

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

The Unit-Test/Code Cycle

We can start to settle into the TDD *unit-test/code cycle* now:

1. In the terminal, run the unit tests and see how they fail.
2. In the editor, make a minimal code change to address the current test failure.

And repeat!

The more nervous we are about getting our code right, the smaller and more minimal we make each code change—the idea is to be absolutely sure that each bit of code is justified by a test. It may seem laborious, but once you get into the swing of things, it really moves quite fast—so much so that, at work, we usually keep our code changes microscopic even when we're confident we could skip ahead.

Let's see how fast we can get this cycle going:

- Minimal code change:

lists/views.py

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

- Tests:

```
        self.assertTrue(response.content.startswith(b'<html>'))
    AttributeError: 'NoneType' object has no attribute 'content'
```

- Code—we use `django.http.HttpResponse`, as predicted:

lists/views.py

```
from django.http import HttpResponse

# Create your views here.
```

```
def home_page(request):
    return HttpResponse()
```

- Tests again:

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

- Code again:

lists/views.py.

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

- Tests:

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

- Code:

lists/views.py.

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

- Tests—almost there?

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

- Come on, one last effort:

lists/views.py.

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

- Surely?

```
$ python3 manage.py test
Creating test database for alias 'default'...
..
 ----------------------------------------------------------------------
Ran 2 tests in 0.001s

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

Yes! Now, let's run our functional tests. Don't forget to spin up the dev server again, if it's not still running. It feels like the final heat of the race here, surely this is it … could it be?

```
$ python3 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 20, 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)
```

Failed? What? Oh, it's just our little reminder? Yes? Yes! We have a web page!

Ahem. Well, *I* thought it was a thrilling end to the chapter. You may still be a little baffled, perhaps keen to hear a justification for all these tests, and don't worry, all that will come, but I hope you felt just a tinge of excitement near the end there.

Just a little commit to calm down, and reflect on what we've covered:

```
$ git diff  # should show our new test in tests.py, and the view in views.py
$ git commit -am"Basic view now returns minimal HTML"
```

That was quite a chapter! Why not try typing `git log`, possibly using the `--oneline` flag, for a reminder of what we got up to:

```
$ 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
[...]
```

Not bad—we covered:

- Starting a Django app
- The Django unit test runner
- The difference between FTs and unit tests
- Django URL resolving and *urls.py*
- Django view functions, request and response objects
- And returning basic HTML

Useful Commands and Concepts

Running the Django dev server

```
python3 manage.py runserver
```

Running the functional tests

```
python3 functional_tests.py
```

Running the unit tests

```
python3 manage.py test
```

The unit-test/code cycle

1. Run the unit tests in the terminal.
2. Make a minimal code change in the editor.
3. Repeat!

CHAPTER 4
What Are We Doing with All These Tests?

Now that we've seen the basics of TDD in action, it's time to pause and talk about why we're doing it.

I'm imagining several of you, dear readers, have been holding back some seething frustration—perhaps some of you have done a bit of unit testing before, and perhaps some of you are just in a hurry. You've been biting back questions like:

- Aren't all these tests a bit excessive?
- Surely some of them are redundant? There's duplication between the functional tests and the unit tests.
- I mean, what are you doing importing `django.core.urlresolvers` in your unit tests? Isn't that testing Django, ie, testing third-party code? I thought that was a no-no?
- Those unit tests seemed way too trivial—testing one line of declaration, and a one-line function that returns a constant! Isn't that just a waste of time? Shouldn't we save our tests for more complex things?
- What about all those tiny changes during the unit-test/code cycle? Surely we could have just skipped to the end? I mean, `home_page = None`!? Really?
- You're not telling me you *actually* code like this in real life?

Ah, young grasshopper. I too was once full of questions like these. But only because they're perfectly good questions. In fact, I still ask myself questions like these, all the time. Does all this stuff really have value? Is this a bit of a cargo cult?

Programming Is like Pulling a Bucket of Water up from a Well

Ultimately, programming is hard. Often, we are smart, so we succeed. TDD is there to help us out when we're not so smart. Kent Beck (who basically invented TDD) uses the metaphor of lifting a bucket of water out of a well with a rope: when the well isn't too deep, and the bucket isn't very full, it's easy. And even lifting a full bucket is pretty easy at first. But after a while, you're going to get tired. TDD is like having a ratchet that lets you save your progress, take a break, and make sure you never slip backwards. That way you don't have to be smart *all* the time.

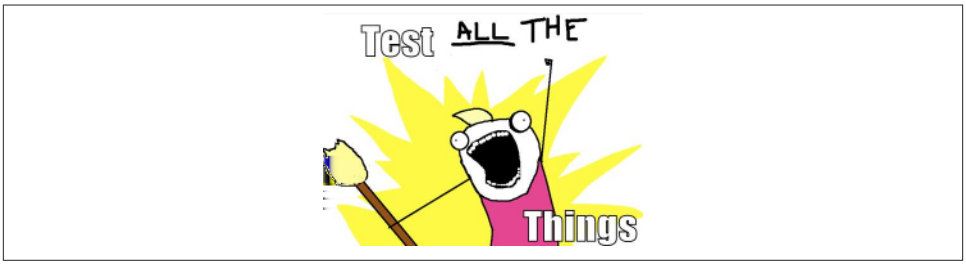


Figure 4-1. Test ALL the things (original illustration source: Allie Brosh, Hyperbole and a Half)

OK, perhaps *in general*, you're prepared to concede that TDD is a good idea, but maybe you still think I'm overdoing it? Testing the tiniest thing, and taking ridiculously many small steps?

TDD is a *discipline*, and that means it's not something that comes naturally; because many of the payoffs aren't immediate but only come in the longer term, you have to force yourself to do it in the moment. That's what the image of the Testing Goat is supposed to illustrate—you need to be a bit bloody-minded about it.

On the Merits of Trivial Tests for Trivial Functions

In the short term it may feel a bit silly to write tests for simple functions and constants. It's perfectly possible to imagine still doing "mostly" TDD, but following more relaxed rules where you don't unit test *absolutely* everything. But in this book my aim is to demonstrate full, rigorous TDD. Like a kata in a martial art, the idea is to learn the motions in a controlled context, when there is no adversity, so that the techiques are part of your muscle memory. It seems trivial now, because we've started with a very simple example. The problem comes when your application gets complex—that's when you really need your tests. And the danger is that complexity tends to sneak up on you, gradually. You may not notice it happening, but quite soon you're a boiled frog.

There are two other things to say in favour of tiny, simple tests for simple functions:

Firstly, if they're really trivial tests, then they won't take you that long to write them. So stop moaning and just write them already.

Secondly, it's always good to have a placeholder. Having a test *there* for a simple function means it's that much less of a psychological barrier to overcome when the simple function gets a tiny bit more complex—perhaps it grows an `if`. Then a few weeks later it grows a `for` loop. Before you know it, it's a recursive metaclass-based polymorphic tree parser factory. But because it's had tests from the very beginning, adding a new test each time has felt quite natural, and it's well tested. The alternative involves trying to decide when a function becomes "complicated enough" which is highly subjective, but worse, because there's no placeholder, it seems like that much more effort, and you're tempted each time to put it off a little longer, and pretty soon—frog soup!

Instead of trying to figure out some hand-wavy subjective rules for when you should write tests, and when you can get away with not bothering, I suggest following the discipline for now—like any discipline, you have to take the time to learn the rules before you can break them.

Now, back to our onions.

Using Selenium to Test User Interactions

Where were we at the end of the last chapter? Let's rerun the test and find out:

```
$ python3 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 20, 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)
```

Did you try it, and get an error saying *Problem loading page* or *Unable to connect*? So did I. It's because we forgot to spin up the dev server first using `manage.py runserver`. Do that, and you'll get the failure message we're after.

One of the great things about TDD is that you never have to worry about forgetting what to do next—just rerun your tests and they will tell you what you need to work on.

"Finish the test", it says, so let's do just that! Open up *functional_tests.py* and we'll extend our FT:

functional_tests.py.

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

class NewVisitorTest(unittest.TestCase):

    def setUp(self):
        self.browser = webdriver.Firefox()
        self.browser.implicitly_wait(3)

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

    def test_can_start_a_list_and_retrieve_it_later(self):
        # Edith has heard about a cool new online to-do app. She goes
        # to check out its homepage
        self.browser.get('http://localhost:8000')

        # She notices the page title and header mention to-do lists
        self.assertIn('To-Do', self.browser.title)
        header_text = self.browser.find_element_by_tag_name('h1').text
        self.assertIn('To-Do', header_text)

        # She is invited to enter a to-do item straight away
        inputbox = self.browser.find_element_by_id('id_new_item')
        self.assertEqual(
                inputbox.get_attribute('placeholder'),
                'Enter a to-do item'
        )

        # She types "Buy peacock feathers" into a text box (Edith's hobby
        # is tying fly-fishing lures)
        inputbox.send_keys('Buy peacock feathers')

        # When she hits enter, the page updates, and now the page lists
        # "1: Buy peacock feathers" as an item in a to-do list table
        inputbox.send_keys(Keys.ENTER)

        table = self.browser.find_element_by_id('id_list_table')
        rows = table.find_elements_by_tag_name('tr')
        self.assertTrue(
```

```
            any(row.text == '1: Buy peacock feathers' for row in rows)
        )

        # There is still a text box inviting her to add another item. She
        # enters "Use peacock feathers to make a fly" (Edith is very
        # methodical)
        self.fail('Finish the test!')

        # The page updates again, and now shows both items on her list
        [...]
```

We're using several of the methods that Selenium provides to examine web pages: `find_element_by_tag_name`, `find_element_by_id`, and `find_elements_by_tag_name` (notice the extra `s`, which means it will return several elements rather than just one). We also use `send_keys`, which is Selenium's way of typing into input elements. You'll also see the `Keys` class (don't forget to import it), which lets us send special keys like Enter, but also modifiers like Ctrl.

Watch out for the difference between the Selenium `find_element_by...` and `find_elements_by...` functions. One returns an element, and raises an exception if it can't find it, whereas the other returns a list, which may be empty.

Also, just look at that `any` function. It's a little-known Python built-in. I don't even need to explain it, do I? Python is such a joy.

Although, if you're one of my readers who doesn't know Python, what's happening inside the `any` is a *generator expression*, which is like a *list comprehension* but awesomer. You need to read up on this. If you Google it, you'll find Guido himself explaining it nicely. Come back and tell me that's not pure joy!

Let's see how it gets on:

```
$ python3 functional_tests.py
[...]
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"tag name","selector":"h1"}' ; Stacktrace: [...]
```

Decoding that, the test is saying it can't find an `<h1>` element on the page. Let's see what we can do to add that to the HTML of our home page.

Big changes to a functional test are usually a good thing to commit on their own. I failed to do so in my first draft, and I regretted it later when I changed my mind and had the change mixed up with a bunch of others. The more atomic your commits, the better:

```
$ git diff  # should show changes to functional_tests.py
$ git commit -am "Functional test now checks we can input a to-do item"
```

The "Don't Test Constants" Rule, and Templates to the Rescue

Let's take a look at our unit tests, *lists/tests.py*. Currently we're looking for specific HTML strings, but that's not a particularly efficient way of testing HTML. In general, one of the rules of unit testing is *Don't test constants*, and testing HTML as text is a lot like testing a constant.

In other words, if you have some code that says:

```
wibble = 3
```

There's not much point in a test that says:

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

Unit tests are really about testing logic, flow control, and configuration. Making assertions about exactly what sequence of characters we have in our HTML strings isn't doing that.

What's more, mangling raw strings in Python really isn't a great way of dealing with HTML. There's a much better solution, which is to use templates. Quite apart from anything else, if we can keep HTML to one side in a file whose name ends in *.html*, we'll get better syntax highlighting! There are lots of Python templating frameworks out there, and Django has its own which works very well. Let's use that.

Refactoring to Use a Template

What we want to do now is make our view function return exactly the same HTML, but just using a different process. That's a refactor—when we try to improve the code *without changing its functionality*.

That last bit is really important. If you try and add new functionality at the same time as refactoring, you're much more likely to run into trouble. Refactoring is actually a whole discipline in itself, and it even has a reference book: Martin Fowler's Refactoring.

The first rule is that you can't refactor without tests. Thankfully, we're doing TDD, so we're way ahead of the game. Let's check our tests pass; they will be what makes sure that our refactoring is behaviour preserving:

```
$ python3 manage.py test
[...]
OK
```

Great! We'll start by taking our HTML string and putting it into its own file. Create a directory called *lists/templates* to keep templates in, and then open a file at *lists/templates/home.html*, to which we'll transfer our HTML:[1]

lists/templates/home.html.

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

Mmmh, syntax-highlighted … much nicer! Now to change our view function:

lists/views.py.

```
from django.shortcuts import render

def home_page(request):
    return render(request, 'home.html')
```

Instead of building our own `HttpResponse`, we now use the Django `render` function. It takes the request as its first parameter (for reasons we'll go into later) and the name of the template to render. Django will automatically search folders called *templates* inside any of your apps' directories. Then it builds an `HttpResponse` for you, based on the content of the template.

Templates are a very powerful feature of Django's, and their main strength consists of substituting Python variables into HTML text. We're not using this feature yet, but we will in future chapters. That's why we use `render` and (later) `render_to_ string` rather than, say, manually reading the file from disk with the built-in `open`.

Let's see if it works:

```
$ python3 manage.py test
[...]
======================================================================
ERROR: test_home_page_returns_correct_html (lists.tests.HomePageTest)❶
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/workspace/superlists/lists/tests.py", line 17, in
test_home_page_returns_correct_html
    response = home_page(request)❷
  File "/workspace/superlists/lists/views.py", line 5, in home_page
    return render(request, 'home.html')❸
  File "/usr/local/lib/python3.3/dist-packages/django/shortcuts.py", line 48,
in render
    return HttpResponse(loader.render_to_string(*args, **kwargs),
```

1. Some people like to use another subfolder named after the app (ie, *lists/templates/lists*) and then refer to the template as *lists/home.html*. This is called "template namespacing". I figured it was overcomplicated for this small project, but it may be worth it on larger projects. There's more in the Django tutorial.

```
  File "/usr/local/lib/python3.3/dist-packages/django/template/loader.py", line
170, in render_to_string
    t = get_template(template_name, dirs)
  File "/usr/local/lib/python3.3/dist-packages/django/template/loader.py", line
144, in get_template
    template, origin = find_template(template_name, dirs)
  File "/usr/local/lib/python3.3/dist-packages/django/template/loader.py", line
136, in find_template
    raise TemplateDoesNotExist(name)
django.template.base.TemplateDoesNotExist: home.html❹

 ----------------------------------------------------------------------
Ran 2 tests in 0.004s
```

Another chance to analyse a traceback:

❹ We start with the error: it can't find the template.

❶ Then we double-check what test is failing: sure enough, it's our test of the view HTML.

❷ Then we find the line in our tests that caused the failure: it's when we call the `home_page` function.

❸ Finally, we look for the part of our own application code that caused the failure: it's when we try and call `render`.

So why can't Django find the template? It's right where it's supposed to be, in the *lists/templates* folder.

The thing is that we haven't yet *officially* registered our lists app with Django. Unfortunately, just running the `startapp` command and having what is obviously an app in your project folder isn't quite enough. You have to tell Django that you *really* mean it, and add it to *settings.py* as well. Belt and braces. Open it up and look for a variable called `INSTALLED_APPS`, to which we'll add `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',
)
```

You can see there's lots of apps already in there by default. We just need to add ours, `lists`, to the bottom of the list. Don't forget the trailing comma—it may not be required,

but one day you'll be really annoyed when you forget it and Python concatenates two strings on different lines…

Now we can try running the tests again:

```
$ python3 manage.py test
    [...]
    self.assertTrue(response.content.endswith(b'</html>'))
AssertionError: False is not true
```

Darn, not quite.

Depending on whether your text editor insists on adding newlines to the end of files, you may not even see this error. If so, you can safely ignore the next bit, and skip straight to where you can see the listing says OK.

But it did get further! It seems it's managed to find our template, but the last of the three assertions is failing. Apparently there's something wrong at the end of the output. I had to do a little `print repr(response.content)` to debug this, but it turns out that the switch to templates has introduced an additional newline (`\n`) at the end. We can get them to pass like this:

lists/tests.py.

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

It's a tiny bit of a cheat, but whitespace at the end of an HTML file really shouldn't matter to us. Let's try running the tests again:

```
$ python3 manage.py test
[...]
OK
```

Our refactor of the code is now complete, and the tests mean we're happy that behaviour is preserved. Now we can change the tests so that they're no longer testing constants; instead, they should just check that we're rendering the right template. Another Django helper function called `render_to_string` is our friend here:

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)
        expected_html = render_to_string('home.html')
        self.assertEqual(response.content.decode(), expected_html)
```

We use `.decode()` to convert the `response.content` bytes into a Python unicode string, which allows us to compare strings with strings, instead of bytes with bytes as we did earlier.

The main point, though, is that instead of testing constants we're testing our implementation. Great!

Django has a test client with tools for testing templates, which we'll use in later chapters. For now we'll use the low-level tools to make sure we're comfortable with how everything works. No magic!

On Refactoring

That was an absolutely trivial example of refactoring. But, as Kent Beck puts it in *Test-Driven Development: By Example*, "Am I recommending that you actually work this way? No. I'm recommending that you be *able* to work this way."

In fact, as I was writing this my first instinct was to dive in and change the test first—make it use the `render_to_string` function straight away, delete the three superfluous assertions, leaving just a check of the contents against the expected render, and then go ahead and make the code change. But notice how that actually would have left space for me to break things: I could have defined the template as containing *any* arbitrary string, instead of the string with the right `<html>` and `<title>` tags.

When refactoring, work on either the code or the tests, but not both at once.

There's always a tendency to skip ahead a couple of steps, to make a couple of tweaks to the behaviour while you're refactoring, but pretty soon you've got changes to half a dozen different files, you've totally lost track of where you are, and nothing works any more. If you don't want to end up like Refactoring Cat (Figure 4-2), stick to small steps; keep refactoring and functionality changes entirely separate.

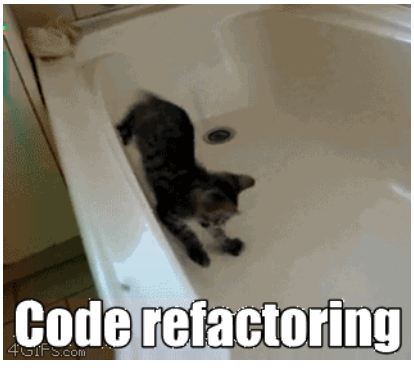


Figure 4-2. Refactoring Cat—be sure to look up the full animated GIF (source: 4GIFs.com)

We'll come across "Refactoring Cat" again during this book, as an example of what happens when we get carried away and want to change too many things at once. Think of it as the little cartoon demon counterpart to the Testing Goat, popping up over your other shoulder and giving you bad advice…

It's a good idea to do a commit after any refactoring:

```
$ git status # see tests.py, views.py, settings.py, + new templates folder
$ git add .  # will also add the untracked templates folder
$ git diff --staged # review the changes we're about to commit
$ git commit -m"Refactor home page view to use a template"
```

A Little More of Our Front Page

In the meantime, our functional test is still failing. Let's now make an actual code change to get it passing. Because our HTML is now in a template, we can feel free to make changes to it, without needing to write any extra unit tests. We wanted an `<h1>`:

lists/templates/home.html.

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

Let's see if our functional test likes it a little better:

```
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"id","selector":"id_new_item"}' ; Stacktrace: [...]
```

OK…

lists/templates/home.html.

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

And now?

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

We add our placeholder text…

lists/templates/home.html.

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

Which gives:

```
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"id","selector":"id_list_table"}' ; Stacktrace: [...]
```

So we can go ahead and put the table onto the page. At this stage it'll just be empty…

lists/templates/home.html.

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

Now what does the FT say?

```
  File "functional_tests.py", line 42, 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
```

Slightly cryptic. We can use the line number to track it down, and it turns out it's that `any` function I was so smug about earlier—or, more precisely, the `assertTrue`, which doesn't have a very explicit failure message. We can pass a custom error message as an argument to most `assertX` methods in `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"
    )
```

If you run the FT again, you should see our message:

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

But now, to get this to pass, we will need to actually process the user's form submission. And that's a topic for the next chapter.

For now let's do a commit:

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

Thanks to a bit of refactoring, we've got our view set up to render a template, we've stopped testing constants, and we're now well placed to start processing user input.

Recap: The TDD Process

We've now seen all the main aspects of the TDD process, in practice:

- Functional tests
- Unit tests
- The unit-test/code cycle
- Refactoring

It's time for a little recap, and perhaps even some flowcharts. Forgive me, years misspent as a management consultant have ruined me. On the plus side, it will feature recursion.

What is the overall TDD process? See Figure 4-3.

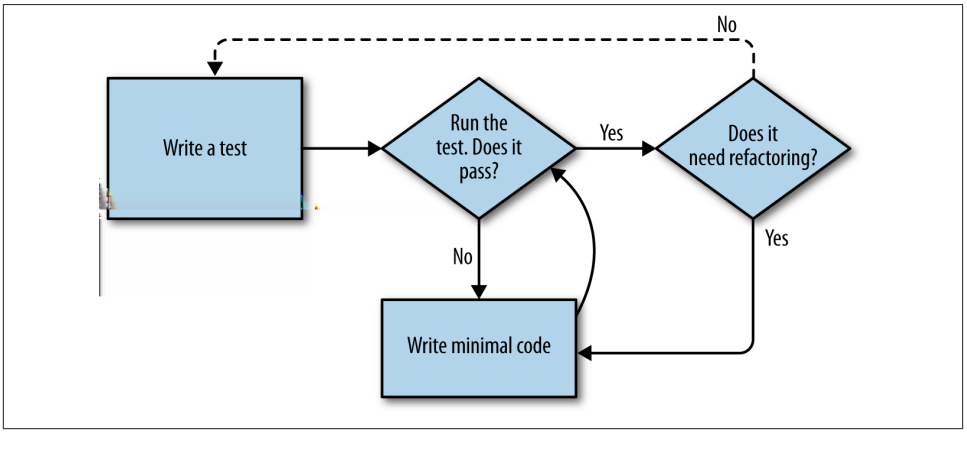


Figure 4-3. Overall TDD process

We write a test. We run the test and see it fail. We write some minimal code to get it a little further. We rerun the test and repeat until it passes. Then, optionally, we might refactor our code, using our tests to make sure we don't break anything.

But how does this apply when we have functional tests *and* unit tests? Well, you can think of the functional test as being a high-level view of the cycle, where "writing the

code" to get the functional tests to pass actually involves using another, smaller TDD cycle which uses unit tests. See Figure 4-4.

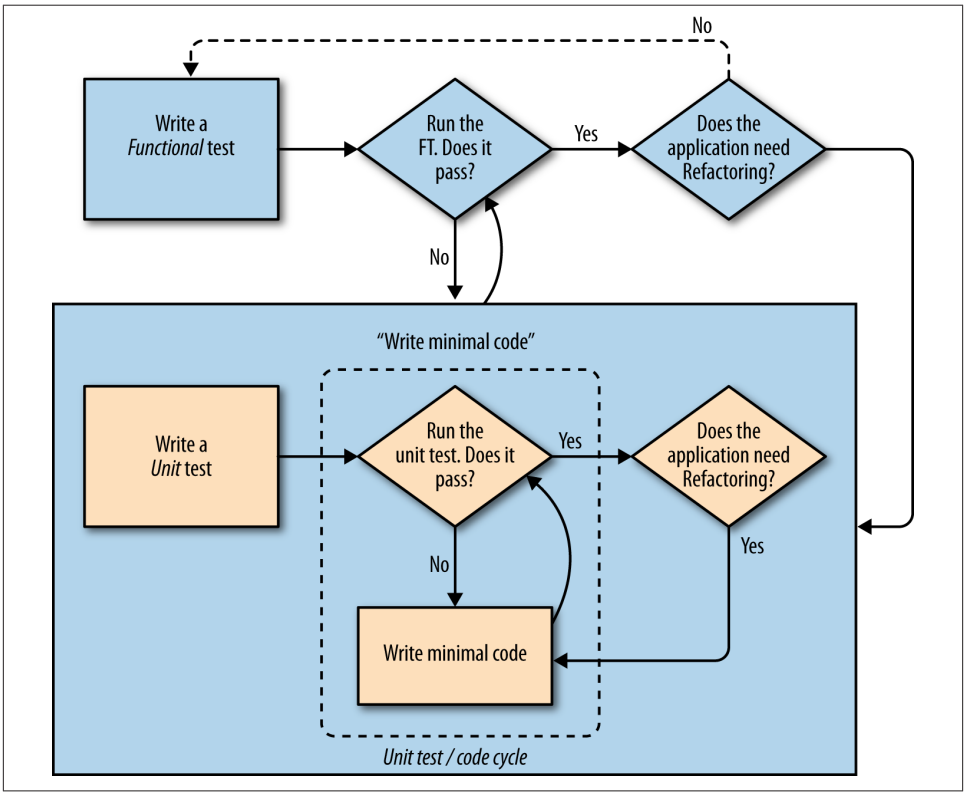


Figure 4-4. The TDD process with functional and unit tests

We write a functional test and see it fail. Then, the process of "writing code" to get it to pass is a mini-TDD cycle of its own: we write one or more unit tests, and go into the unit-test/code cycle until the unit tests pass. Then, we go back to our FT to check that it gets a little further, and we can write a bit more of our application—using more unit tests, and so on.

What about refactoring, in the context of functional tests? Well, that means we use the functional test to check that we've preserved the behaviour of our application, but we can change or add and remove unit tests, and use a unit test cycle to actually change the implementation.

The functional tests are the ultimate judge of whether your application works or not. The unit tests are a tool to help you along the way.

This way of looking at things is sometimes called "Double-Loop TDD". One of my eminent tech reviewers, Emily Bache, wrote a blog post on the topic, which I recommend for a different perspective.

We'll explore all of the different parts of this workflow in more detail over the coming chapters.

How to "Check" Your Code, or Skip Ahead (If You Must)

All of the code examples I've used in the book are available in my repo on GitHub. So, if you ever want to compare your code against mine, you can take a look at it there.

Each chapter has its own branch following the convention `chapter_XX`:

- Chapter 3: *https://github.com/hjwp/book-example/tree/chapter_03*
- Chapter 4: *https://github.com/hjwp/book-example/tree/chapter_04*
- Chapter 5: *https://github.com/hjwp/book-example/tree/chapter_05*
- Etc.

Be aware that each branch contains all of the commits for that chapter, so its state represents the code at the *end* of the chapter.

Using Git to check your progress

If you feel like developing your Git-Fu a little further, you can add my repo as a *remote*:

```
git remote add harry https://github.com/hjwp/book-example.git
git fetch harry
```

And then, to check your difference from the *end* of Chapter 4:

```
git diff harry/chapter_04
```

Git can handle multiple remotes, so you can still do this even if you're already pushing your code up to GitHub or Bitbucket.

Be aware that the precise order of, say, methods in a class may differ between your version and mine. It may make diffs hard to read.

Downloading a ZIP file for a chapter

If, for whatever reason, you want to "start from scratch" for a chapter, or skip ahead,[2] and/or you're just not comfortable with Git, you can download a version of my code as a ZIP file, from URLs following this pattern:

https://github.com/hjwp/book-example/archive/chapter_05.zip

https://github.com/hjwp/book-example/archive/chapter_06.zip

Don't let it become a crutch!

Try not to sneak a peak at the answers unless you're really, really stuck. Like I said at the beginning of the last chapter, there's a lot of value in debugging errors all by yourself, and in real life, there's no "harrys repo" to check against and find all the answers.

2. I don't recommend skipping ahead. I haven't designed the chapters to stand on their own; each relies on the previous ones, so it may be more confusing than anything else…

CHAPTER 5
Saving User Input

We want to take the to-do item input from the user and send it to the server, so that we can save it somehow and display it back to her later.

As I started writing this chapter, I immediately skipped to what I thought was the right design: multiple models for lists and list items, a bunch of different URLs for adding new lists and items, three new view functions, and about half a dozen new unit tests for all of the above. But I stopped myself. Although I was pretty sure I was smart enough to handle all those problems at once, the point of TDD is to allow you to do one thing at a time, when you need to. So I decided to be deliberately short-sighted, and at any given moment only do what was necessary to get the functional tests a little further.

It's a demonstration of how TDD can support an iterative style of development—it may not be the quickest route, but you do get there in the end. There's a neat side benefit, which is that it allows me to introduce new concepts like models, dealing with POST requests, Django template tags, and so on *one at a time* rather than having to dump them on you all at once.

None of this says that you *shouldn't* try and think ahead, and be clever. In the next chapter we'll use a bit more design and up-front thinking, and show how that fits in with TDD. But for now let's plough on mindlessly and just do what the tests tell us to.

Wiring Up Our Form to Send a POST Request

At the end of the last chapter, the tests were telling us we weren't able to save the user's input. For now, we'll use a standard HTML POST request. A little boring, but also nice and easy to deliver—we can use all sorts of sexy HTML5 and JavaScript later in the book.

To get our browser to send a POST request, we give the `<input>` element a `name=` attribute, wrap it in a `<form>` tag with `method="POST"`, and the browser will take care of

sending the POST request to the server for us. Let's adjust our template at *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">
```

Now, running our FTs gives us a slightly cryptic, unexpected error:

```
$ python3 functional_tests.py
[...]
Traceback (most recent call last):
  File "functional_tests.py", line 39, 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: {"method":"id","selector":"id_list_table"}' ; Stacktrace [...]
```

When a functional test fails with an unexpected failure, there are several things we can do to debug them:

- Add `print` statements, to show, eg, what the current page text is.
- Improve the *error message* to show more info about the current state.
- Manually visit the site yourself.
- Use `time.sleep` to pause the test during execution.

We'll look at all of these over the course of this book, but the `time.sleep` option is one I find myself using very often. Let's try it now. We add the sleep just before the error occurs:

functional_tests.py.

```
        # When she hits enter, the page updates, and now the page lists
        # "1: Buy peacock feathers" as an item in a to-do list table
        inputbox.send_keys(Keys.ENTER)

        import time
        time.sleep(10)
        table = self.browser.find_element_by_id('id_list_table')
```

Depending on how fast Selenium runs on your PC, you may have caught a glimpse of this already, but when we run the functional tests again, we've got time to see what's going on: you should see a page that looks like Figure 5-1, with lots of Django debug information.

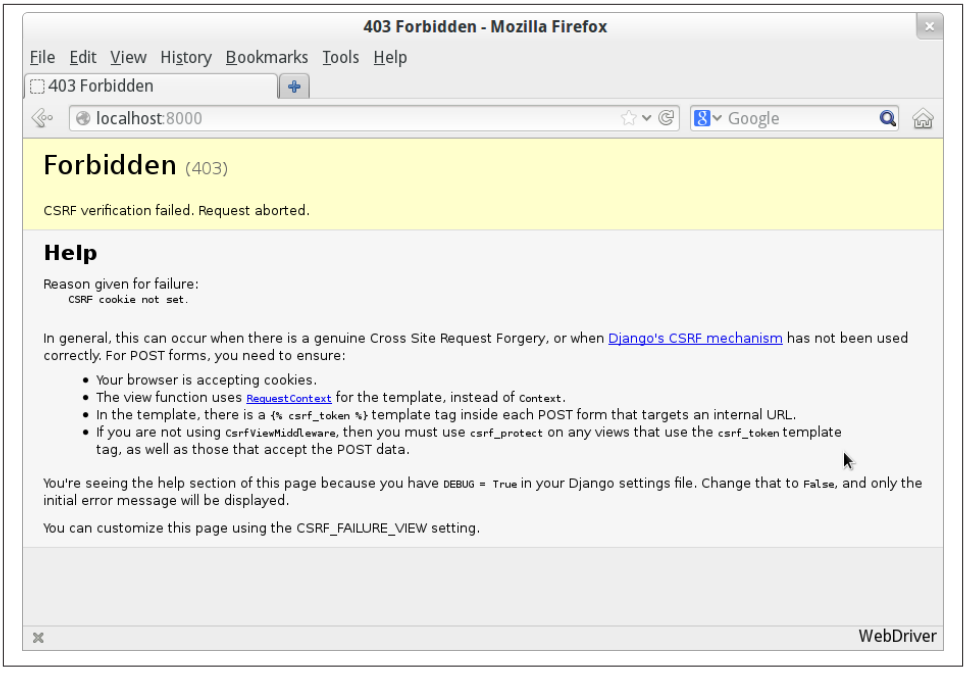


Figure 5-1. Django DEBUG page showing CSRF error

Security: Surprisingly Fun!

If you've never heard of a *Cross-Site Request Forgery* exploit, why not look it up now? Like all security exploits, it's entertaining to read about, being an ingenious use of a system in unexpected ways…

When I went back to university to get my Computer Science degree, I signed up for the Security module out of a sense of duty: *Oh well, it'll probably be very dry and boring, but I suppose I'd better take it*. It turned out to be one of the most fascinating modules of the whole course—absolutely full of the joy of hacking, of the particular mindset it takes to think about how systems can be used in unintended ways.

I want to recommend the textbook for my course, Ross Anderson's Security Engineering. It's quite light on pure crypto, but it's absolutely full of interesting discussions of unexpected topics like lock-picking, forging bank notes, inkjet printer cartridge economics, and spoofing South African Air Force jets with replay attacks. It's a huge tome, about three inches thick, and I promise you it's an absolute page-turner.

Django's CSRF protection involves placing a little auto-generated token into each generated form, to be able to identify POST requests as having come from the original site.

So far our template has been pure HTML, and in this step we make the first use of Django's template magic. To add the CSRF token we use a *template tag*, which has the curly-bracket/percent syntax, `{% … %}`—famous for being the world's most annoying two-key touch-typing combination:

lists/templates/home.html.

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

Django will substitute that during rendering with an `<input type="hidden">` containing the CSRF token. Rerunning the functional test will now give us an expected failure:

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

Since our `time.sleep` is still there, the test will pause on the final screen, showing us that the new item text disappears after the form is submitted, and the page refreshes to show an empty form again. That's because we haven't wired up our server to deal with the POST request yet—it just ignores it and displays the normal home page.

We can remove the `time.sleep` now though:

functional_tests.py.

```
        # "1: Buy peacock feathers" as an item in a to-do list table
        inputbox.send_keys(Keys.ENTER)

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

Processing a POST Request on the Server

Because we haven't specified an `action=` attribute in the form, it is submitting back to the same URL it was rendered from by default (ie, `/`), which is dealt with by our `home_page` function. Let's adapt the view to be able to deal with a POST request.

That means a new unit test for the `home_page` view. Open up *lists/tests.py*, and add a new method to `HomePageTest`—I copied the previous method, then adapted it to add our POST request and check that the returned HTML will have the new item text in it:

lists/tests.py (ch05l005).

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


def test_home_page_can_save_a_POST_request(self):
    request = HttpRequest()
    request.method = 'POST'
    request.POST['item_text'] = 'A new list item'

    response = home_page(request)
```

```
    self.assertIn('A new list item', response.content.decode())
```

Are you wondering about the line spacing in the test? I'm grouping together three lines at the beginning which set up the test, one line in the middle which actually calls the function under test, and the assertions at the end. This isn't obligatory, but it does help see the structure of the test. Setup, Exercise, Assert is the typical structure for a unit test.

You can see that we're using a couple of special attributes of the `HttpRequest`: `.method` and `.POST` (they're fairly self-explanatory, although now might be a good time for a peek at the Django request and response documentation). Then we check that the text from our POST request ends up in the rendered HTML. That gives us our expected fail:

```
$ python3 manage.py test
[...]
AssertionError: 'A new list item' not found in '<html> [...]
```

We can get the test to pass by adding an `if` and providing a different code path for POST requests. In typical TDD style, we start with a deliberately silly return value:

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')
```

That gets our unit tests passing, but it's not really what we want. What we really want to do is add the POST submission to the table in the home page template.

Passing Python Variables to Be Rendered in the Template

We've already had a hint of it, and now it's time to start to get to know the real power of the Django template syntax, which is to pass variables from our Python view code into HTML templates.

Let's start by seeing how the template syntax lets us include a Python object in our template. The notation is `{{ ... }}`, which displays the object as a 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>
```

How can we test that our view is passing in the correct value for `new_item_text`? How do we pass a variable to a template? We can find out by actually doing it in the unit test—we've already used the `render_to_string` function in a previous unit test to manually render a template and compare it with the HTML the view returns. Now let's add the variable we want to pass in:

lists/tests.py.

```
        self.assertIn('A new list item', response.content.decode())
        expected_html = render_to_string(
            'home.html',
            {'new_item_text':  'A new list item'}
        )
        self.assertEqual(response.content.decode(), expected_html)
```

As you can see, the `render_to_string` function takes, as its second parameter, a mapping of variable names to values. We're giving the template a variable named `new_item_text`, whose value is the expected item text from our POST request.

When we run the unit test, `render_to_string` will substitute `{{ new_item_text }}` for *A new list item* inside the `<td>`. That's something the actual view isn't doing yet, so we should see a test failure:

```
        self.assertEqual(response.content.decode(), expected_html)
AssertionError: 'A new list item' != '<html>\n    <head>\n [...]
```

Good, our deliberately silly return value is now no longer fooling our tests, so we are allowed to rewrite our view, and tell it to pass the POST parameter to the template:

lists/views.py (ch05l009).

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

Running the unit tests again:

```
ERROR: test_home_page_returns_correct_html (lists.tests.HomePageTest)
[...]
    'new_item_text': request.POST['item_text'],
KeyError: 'item_text'
```

An *unexpected failure*.

If you remember the rules for reading tracebacks, you'll spot that it's actually a failure in a *different* test. We got the actual test we were working on to pass, but the unit tests have picked up an unexpected consequence, a regression: we broke the code path where there is no POST request.

This is the whole point of having tests. Yes, we could have predicted this would happen, but imagine if we'd been having a bad day or weren't paying attention: our tests have just saved us from accidentally breaking our application, and, because we're using TDD, we found out immediately. We didn't have to wait for a QA team, or switch to a web browser and click through our site manually, and we can get on with fixing it straight away. Here's how:

lists/views.py.

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

Look up `dict.get` if you're not sure what's going on there.

The unit tests should now pass. Let's see what the functional tests say:

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

Hmm, not a wonderfully helpful error. Let's use another of our FT debugging techniques: improving the error message. This is probably the most constructive technique, because those improved error messages stay around to help debug any future errors:

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 -- its text was:\n%s" % (
                table.text,
            )
        )
```

That gives us a more helpful error message:

```
AssertionError: False is not true : New to-do item did not appear in table --
its text was:
Buy peacock feathers
```

You know what could be even better than that? Making that assertion a bit less clever. As you may remember, I was very pleased with myself for using the `any` function, but one of my Early Release readers (thanks Jason!) suggested a much simpler implementation. We can replace all six lines of the `assertTrue` with a single `assertIn`:

functional_tests.py.

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

Much better. You should always be very worried whenever you think you're being clever, because what you're probably being is *overcomplicated*. And we get the error message for free:

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

Consider me suitably chastened. The point is that the FT wants us to enumerate list items with a "1:" at the beginning of the first list item. The fastest way to get that to pass is with a quick "cheating" change to the template:

lists/templates/home.html.

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

Red/Green/Refactor and Triangulation

The unit-test/code cycle is sometimes taught as *Red, Green, Refactor*:

- Start by writing a unit test which fails (*Red*).
- Write the simplest possible code to get it to pass (*Green*), *even if that means cheating*.
- *Refactor* to get to better code that makes more sense.

So what do we do during the Refactor stage? What justifies moving from an implementation where we "cheat" to one we're happy with?

One methodology is *eliminate duplication*: if your test uses a magic constant (like the "1:" in front of our list item), and your application code also uses it, that counts as duplication, so it justifies refactoring. Removing the magic constant from the application code usually means you have to stop cheating.

I find that leaves things a little too vague, so I usually like to use a second technique, which is called *triangulation*: if your tests let you get away with writing "cheating" code that you're not happy with, like returning a magic constant, *write another test* that forces you to write some better code. That's what we're doing when we extend the FT to check that we get a "2:" when inputting a *second* list item.

Now we get to the `self.fail('Finish the test!')`. If we extend our FT to check for adding a second item to the table (copy and paste is our friend), we begin to see that our first cut solution really isn't going to, um, cut it:

functional_tests.py.

```
        # There is still a text box inviting her to add another item. She
        # enters "Use peacock feathers to make a fly" (Edith is very
        # methodical)
        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)

        # The page updates again, and now shows both items on her list
        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 wonders whether the site will remember her list. Then she sees
        # that the site has generated a unique URL for her -- there is some
        # explanatory text to that effect.
        self.fail('Finish the test!')

        # She visits that URL - her to-do list is still there.
```

Sure enough, the functional tests return an error:

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

Three Strikes and Refactor

Before we go further—we've got a bad *code smell*[1] in this FT. We have three almost identical code blocks checking for new items in the list table. There's a principle called *don't repeat yourself* (DRY), which we like to apply by following the mantra *three strikes and refactor*. You can copy and paste code once, and it may be premature to try and remove the duplication it causes, but once you get three occurrences, it's time to remove duplication.

We start by committing what we have so far. Even though we know our site has a major flaw—it can only handle one list item—it's still further ahead than it was. We may have to rewrite it all, and we may not, but the rule is that before you do any refactoring, always do a commit:

```
$ git diff
# should show changes to functional_tests.py, home.html,
# tests.py and views.py
$ git commit -a
```

Back to our functional test refactor: we could use an inline function, but that upsets the flow of the test slightly. Let's use a helper method—remember, only methods that begin with `test_` will get run as tests, so you can use other methods for your own purposes:

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')
```

1. If you've not come across the concept, a "code smell" is something about a piece of code that makes you want to rewrite it. Jeff Atwood has a compilation on his blog Coding Horror. The more experience you gain as a programmer, the more fine-tuned your nose becomes to code smells…

```
        self.assertIn(row_text, [row.text for row in rows])


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

I like to put helper methods near the top of the class, between the `tearDown` and the first test. Let's use it in the FT:

functional_tests.py.

```
    # When she hits enter, the page updates, and now the page lists
    # "1: Buy peacock feathers" as an item in a to-do list table
    inputbox.send_keys(Keys.ENTER)
    self.check_for_row_in_list_table('1: Buy peacock feathers')

    # There is still a text box inviting her to add another item. She
    # enters "Use peacock feathers to make a fly" (Edith is very
    # methodical)
    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)

    # The page updates again, and now shows both items on her list
    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 wonders whether the site will remember her list. Then she sees
    [...]
```

We run the FT again to check that it still behaves in the same way…

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

Good. Now we can commit the FT refactor as its own small, atomic change:

```
$ git diff # check the changes to functional_tests.py
$ git commit -a
```

And back to work. If we're ever going to handle more than one list item, we're going to need some kind of persistence, and databases are a stalwart solution in this area.

The Django ORM and Our First Model

An *Object-Relational Mapper* (ORM) is a layer of abstraction for data stored in a database with tables, rows, and columns. It lets us work with databases using familiar object-oriented metaphors which work well with code. Classes map to database tables, attributes map to columns, and an individual instance of the class represents a row of data in the database.

Django comes with an excellent ORM, and writing a unit test that uses it is actually an excellent way of learning it, since it exercises code by specifying how we want it to work.

Let's create a new class in *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')
```

You can see that creating a new record in the database is a relatively simple matter of creating an object, assigning some attributes, and calling a `.save()` function. Django also gives us an API for querying the database via a class attribute, `.objects`, and we use the simplest possible query, `.all()`, which retrieves all the records for that table. The results are returned as a list-like object called a `QuerySet`, from which we can extract individual objects, and also call further functions, like `.count()`. We then check the objects as saved to the database, to check whether the right information was saved.

Django's ORM has many other helpful and intuitive features; this might be a good time to skim through the Django tutorial, which has an excellent intro to them.

I've written this unit test in a very verbose style, as a way of introducing the Django ORM. You can actually write a much shorter test for a model class, which we'll see later on, in Chapter 11.

Terminology 2: Unit Tests Versus Integrated Tests, and the Database

Purists will tell you that a "real" unit test should never touch the database, and that the test I've just written should be more properly called an integrated test, because it doesn't only test our code, but also relies on an external system, ie a database.

It's OK to ignore this distinction for now—we have two types of test, the high-level functional tests which test the application from the user's point of view, and these lower-level tests which test it from the programmer's point of view.

We'll come back to this and talk about unit tests and integrated tests in Chapter 19, towards the end of the book.

Let's try running the unit test. Here comes another unit-test/code cycle:

```
ImportError: cannot import name 'Item'
```

Very well, let's give it something to import from *lists/models.py*. We're feeling confident so we'll skip the `Item = None` step, and go straight to creating a class:

lists/models.py.

```
from django.db import models

class Item(object):
    pass
```

That gets our test as far as:

```
    first_item.save()
AttributeError: 'Item' object has no attribute 'save'
```

To give our `Item` class a `save` method, and to make it into a real Django model, we make it inherit from the `Model` class:

lists/models.py.

```
from django.db import models

class Item(models.Model):
    pass
```

Our First Database Migration

The next thing that happens is a database error:

```
first_item.save()
  File "/usr/local/lib/python3.4/dist-packages/django/db/models/base.py", line
593, in save
[...]
    return Database.Cursor.execute(self, query, params)
django.db.utils.OperationalError: no such table: lists_item
```

In Django, the ORM's job is to model the database, but there's a second system that's in charge of actually building the database called *migrations*. Its job is to give you the ability to add and remove tables and columns, based on changes you make to your *models.py* files.

One way to think of it is as a version control system for your database. As we'll see later, it comes in particularly useful when we need to upgrade a database that's deployed on a live server.

For now all we need to know is how to build our first database migration, which we do using the `makemigrations` command:

```
$ python3 manage.py makemigrations
Migrations for 'lists':
  0001_initial.py:
    - Create model Item
$ ls lists/migrations
0001_initial.py  __init__.py  __pycache__
```

If you're curious, you can go and take a look in the migrations file, and you'll see it's a representation of our additions to *models.py*.

In the meantime, we should find our tests get a little further.

The Test Gets Surprisingly Far

The test actually gets surprisingly far:

```
$ python3 manage.py test lists
[...]
    self.assertEqual(first_saved_item.text, 'The first (ever) list item')
AttributeError: 'Item' object has no attribute 'text'
```

That's a full eight lines later than the last failure—we've been all the way through saving the two Items, we've checked they're saved in the database, but Django just doesn't seem to have remembered the `.text` attribute.

Incidentally, if you're new to Python, you might have been surprised we were allowed to assign the `.text` attribute at all. In something like Java, that would probably give you a compilation error. Python is more relaxed about things like that.

Classes that inherit from `models.Model` map to tables in the database. By default they get an auto-generated `id` attribute, which will be a primary key column in the database, but you have to define any other columns you want explicitly. Here's how we set up a text field:

lists/models.py.

```
class Item(models.Model):
    text = models.TextField()
```

Django has many other field types, like `IntegerField`, `CharField`, `DateField`, and so on. I've chosen `TextField` rather than `CharField` because the latter requires a length restriction, which seems arbitrary at this point. You can read more on field types in the Django tutorial and in the documentation.

A New Field Means a New Migration

Running the tests gives us another database error:

```
django.db.utils.OperationalError: table lists_item has no column named text
```

It's because we've added another new field to our database, which means we need to create another migration. Nice of our tests to let us know!

Let's try it:

```
$ python3 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)
 2) Quit, and let me add a default in models.py
Select an option:2
```

Ah. It won't let us add the column without a default value. Let's pick option 2 and set a default in *models.py*. I think you'll find the syntax reasonably self-explanatory:

lists/models.py.

```
class Item(models.Model):
    text = models.TextField(default='')
```

And now the migration should complete:

```
$ python3 manage.py makemigrations
Migrations for 'lists':
  0002_item_text.py:
    - Add field text to item
```

So, two new lines in *models.py*, two database migrations, and as a result, the `.text` attribute on our model objects is now recognised as a special attribute, so it does get saved to the database, and the tests pass…

```
$ python3 manage.py test lists
[...]

Ran 4 tests in 0.010s
OK
```

So let's do a commit for our first ever model!

```
$ git status # see tests.py, models.py, and 2 untracked migrations
$ git diff # review changes to tests.py and models.py
$ git add lists
$ git commit -m"Model for list Items and associated migration"
```

Saving the POST to the Database

Let's adjust the test for our home page POST request, and say we want the view to save a new item to the database instead of just passing it through to its response. We can do that by adding three new lines to the existing test called `test_home_page_can_save_a_POST_request`:

lists/tests.py

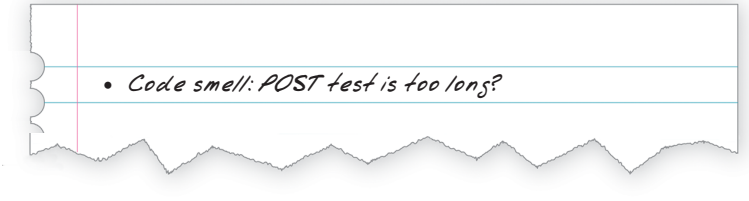
```
    def test_home_page_can_save_a_POST_request(self):
        request = HttpRequest()
        request.method = 'POST'
        request.POST['item_text'] = 'A new list item'

        response = home_page(request)

        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())
        expected_html = render_to_string(
            'home.html',
            {'new_item_text':  'A new list item'}
        )
        self.assertEqual(response.content.decode(), expected_html)
```

❶ We check that one new `Item` has been saved to the database. `objects.count()` is a shorthand for `objects.all().count()`.

❷ `objects.first()` is the same as doing `objects.all()[0]`.

❸ We check that the item's text is correct.

This test is getting a little long-winded. It seems to be testing lots of different things. That's another *code smell*—a long unit test either needs to be broken into two, or it may be an indication that the thing you're testing is too complicated. Let's add that to a little to-do list of our own, perhaps on a piece of scrap paper:



Writing it down on a scratchpad like this reassures us that we won't forget, so we are comfortable getting back to what we were working on. We rerun the tests and see an expected failure:

```
        self.assertEqual(Item.objects.count(), 1)
    AssertionError: 0 != 1
```

Let's adjust our 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', ''),
    })
```

I've coded a very naive solution and you can probably spot a very obvious problem, which is that we're going to be saving empty items with every request to the home page. Let's add that to our list of things to fix later. You know, along with the painfully obvious fact that we currently have no way at all of having different lists for different people. That we'll keep ignoring for now.

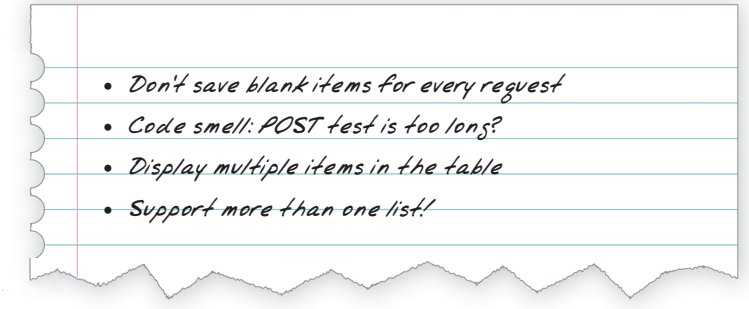
Remember, I'm not saying you should always ignore glaring problems like this in "real life". Whenever we spot problems in advance, there's a judgement call to make over whether to stop what you're doing and start again, or leave them until later. Sometimes finishing off what you're doing is still worth it, and sometimes the problem may be so major as to warrant a stop and rethink.

Let's see how the unit tests get on … they pass! Good. We can do a bit of refactoring:

lists/views.py.

```
    return render(request, 'home.html', {
        'new_item_text': item.text
    })
```

Let's have a little look at our scratchpad. I've added a couple of the other things that are on our mind:



Let's start with the first one. We could tack on an assertion to an existing test, but it's best to keep unit tests to testing one thing at a time, so let's add a new one:

lists/tests.py.

```
class HomePageTest(TestCase):
    [...]

    def test_home_page_only_saves_items_when_necessary(self):
        request = HttpRequest()
        home_page(request)
        self.assertEqual(Item.objects.count(), 0)
```

That gives us a `1 != 0` failure. Let's fix it. Watch out; although it's quite a small change to the logic of the view, there are quite a few little tweaks to the implementation in code:

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 = ''  #❸

    return render(request, 'home.html', {
        'new_item_text': new_item_text,  #❹
    })
```

❶ ❸ ❹ We use a variable called `new_item_text`, which will either hold the POST contents, or the empty string.

❷ `.objects.create` is a neat shorthand for creating a new `Item`, without needing to call `.save()`.

And that gets the test passing:

```
Ran 5 tests in 0.010s

OK
```

Redirect After a POST

But, yuck, that whole `new_item_text = ''` dance is making me pretty unhappy. Thankfully the next item on the list gives us a chance to fix it. Always redirect after a POST, they say, so let's do that. Once again we change our unit test for saving a POST request to say that, instead of rendering a response with the item in it, it should redirect back to the home page:

lists/tests.py.

```
    def test_home_page_can_save_a_POST_request(self):
        request = HttpRequest()
        request.method = 'POST'
        request.POST['item_text'] = 'A new list item'

        response = home_page(request)

        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'], '/')
```

We no longer expect a response with a `.content` rendered by a template, so we lose the assertions that look at that. Instead, the response will represent an HTTP *redirect*, which should have status code 302, and points the browser towards a new location.

That gives us the error `200 != 302`. We can now tidy up our view substantially:

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')
```

And the tests should now pass:

```
Ran 5 tests in 0.010s

OK
```

Better Unit Testing Practice: Each Test Should Test One Thing

Our view now does a redirect after a POST, which is good practice, and we've shortened the unit test somewhat, but we can still do better. Good unit testing practice says that each test should only test one thing. The reason is that it makes it easier to track down bugs. Having multiple assertions in a test means that, if the test fails on an early assertion,

you don't know what the status of the later assertions is. As we'll see in the next chapter, if we ever break this view accidentally, we want to know whether it's the saving of objects that's broken, or the type of response.

You may not always write perfect unit tests with single assertions on your first go, but now feels like a good time to separate out our concerns:

lists/tests.py.

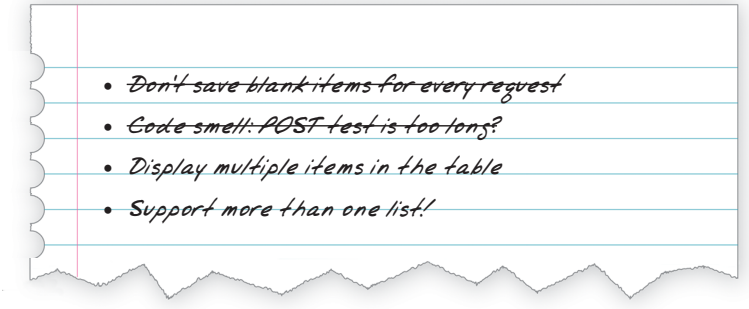
```
    def test_home_page_can_save_a_POST_request(self):
        request = HttpRequest()
        request.method = 'POST'
        request.POST['item_text'] = 'A new list item'

        response = home_page(request)

        self.assertEqual(Item.objects.count(), 1)
        new_item = Item.objects.first()
        self.assertEqual(new_item.text, 'A new list item')


    def test_home_page_redirects_after_POST(self):
        request = HttpRequest()
        request.method = 'POST'
        request.POST['item_text'] = 'A new list item'

        response = home_page(request)

        self.assertEqual(response.status_code, 302)
        self.assertEqual(response['location'], '/')
```

And we should now see six tests pass instead of five:

```
Ran 6 tests in 0.010s

OK
```

Rendering Items in the Template

Much better! Back to our to-do list:



Crossing things off the list is almost as satisfying as seeing tests pass!

The third item is the last of the "easy" ones. Let's have a new unit test that checks that the template can also display multiple list items:

lists/tests.py.

```
class HomePageTest(TestCase):
    [...]

    def test_home_page_displays_all_list_items(self):
        Item.objects.create(text='itemey 1')
        Item.objects.create(text='itemey 2')

        request = HttpRequest()
        response = home_page(request)

        self.assertIn('itemey 1', response.content.decode())
        self.assertIn('itemey 2', response.content.decode())
```

That fails as expected:

```
AssertionError: 'itemey 1' not found in '<html>\n    <head>\n [...]
```

The Django template syntax has a tag for iterating through lists, `{% for .. in .. %}`; we can use it like this:

lists/templates/home.html.

```
<table id="id_list_table">
    {% for item in items %}
        <tr><td>1: {{ item.text }}</td></tr>
    {% endfor %}
</table>
```

This is one of the major strengths of the templating system. Now the template will render with multiple `<tr>` rows, one for each item in the variable `items`. Pretty neat! I'll introduce a few more bits of Django template magic as we go, but at some point you'll want to go and read up on the rest of them in the Django docs.

Just changing the template doesn't get our tests to pass; we need to actually pass the items to it from our home page view:

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})
```

That does get the unit tests to pass … moment of truth, will the functional test pass?

```
$ python3 functional_tests.py
[...]
AssertionError: 'To-Do' not found in 'OperationalError at /'
```

Oops, apparently not. Let's use another functional test debugging technique, and it's one of the most straightforward: manually visiting the site! Open up *http://localhost:8000* in your web browser, and you'll see a Django debug page saying "no such table: lists_item", as in Figure 5-2.

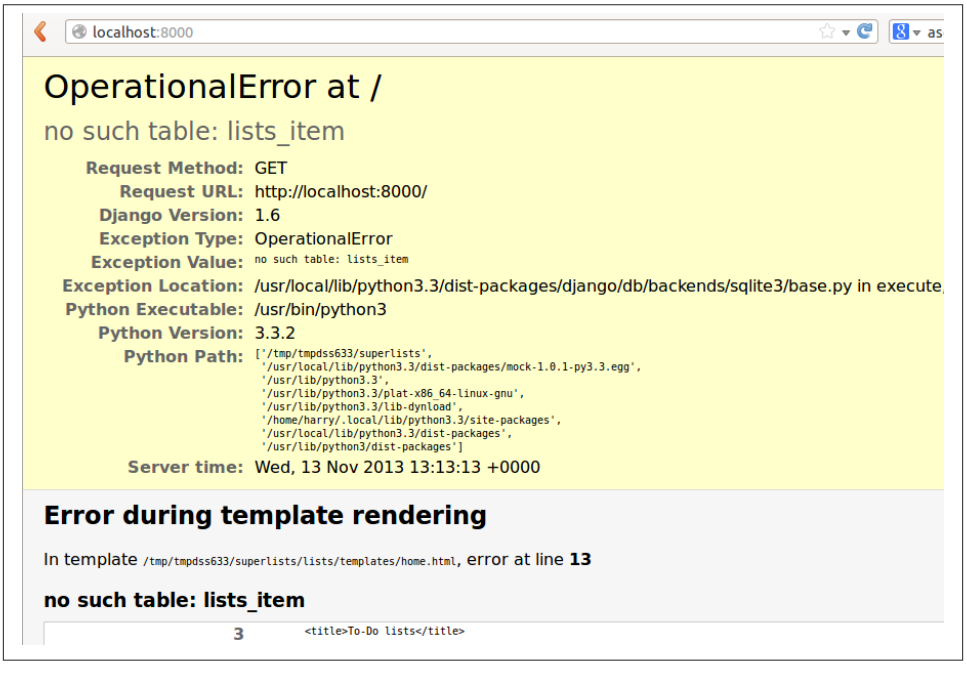


Figure 5-2. Another helpful debug message

Creating Our Production Database with migrate

Another helpful error message from Django, which is basically complaining that we haven't set up the database properly. How come everything worked fine in the unit tests,

I hear you ask? Because Django creates a special *test database* for unit tests; it's one of the magical things that Django's `TestCase` does.

To set up our "real" database, we need to create it. SQLite databases are just a file on disk, and you'll see in *settings.py* that Django, by default, will just put it in a file called *db.sqlite3* in the base project directory:

superlists/settings.py.

```
[...]
# Database
# https://docs.djangoproject.com/en/1.7/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

We've told Django everything it needs to create the database, first via *models.py* and then when we created the migrations file. To actually apply it to creating a real database, we use another Django Swiss Army knife *manage.py* command, `migrate`:

```
$ python3 manage.py migrate
Operations to perform:
  Synchronize unmigrated apps: contenttypes, sessions, admin, auth
  Apply all migrations: lists
Synchronizing apps without migrations:
  Creating tables...
    Creating table django_admin_log
    Creating table auth_permission
    Creating table auth_group_permissions
    Creating table auth_group
    Creating table auth_user_groups
    Creating table auth_user_user_permissions
    Creating table auth_user
    Creating table django_content_type
    Creating table django_session
  Installing custom SQL...
  Installing indexes...
Running migrations:
  Applying lists.0001_initial... OK
  Applying lists.0002_item_text... OK

You have installed Django's auth system, and don't have any superusers defined.
Would you like to create one now? (yes/no):
no
```

I said "no" to the question about superusers—we don't need one yet, but we will look at it in a later chapter. For now we can refresh the page on *localhost*, see that our error is gone, and try running the functional tests again:[2]

```
AssertionError: '2: Use peacock feathers to make a fly' not found in ['1: Buy
peacock feathers', '1: Use peacock feathers to make a fly']
```

So close! We just need to get our list numbering right. Another awesome Django template tag, `forloop.counter`, will help here:

lists/templates/home.html.

```
        {% for item in items %}
            <tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>
        {% endfor %}
```

If you try it again, you should now see the FT get to the end:

```
        self.fail('Finish the test!')
    AssertionError: Finish the test!
```

But, as it's running, you may notice something is amiss, like in Figure 5-3.

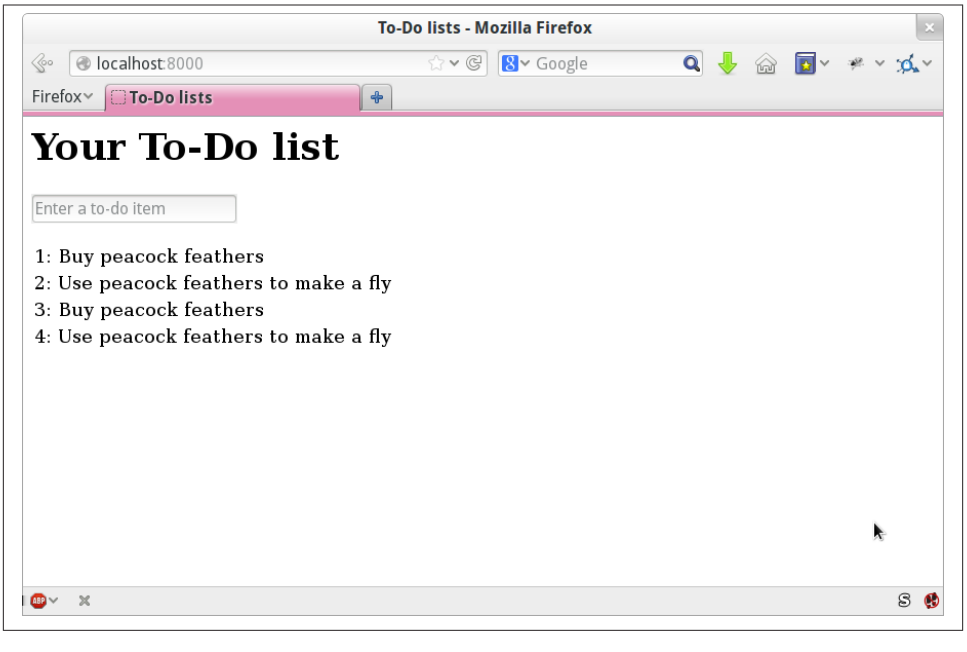


Figure 5-3. There are list items left over from the last run of the test

2. If you get a different error at this point, try restarting your dev server—it may have gotten confused by the changes to the database happening under its feet.

Oh dear. It looks like previous runs of the test are leaving stuff lying around in our database. In fact, if you run the tests again, you'll see it gets worse:

```
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. We're so close! We're going to need some kind of automated way of tidying up after ourselves. For now, if you feel like it, you can do it manually, by deleting the database and re-creating it fresh with `migrate`:

```
$ rm db.sqlite3
$ python3 manage.py migrate --noinput
```

And then reassure yourself that the FT still passes.

Apart from that little bug in our functional testing, we've got some code that's more or less working. Let's do a commit.

Start by doing a **`git status`** and a **`git diff`**, and you should see changes to *home.html*, *tests.py*, and *views.py*. Let's add them:

```
$ git add lists
$ git commit -m"Redirect after POST, and show all items in template"
```

You might find it useful to add markers for the end of each chapter, like **`git tag end-of-chapter-05`**.

Where are we?

- We've got a form set up to add new items to the list using POST.
- We've set up a simple model in the database to save list items.
- We've used at least three different FT debugging techniques.

But we've got a couple of items on our own to-do list, namely getting the FT to clean up after itself, and perhaps more critically, adding support for more than one list.

I mean, we *could* ship the site as it is, but people might find it strange that the entire human population has to share a single to-do list. I suppose it might get people to stop and think about how connected we all are to one another, how we all share a common destiny here on Spaceship Earth, and how we must all work together to solve the global problems that we face.

But in practical terms, the site wouldn't be very useful.

Ah well.

Useful TDD Concepts

Regression
: When new code breaks some aspect of the application which used to work.

Unexpected failure
: When a test fails in a way we weren't expecting. This either means that we've made a mistake in our tests, or that the tests have helped us find a regression, and we need to fix something in our code.

Red/Green/Refactor
: Another way of describing the TDD process. Write a test and see it fail (Red), write some code to get it to pass (Green), then Refactor to improve the implementation.

Triangulation
: Adding a test case with a new specific example for some existing code, to justify generalising the implementation (which may be a "cheat" until that point).

Three strikes and refactor
: A rule of thumb for when to remove duplication from code. When two pieces of code look very similar, it often pays to wait until you see a third use case, so that you're more sure about what part of the code really is the common, re-usable part to refactor out.

The scratchpad to-do list
: A place to write down things that occur to us as we're coding, so that we can finish up what we're doing and come back to them later.

CHAPTER 6
Getting to the Minimum Viable Site

In this chapter we're going to address the problems we discovered at the end of the last chapter. In the immediate, the problem of cleaning up after functional test runs. Later, the more general problem, which is that our design only allows for one global list. I'll demonstrate a critical TDD technique: how to adapt existing code using an incremental, step-by-step process which takes you from working code to working code. Testing Goat, not Refactoring Cat.

Ensuring Test Isolation in Functional Tests

We ended the last chapter with a classic testing problem: how to ensure *isolation* between tests. Each run of our functional tests was leaving list items lying around in the database, and that would interfere with the test results when you next ran the tests.

When we run *unit* tests, the Django test runner automatically creates a brand new test database (separate from the real one), which it can safely reset before each individual test is run, and then throw away at the end. But our functional tests currently run against the "real" database, *db.sqlite3*.

One way to tackle this would be to "roll our own" solution, and add some code to *functional_tests.py* which would do the cleaning up. The `setUp` and `tearDown` methods are perfect for this sort of thing.

Since Django 1.4 though, there's a new class called `LiveServerTestCase` which can do this work for you. It will automatically create a test database (just like in a unit test run), and start up a development server for the functional tests to run against. Although as a tool it has some limitations which we'll need to work around later, it's dead useful at this stage, so let's check it out.

`LiveServerTestCase` expects to be run by the Django test runner using *manage.py*. As of Django 1.6, the test runner will find any files whose name begins with *test*. To keep

things neat and tidy, let's make a folder for our functional tests, so that it looks a bit like an app. All Django needs is for it to be a valid Python module (ie, one with a *__init__.py* in it):

```
$ mkdir functional_tests
$ touch functional_tests/__init__.py
```

Then we *move* our functional tests, from being a standalone file called *functional_tests.py*, to being the *tests.py* of the `functional_tests` app. We use **`git mv`** so that Git notices that we've moved the file:

```
$ git mv functional_tests.py functional_tests/tests.py
$ git status # shows the rename to functional_tests/tests.py and __init__.py
```

At this point your directory tree should look like this:

```
.
├── db.sqlite3
├── functional_tests
│   ├── __init__.py
│   └── tests.py
├── lists
│   ├── admin.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 is gone, and has turned into *functional_tests/tests.py*. Now, whenever we want to run our functional tests, instead of running `python3 functional_tests.py`, we will use `python3 manage.py test functional_tests`.

You could mix your functional tests into the tests for the `lists` app. I tend to prefer to keep them separate, because functional tests usually have cross-cutting concerns that run across different apps. FTs are meant to see things from the point of view of your users, and your users don't care about how you've split work between different apps!

Now let's edit *functional_tests/tests.py* and change our `NewVisitorTest` class to make it use `LiveServerTestCase`:

functional_tests/tests.py (ch06l001).

```
from django.test import LiveServerTestCase
from selenium import webdriver
from selenium.webdriver.common.keys import Keys

class NewVisitorTest(LiveServerTestCase):

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

Next,[1] instead of hardcoding the visit to localhost port 8000, `LiveServerTestCase` gives us an attribute called `live_server_url`:

functional_tests/tests.py (ch06l002).

```
    def test_can_start_a_list_and_retrieve_it_later(self):
        # Edith has heard about a cool new online to-do app. She goes
        # to check out its homepage
        self.browser.get(self.live_server_url)
```

We can also remove the `if __name__ == '__main__'` from the end if we want, since we'll be using the Django test runner to launch the FT.

Now we are able to run our functional tests using the Django test runner, by telling it to run just the tests for our new `functional_tests` app:

```
$ python3 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 "/workspace/superlists/functional_tests/tests.py", line 61, in
test_can_start_a_list_and_retrieve_it_later
    self.fail('Finish the test!')
AssertionError: Finish the test!
```

1. Are you unable to move on because you're wondering what those *ch06l0xx* things are, next to some of the code listings? They refer to specific commits in the book's example repo. It's all to do with my book's correctness tests. You know, the tests for the tests in the book about testing. They have tests of their own, incidentally.

```
 ----------------------------------------------------------------------
Ran 1 test in 6.378s

FAILED (failures=1)
Destroying test database for alias 'default'...
```

The FT gets through to the `self.fail`, just like it did before the refactor. You'll also notice that if you run the tests a second time, there aren't any old list items lying around from the previous test—it has cleaned up after itself. Success! We should commit it as an atomic change:

```
$ git status # functional_tests.py renamed + modified, new __init__.py
$ git add functional_tests
$ git diff --staged -M
$ git commit  # msg eg "make functional_tests an app, use LiveServerTestCase"
```

The `-M` flag on the `git diff` is a useful one. It means "detect moves", so it will notice that *functional_tests.py* and *functional_tests/tests.py* are the same file, and show you a more sensible diff (try it without the flag!).

Running Just the Unit Tests

Now if we run `manage.py test`, Django will run both the functional and the unit tests:

```
$ python3 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 8 tests in 3.132s

FAILED (failures=1)
Destroying test database for alias 'default'...
```

In order to run just the unit tests, we can specify that we want to only run the tests for the `lists` app:

```
$ python3 manage.py test lists
Creating test database for alias 'default'...
.......
 ----------------------------------------------------------------------
Ran 7 tests in 0.009s

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

Useful Commands Updated

To run the functional tests

```
python3 manage.py test functional_tests
```

To run the unit tests

```
python3 manage.py test lists
```

What to do if I say "run the tests", and you're not sure which ones I mean? Have another look at the flowchart at the end of Chapter 4, and try and figure out where we are. As a rule of thumb, we usually only run the functional tests once all the unit tests are passing, so if in doubt, try both!

Now let's move on to thinking about how we want support for multiple lists to work. Currently the FT (which is the closest we have to a design document) says this:

functional_tests/tests.py.

```
        # Edith wonders whether the site will remember her list. Then she sees
        # that the site has generate a unique URL for her -- there is some
        # explanatory text to that effect.
        self.fail('Finish the test!')

        # She visits that URL - her to-do list is still there.

        # Satisfied, she goes back to sleep
```

But really we want to expand on this, by saying that different users don't see each other's lists, and each get their own URL as a way of going back to their saved lists. Let's think about this a bit more.

Small Design When Necessary

TDD is closely associated with the agile movement in software development, which includes a reaction against *Big Design Up Front* the traditional software engineering practice whereby, after a lengthy requirements gathering exercise, there is an equally lengthy design stage where the software is planned out on paper. The agile philosophy is that you learn more from solving problems in practice than in theory, especially when you confront your application with real users as soon as possible. Instead of a long up-front design phase, we try and put a *minimum viable application* out there early, and let the design evolve gradually based on feedback from real-world usage.

But that doesn't mean that thinking about design is outright banned! In the last chapter we saw how just blundering ahead without thinking can *eventually* get us to the right answer, but often a little thinking about design can help us get there faster. So, let's think about our minimum viable lists app, and what kind of design we'll need to deliver it.

- We want each user to be able to store their own list—at least one, for now.
- A list is made up of several items, whose primary attribute is a bit of descriptive text.
- We need to save lists from one visit to the next. For now, we can give each user a unique URL for their list. Later on we may want some way of automatically recognising users and showing them their lists.

To deliver the "for now" items, it sounds like we're going to store lists and their items in a database. Each list will have a unique URL, and each list item will be a bit of descriptive text, associated with a particular list.

YAGNI!

Once you start thinking about design, it can be hard to stop. All sorts of other thoughts are occurring to us—we might want to give each list a name or title, we might want to recognise users using usernames and passwords, we might want to add a longer notes field as well as short descriptions to our list, we might want to store some kind of ordering, and so on. But we obey another tenet of the agile gospel: "YAGNI" (pronounced yag-knee), which stands for "You aint gonna need it!" As software developers, we have fun creating things, and sometimes it's hard to resist the urge to build things just because an idea occurred to us and we *might* need it. The trouble is that more often than not, no matter how cool the idea was, you *won't* end up using it. Instead you have a load of unused code, adding to the complexity of your application. YAGNI is the mantra we use to resist our overenthusiastic creative urges.

REST

We have an idea of the data structure we want—the Model part of Model-View-Controller (MVC). What about the View and Controller parts? How should the user interact with Lists and their Items using a web browser?

Representational State Transfer (REST) is an approach to web design that's usually used to guide the design of web-based APIs. When designing a user-facing site, it's not possible to stick *strictly* to the REST rules, but they still provide some useful inspiration.

REST suggests that we have a URL structure that matches our data structure, in this case lists and list items. Each list can have its own URL:

```
/lists/<list identifier>/
```

That will fulfill the requirement we've specified in our FT. To view a list, we use a GET request (a normal browser visit to the page).

To create a brand new list, we'll have a special URL that accepts POST requests:

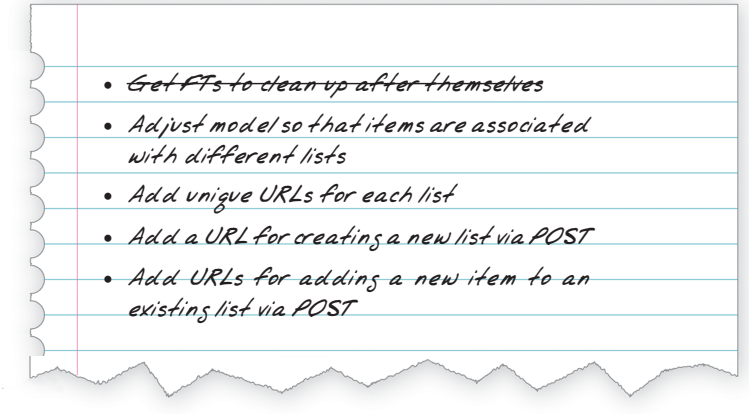
```
/lists/new
```

To add a new item to an existing list, we'll have a separate URL, to which we can send POST requests:

```
/lists/<list identifier>/add_item
```

(Again, we're not trying to perfectly follow the rules of REST, which would use a PUT request here—we're just using REST for inspiration.)

In summary, our scratchpad for this chapter looks something like this:



Implementing the New Design Using TDD

How do we use TDD to implement the new design? Let's take another look at the flowchart for the TDD process in Figure 6-1.

At the top level, we're going to use a combination of adding new functionality (by extending the FT and writing new application code), and refactoring our application—ie, rewriting some of the existing implementation so that it delivers the same functionality to the user but using aspects of our new design. At the unit test level, we'll be adding new tests or modifying existing ones to test for the changes we want, and we'll be able to use the untouched unit tests to make sure we don't break anything in the process.

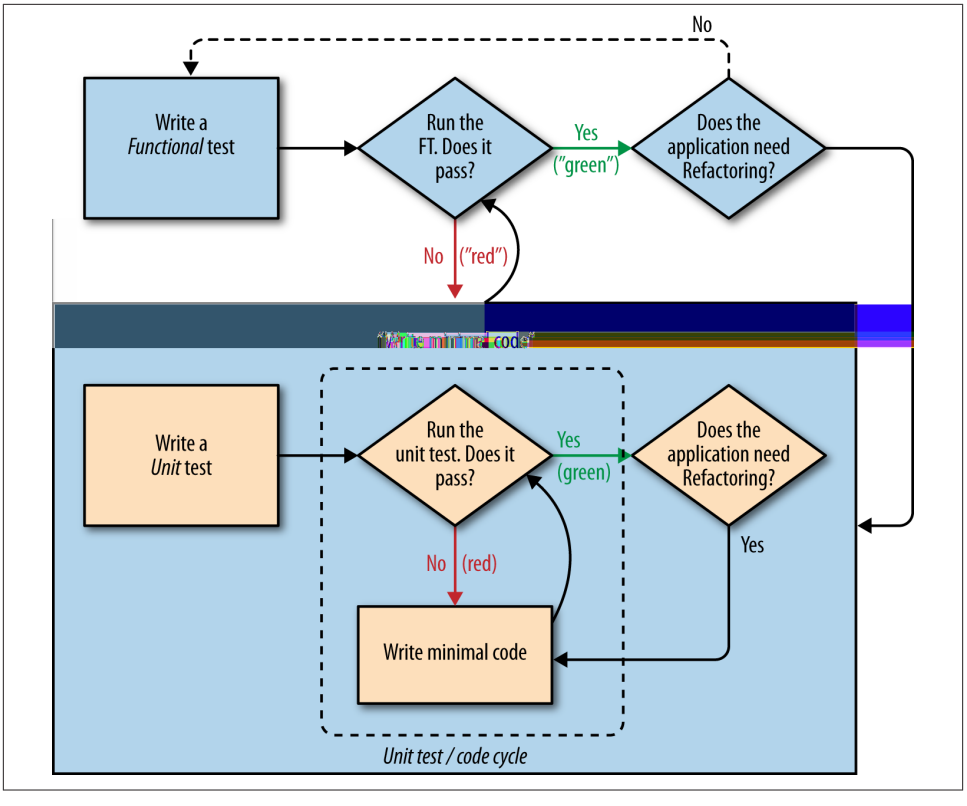


Figure 6-1. The TDD process with functional and unit tests

Let's translate our scratchpad into our functional test. As soon as Edith submits a first list item, we'll want to create a new list, adding one item to it, and take her to the URL for her list. Look for the point at which we say `inputbox.send_keys('Buy peacock feathers')`, and amend the next block of code like this:

functional_tests/tests.py.

```
        inputbox.send_keys('Buy peacock feathers')

        # When she hits enter, she is taken to a new URL,
        # and now the page lists "1: Buy peacock feathers" as an item in a
        # to-do list table
        inputbox.send_keys(Keys.ENTER)
        edith_list_url = self.browser.current_url
        self.assertRegex(edith_list_url, '/lists/.+') #❶
        self.check_for_row_in_list_table('1: Buy peacock feathers')

        # There is still a text box inviting her to add another item. She
        [...]
```

❶ `assertRegex` is a helper function from `unittest` that checks whether a string matches a regular expression. We use it to check that our new REST-ish design has been implemented. Find out more in the `unittest` documentation.

Let's also change the end of the test and imagine a new user coming along. We want to check that they don't see any of Edith's items when they visit the home page, and that they get their own unique URL for their list.

Delete everything from the comments just before the `self.fail` (they say "Edith wonders whether the site will remember her list …") and replace them with a new ending to our FT:

functional_tests/tests.py.

```
[...]
# The page updates again, and now shows both items on her list
self.check_for_row_in_list_table('2: Use peacock feathers to make a fly')
self.check_for_row_in_list_table('1: Buy peacock feathers')

# Now a new user, Francis, comes along to the site.

## We use a new browser session to make sure that no information
## of Edith's is coming through from cookies etc #❶
self.browser.quit()
self.browser = webdriver.Firefox()

# Francis visits the home page.  There is no sign of Edith's
# list
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 starts a new list by entering a new item. He
# is less interesting than Edith...
inputbox = self.browser.find_element_by_id('id_new_item')
inputbox.send_keys('Buy milk')
inputbox.send_keys(Keys.ENTER)

# Francis gets his own unique URL
francis_list_url = self.browser.current_url
self.assertRegex(francis_list_url, '/lists/.+')
self.assertNotEqual(francis_list_url, edith_list_url)

# Again, there is no trace of Edith's list
page_text = self.browser.find_element_by_tag_name('body').text
self.assertNotIn('Buy peacock feathers', page_text)
self.assertIn('Buy milk', page_text)

# Satisfied, they both go back to sleep
```

❶ I'm using the convention of double-hashes (##) to indicate "meta-comments"—comments about *how* the test is working and why—so that we can distinguish them from regular comments in FTs which explain the User Story. They're a message to our future selves, which might otherwise be wondering why the heck we're quitting the browser and starting a new one…

Other than that, the changes are fairly self-explanatory. Let's see how they do when we run our FTs:

```
AssertionError: Regex didn't match: '/lists/.+' not found in
'http://localhost:8081/'
```

As expected. Let's do a commit, and then go and build some new models and views:

```
$ git commit -a
```

I found the FTs hung when I tried to run them today. It turns out I needed to upgrade Selenium, with a `pip3 install --upgrade selenium`. You may remember from the preface that it's important to have the latest version of Selenium installed—it's only been a couple of months since I last upgraded, and Selenium had gone up by six point versions. If something weird is happening, always try upgrading Selenium!

Iterating Towards the New Design

Being all excited about our new design, I had an overwhelming urge to dive in at this point and start changing *models.py*, which would have broken half the unit tests, and then pile in and change almost every single line of code, all in one go. That's a natural urge, and TDD, as a discipline, is a constant fight against it. Obey the Testing Goat, not Refactoring Cat! We don't need to implement our new, shiny design in a single big bang. Let's make small changes that take us from a working state to a working state, with our design guiding us gently at each stage.

There are four items on our to-do list. The FT, with its `Regexp didn't match`, is telling us that the second item—giving lists their own URL and identifier—is the one we should work on next. Let's have a go at fixing that, and only that.

The URL comes from the redirect after POST. In *lists/tests.py*, find `test_home_page_redirects_after_POST`, and change the expected redirect location:

lists/tests.py

```
    self.assertEqual(response.status_code, 302)
    self.assertEqual(response['location'], '/lists/the-only-list-in-the-world/')
```

Does that seem slightly strange? Clearly, */lists/the-only-list-in-the-world* isn't a URL that's going to feature in the final design of our application. But we're committed to changing

one thing at a time. While our application only supports one list, this is the only URL that makes sense. We're still moving forwards, in that we'll have a different URL for our list and our home page, which is a step along the way to a more REST-ful design. Later, when we have multiple lists, it will be easy to change.

Another way of thinking about it is as a problem-solving technique: our new URL design is currently not implemented, so it works for 0 items. Ultimately, we want to solve for *n* items, but solving for 1 item is a good step along the way.

Running the unit tests gives us an expected fail:

```
$ python3 manage.py test lists
[...]
AssertionError: '/' != '/lists/the-only-list-in-the-world/'
```

We can go adjust our `home_page` view in *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})
```

Of course, that will now totally break the functional tests, because there is no such URL on our site yet. Sure enough, if you run them, you'll find they fail just after trying to submit the first item, saying that they can't find the list table; it's because URL */the-only-list-in-the-world/* doesn't exist yet!

```
    self.check_for_row_in_list_table('1: Buy peacock feathers')
[...]
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"id","selector":"id_list_table"}' ; Stacktrace:
```

So, let's build a special URL for our one and only list.

Testing Views, Templates, and URLs Together with the Django Test Client

In previous chapters we've used unit tests that check the URL resolution explicitly, that test view functions by actually calling them, and that check that views render templates correctly too. Django actually provides us with a little tool that can do all three at once, which we'll use now.

I wanted to show you how to "roll your own" first, partially because it's a better introduction to how Django works, but also because those techniques are portable—you may not always use Django, but you'll almost always have view functions, templates, and URL mappings, and now you know how to test them.

A New Test Class

So let's use the Django test client. Open up *lists/tests.py*, and add a new test class called `ListViewTest`. Then copy the method called `test_home_page_displays_all_list_items` across from `HomePageTest` into our new class, rename it, and adapt it slightly:

lists/tests.py (ch06l009).

```
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/') #❶

        self.assertContains(response, 'itemey 1') #❷
        self.assertContains(response, 'itemey 2') #❸
```

❶ Instead of calling the view function directly, we use the Django test client, which is an attribute of the Django `TestCase` called `self.client`. We tell it to `.get` the URL we're testing—it's actually a very similar API to the one that Selenium uses.

❷ ❸ Instead of using the slightly annoying `assertIn/response.content.decode()` dance, Django provides the `assertContains` method which knows how to deal with responses and the bytes of their content.

Some people really don't like the Django test client. They say it provides too much magic, and involves too much of the stack to be used in a real "unit" test—you end up writing what are more properly called integrated tests. They also complain that it is relatively slow (and relatively is measured in milliseconds). We'll explore this argument further in a later chapter. For now we'll use it because it's extremely convenient!

Let's try running the test now:

```
AssertionError: 404 != 200 : Couldn't retrieve content: Response code was 404
```

A New URL

Our singleton list URL doesn't exist yet. We fix that in *superlists/urls.py*.

Watch out for trailing slashes in URLs, both here in the tests and in *urls.py*—They're a common source of bugs.

superlists/urls.py.

```
urlpatterns = patterns('',
    url(r'^$', 'lists.views.home_page', name='home'),
    url(r'^lists/the-only-list-in-the-world/$', 'lists.views.view_list',
        name='view_list'
    ),
    # url(r'^admin/', include(admin.site.urls)),
)
```

Running the tests again, we get:

```
AttributeError: 'module' object has no attribute 'view_list'
[...]
django.core.exceptions.ViewDoesNotExist: Could not import
lists.views.view_list. View does not exist in module lists.views.
```

A New View Function

Nicely self-explanatory. Let's create a dummy view function in *lists/views.py*:

lists/views.py.

```
def view_list(request):
    pass
```

Now we get:

```
ValueError: The view lists.views.view_list didn't return an HttpResponse
object. It returned None instead.
```

Let's copy the two last lines from the `home_page` view and see if they'll do the trick:

lists/views.py.

```
def view_list(request):
    items = Item.objects.all()
    return render(request, 'home.html', {'items': items})
```

Rerun the tests and they should pass:

```
Ran 8 tests in 0.016s
OK
```

And the FTs should get a little further on:

```
AssertionError: '2: Use peacock feathers to make a fly' not found in ['1: Buy
peacock feathers']
```

Green? Refactor

Time for a little tidying up.

In the Red/Green/Refactor dance, we've arrived at green, so we should see what needs a refactor. We now have two views, one for the home page, and one for an individual list. Both are currently using the same template, and passing it all the list items currently in the database. If we look through our unit test methods, we can see some stuff we probably want to change:

```
$ grep -E "class|def" lists/tests.py
class HomePageTest(TestCase):
    def test_root_url_resolves_to_home_page_view(self):
    def test_home_page_returns_correct_html(self):
    def test_home_page_displays_all_list_items(self):
    def test_home_page_can_save_a_POST_request(self):
    def test_home_page_redirects_after_POST(self):
    def test_home_page_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):
```

We can definitely delete the `test_home_page_displays_all_list_items` method, it's no longer needed. If you run **`manage.py test lists`** now, it should say it ran 7 tests instead of 8:

```
Ran 7 tests in 0.016s
OK
```

Next, we don't actually need the home page to display all list items any more; it should just show a single input box inviting you to start a new list.

A Separate Template for Viewing Lists

Since the home page and the list view are now quite distinct pages, they should be using different HTML templates; *home.html* can have the single input box, whereas a new template, *list.html*, can take care of showing the table of existing items.

Let's add a new test test to check that it's using a different template:

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` is one of the more useful functions that the Django test client gives us. Let's see what it says:

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

Great! Let's change the view:

lists/views.py.

```
def view_list(request):
    items = Item.objects.all()
    return render(request, 'list.html', {'items': items})
```

But, obviously, that template doesn't exist yet. If we run the unit tests, we get:

```
django.template.base.TemplateDoesNotExist: list.html
```

Let's create a new file at *lists/templates/list.html*:

```
$ touch lists/templates/list.html
```

A blank template, which gives us this error—good to know the tests are there to make sure we fill it in:

```
AssertionError: False is not true : Couldn't find 'itemey 1' in response
```

The template for an individual list will reuse quite a lot of the stuff we currently have in *home.html*, so we can start by just copying that:

```
$ cp lists/templates/home.html lists/templates/list.html
```

That gets the tests back to passing (green). Now let's do a little more tidying up (refactoring). We said the home page doesn't need to list items, it only needs the new list input field, so we can remove some lines from *lists/templates/home.html*, and maybe slightly tweak the `h1` to say "Start a new To-Do list":

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>
```

We rerun the unit tests to check that hasn't broken anything … good…

There's actually no need to pass all the items to the *home.html* template in our `home_page` view, so we can simplify that:

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')
```

Rerun the unit tests; they still pass. Let's run the functional tests:

```
AssertionError: '2: Use peacock feathers to make a fly' not found in ['1: Buy
peacock feathers']
```

We're still failing to input the second item. What's going on here? Well, the problem is that our new item forms are both missing an `action=` attribute, which means that, by default, they submit to the same URL they were rendered from. That works for the home page, because it's the only one that knows how to deal with POST requests currently, but it won't work for our `view_list` function, which is just ignoring the POST.

We can fix that in *lists/templates/list.html*:

lists/templates/list.html (ch06l019).

```
        <form method="POST" action="/">
```

And try running the FT again:

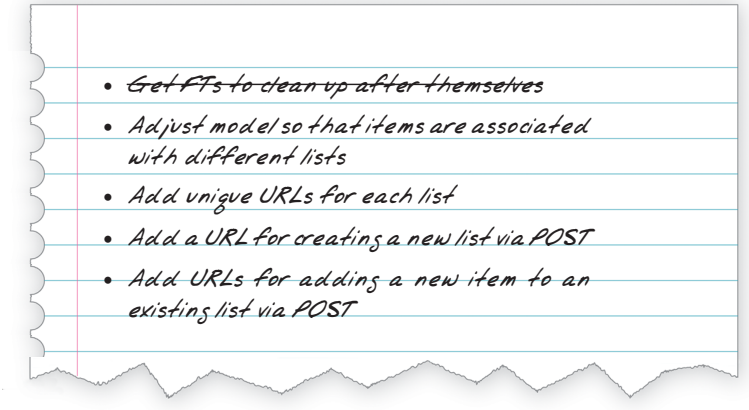
```
        self.assertNotEqual(francis_list_url, edith_list_url)
    AssertionError: 'http://localhost:8081/lists/the-only-list-in-the-world/' ==
    'http://localhost:8081/lists/the-only-list-in-the-world/'
```

Hooray! We're back to where we were earlier, which means our refactoring is complete —we now have a unique URL for our one list. It may feel like we haven't made much headway since, functionally, the site still behaves almost exactly like it did when we started the chapter, but this really is progress. We've started on the road to our new design, and we've implemented a number of stepping stones *without making anything worse than it was before*. Let's commit our progress so far:

```
$ git status # should show 4 changed files and 1 new file, list.html
$ git add lists/templates/list.html
$ git diff # should show we've simplified home.html,
           # moved one test to a new class in lists/tests.py added a new view
           # in views.py, and simplified home_page and made one addition to
           # urls.py
$ git commit -a # add a message summarising the above, maybe something like
                # "new URL, view and template to display lists"
```

Another URL and View for Adding List Items

Where are we with our own to-do list?



We've *sort of* made progress on the third item, even if there's still only one list in the world. Item 2 is a bit scary. Can we do something about items 4 or 5?

Let's have a new URL for adding new list items. If nothing else, it'll simplify the home page view.

A Test Class for New List Creation

Open up *lists/tests.py*, and *move* the `test_home_page_can_save_a_POST_request` and `test_home_page_redirects_after_POST` methods into a new class, then change their names:

lists/tests.py (ch06l021-1).

```
class NewListTest(TestCase):

    def test_saving_a_POST_request(self):
        request = HttpRequest()
        request.method = 'POST'
        [...]

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

Now let's use the Django test client:

lists/tests.py (ch06l021-2).

```
class NewListTest(TestCase):

    def test_saving_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/')
```

This is another place to pay attention to trailing slashes, incidentally. It's `/new`, with no trailing slash. The convention I'm using is that URLs without a trailing slash are "action" URLs which modify the database.

Try running that:

```
    self.assertEqual(Item.objects.count(), 1)
AssertionError: 0 != 1
[...]
    self.assertEqual(response.status_code, 302)
AssertionError: 404 != 302
```

The first failure tells us we're not saving a new item to the database, and the second says that, instead of returning a 302 redirect, our view is returning a 404. That's because we haven't built a URL for */lists/new*, so the `client.post` is just getting a 404 response.

Do you remember how we split this out into two tests in the last chapter? If we only had one test that checked both the saving and the redirect, it would have failed on the `0 != 1` failure, which would have been much harder to debug. Ask me how I know this.

A URL and View for New List Creation

Let's build our new URL now:

superlists/urls.py.

```
urlpatterns = patterns('',
    url(r'^$', 'lists.views.home_page', name='home'),
    url(r'^lists/the-only-list-in-the-world/$', 'lists.views.view_list',
        name='view_list'
    ),
    url(r'^lists/new$', 'lists.views.new_list', name='new_list'),
    # url(r'^admin/', include(admin.site.urls)),
)
```

Next we get a `ViewDoesNotExist`, so let's fix that, in *lists/views.py*:

lists/views.py.

```
def new_list(request):
    pass
```

Then we get "The view lists.views.new_list didn't return an HttpResponse object". (This is getting rather familiar!) We could return a raw `HttpResponse`, but since we know we'll need a redirect, let's borrow a line from `home_page`:

lists/views.py.

```
def new_list(request):
    return redirect('/lists/the-only-list-in-the-world/')
```

That gives:

```
    self.assertEqual(Item.objects.count(), 1)
AssertionError: 0 != 1
[...]
AssertionError: 'http://testserver/lists/the-only-list-in-the-world/' !=
'/lists/the-only-list-in-the-world/'
```

Let's start with the first failure, because it's reasonably straightforward. We borrow another line from `home_page`:

lists/views.py.

```
def new_list(request):
    Item.objects.create(text=request.POST['item_text'])
    return redirect('/lists/the-only-list-in-the-world/')
```

And that takes us down to just the second, unexpected failure:

```
    self.assertEqual(response['location'],
'/lists/the-only-list-in-the-world/')
AssertionError: 'http://testserver/lists/the-only-list-in-the-world/' !=
'/lists/the-only-list-in-the-world/'
```

It's happening because the Django test client behaves slightly differently to our pure view function; it's using the full Django stack which adds the domain to our relative URL. Let's use another of Django's test helper functions, instead of our two-step check for the redirect:

lists/tests.py.

```
    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/')
```

That now passes:

```
Ran 8 tests in 0.030s

OK
```

Removing Now-Redundant Code and Tests

We're looking good. Since our new views are now doing most of the work that `home_page` used to do, we should be able to massively simplify it. Can we remove the whole `if request.method == 'POST'` section, for example?

lists/views.py.

```
def home_page(request):
    return render(request, 'home.html')
```

Yep!

```
OK
```

And while we're at it, we can remove the now-redundant `test_home_page_only_saves_items_when_necessary` test too!

Doesn't that feel good? The view functions are looking much simpler. We rerun the tests to make sure…

```
Ran 7 tests in 0.016s
OK
```

Pointing Our Forms at the New URL

Finally, let's wire up our two forms to use this new URL. In *both home.html* and *lists.html*:

lists/templates/home.html, lists/templates/list.html.

```
<form method="POST" action="/lists/new">
```

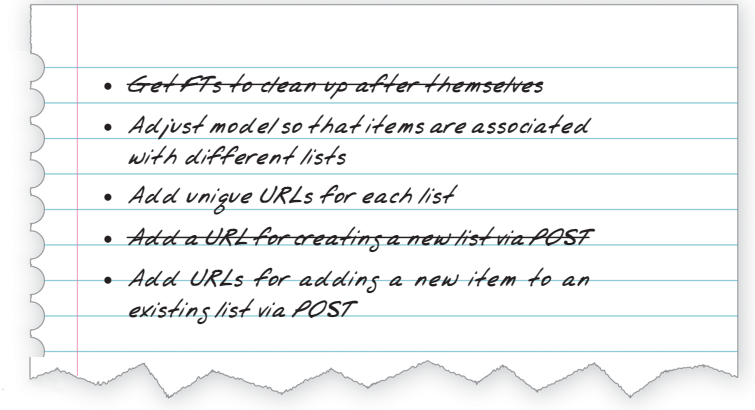
And we rerun our FTs to make sure everything still works, or works at least as well as it did earlier…

```
AssertionError: 'http://localhost:8081/lists/the-only-list-in-the-world/' ==
'http://localhost:8081/lists/the-only-list-in-the-world/'
```

Yup, we get to the same point we did before. That's a nicely self-contained commit, in that we've made a bunch of changes to our URLs, our *views.py* is looking much neater and tidier, and we're sure the application is still working as well as it did before. We're getting good at this refactoring malarkey!

```
$ git status # 5 changed files
$ git diff # URLs for forms x2, moved code in views + tests, new URL
$ git commit -a
```

And we can cross out an item on the to-do list:



Adjusting Our Models

Enough housekeeping with our URLs. It's time to bite the bullet and change our models. Let's adjust the model unit test. Just for a change, I'll present the changes in the form of a diff:

lists/tests.py.

```
@@ -3,7 +3,7 @@ from django.http import HttpRequest
 from django.template.loader import render_to_string
 from django.test import TestCase

-from lists.models import Item
+from lists.models import Item, List
 from lists.views import home_page

 class HomePageTest(TestCase):
@@ -60,22 +60,32 @@ 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_)
```

We create a new `List` object, and then we assign each item to it by assigning it as its `.list` property. We check the list is properly saved, and we check that the two items have also saved their relationship to the list. You'll also notice that we can compare list objects with each other directly (`saved_list` and `list`)—behind the scenes, these will compare themselves by checking their primary key (the `.id` attribute) is the same.

I'm using the variable name `list_` to avoid "shadowing" the Python built-in `list` function. It's ugly, but all the other options I tried were equally ugly or worse (`my_list`, `the_list`, `list1`, `listey`…).

Time for another unit-test/code cycle.

For the first couple of iterations, rather than explicitly showing you what code to enter in between every test run, I'm only going to show you the expected error messages from running the tests. I'll let you figure out what each minimal code change should be on your own:

Your first error should be:

```
ImportError: cannot import name 'List'
```

Fix that, then you should see:

```
AttributeError: 'List' object has no attribute 'save'
```

Next you should see:

```
django.db.utils.OperationalError: no such table: lists_list
```

So we run a `makemigrations`:

```
$ python3 manage.py makemigrations
Migrations for 'lists':
  0003_list.py:
    - Create model List
```

And then you should see:

```
    self.assertEqual(first_saved_item.list, list_)
AttributeError: 'Item' object has no attribute 'list'
```

A Foreign Key Relationship

How do we give our `Item` a list attribute? Let's just try naively making it like the `text` attribute:

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='')
```

As usual, the tests tell us we need a migration:

```
$ python3 manage.py test lists
[...]
django.db.utils.OperationalError: table lists_item has no column named list

$ python3 manage.py makemigrations
Migrations for 'lists':
  0004_item_list.py:
    - Add field list to item
```

Let's see what that gives us:

```
AssertionError: 'List object' != <List: List object>
```

We're not quite there. Look closely at each side of the `!=`. Django has only saved the string representation of the `List` object. To save the relationship to the object itself, we tell Django about the relationship between the two classes using a `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)
```

That'll need a migration too. Since the last one was a red herring, let's delete it and replace it with a new one:

```
$ rm lists/migrations/0004_item_list.py
$ python3 manage.py makemigrations
Migrations for 'lists':
```

```
0004_item_list.py:
  - Add field list to item
```

Deleting migrations is dangerous. If you delete a migration that's already been applied to a database somewhere, Django will be confused about what state it's in, and how to apply future migrations. You should only do it when you're sure the migration hasn't been used. A good rule of thumb is that you should never delete a migration that's been committed to your VCS.

Adjusting the Rest of the World to Our New Models

Back in our tests, now what happens?

```
$ python3 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_saving_a_POST_request (lists.tests.NewListTest)
django.db.utils.IntegrityError: NOT NULL constraint failed: lists_item.list_id

Ran 7 tests in 0.021s

FAILED (errors=3)
```

Oh dear!

There is some good news. Although it's hard to see, our model tests are passing. But three of our view tests are failing nastily.

The reason is because of the new relationship we've introduced between Items and Lists, which requires each list to have a parent item, which our old tests weren't prepared for.

Still, this is exactly why we have tests. Let's get them working again. The easiest is the `ListViewTest`; we just create a parent list for our two test items:

lists/tests.py (ch06l031).

```
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_)
```

That gets us down to two failing tests, both on tests that try to POST to our `new_list` view. Decoding the tracebacks using our usual technique, working back from error, to line of test code, to the line of our own code that caused the failure, we identify:

```
File "/workspace/superlists/lists/views.py", line 14, in new_list
Item.objects.create(text=request.POST['item_text'])
```

It's when we try and create an item without a parent list. So we make a similar change in the 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/')
```

And that gets our tests passing again:

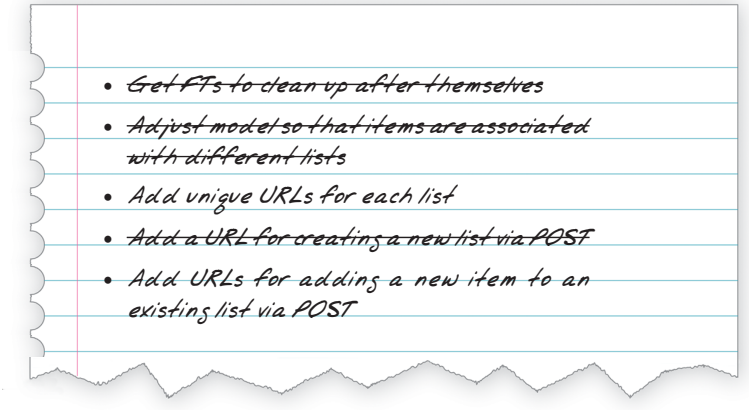
```
OK
```

Are you cringing internally at this point? *Arg! This feels so wrong, we create a new list for every single new item submission, and we're still just displaying all items as if they belong to the same list!* I know, I feel the same. The step-by-step approach, in which you go from working code to working code, is counterintuitive. I always feel like just diving in and trying to fix everything all in one go, instead of going from one weird half-finished state to another. But remember the Testing Goat! When you're up a mountain, you want to think very carefully about where you put each foot, and take one step at a time, checking at each stage that the place you've put it hasn't caused you to fall off a cliff.

So just to reassure ourselves that things have worked, we rerun the FT. Sure enough, it gets all the way through to where we were before. We haven't broken anything, and we've made a change to the database. That's something to be pleased with! Let's commit:

```
$ git status # 3 changed files, plus 2 migrations
$ git add lists
$ git diff --staged
$ git commit
```

And we can cross out another item on the to-do list:



Each List Should Have Its Own URL

What shall we use as the unique identifier for our lists? Probably the simplest thing, for now, is just to use the auto-generated `id` field from the database. Let's change `ListViewTest` so that the two tests point at new URLs.

We'll also change the old `test_displays_all_items` test and call it `test_displays_only_items_for_that_list` instead, and make it check that only the items for a specific list are displayed:

lists/tests.py (ch06l033-1).

```
class ListViewTest(TestCase):

    def test_uses_list_template(self):
        list_ = List.objects.create()
        response = self.client.get('/lists/%d/' % (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('/lists/%d/' % (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')
```

If you're not familiar with Python string substitutions, or the `printf` function from C, maybe that `%d` is a little confusing? *Dive Into Python* has a good overview, if you want to go look them up quickly. We'll see an alternative string substitution syntax later in the book too.

Running the unit tests gives an expected 404, and another related error:

```
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
```

Capturing Parameters from URLs

It's time to learn how we can pass parameters from URLs to views:

superlists/urls.py.

```
urlpatterns = patterns('',
    url(r'^$', 'lists.views.home_page', name='home'),
    url(r'^lists/(.+)/$', 'lists.views.view_list', name='view_list'),
    url(r'^lists/new$', 'lists.views.new_list', name='new_list'),
    # url(r'^admin/', include(admin.site.urls)),
)
```

We adjust the regular expression for our URL to include a *capture group*, `(.+)`, which will match any characters, up to the following `/`. The captured text will get passed to the view as an argument.

In other words, if we go to the URL */lists/1/*, `view_list` will get a second argument after the normal `request` argument, namely the string `"1"`. If we go to */lists/foo/*, we get `view_list(request, "foo")`.

But our view doesn't expect an argument yet! Sure enough, this causes problems:

```
ERROR: test_displays_only_items_for_that_list (lists.tests.ListViewTest)
ERROR: test_uses_list_template (lists.tests.ListViewTest)
ERROR: test_redirects_after_POST (lists.tests.NewListTest)
[...]
TypeError: view_list() takes 1 positional argument but 2 were given
```

We can fix that easily with a dummy parameter in *views.py*:

lists/views.py.

```
def view_list(request, list_id):
    [...]
```

Now we're down to our expected failure:

```
FAIL: test_displays_only_items_for_that_list (lists.tests.ListViewTest)
AssertionError: 1 != 0 : Response should not contain 'other list item 1'
```

Let's make our view discriminate over which items it sends to the 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})
```

Adjusting new_list to the New World

Now we get errors in another test:

```
ERROR: test_redirects_after_POST (lists.tests.NewListTest)
ValueError: invalid literal for int() with base 10:
'the-only-list-in-the-world'
```

Let's take a look at this test then, since it's whining:

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/')
```

It looks like it hasn't been adjusted to the new world of Lists and Items. The test should be saying that this view redirects to the URL of the new list it just created:

lists/tests.py (ch06l036-1).

```
    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, '/lists/%d/' % (new_list.id,))
```

That still gives us the *invalid literal* error. We take a look at the view itself, and change it so it redirects to a valid place:

lists/views.py (ch06l036-2).

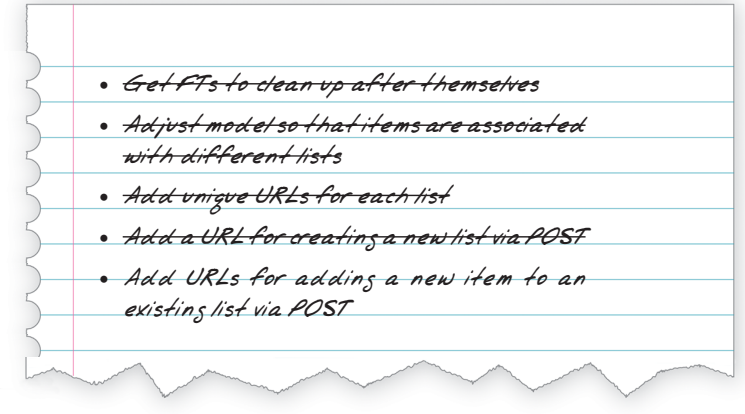
```
def new_list(request):
    list_ = List.objects.create()
    Item.objects.create(text=request.POST['item_text'], list=list_)
    return redirect('/lists/%d/' % (list_.id,))
```

That gets us back to passing unit tests. What about the functional tests? We must be almost there?

```
AssertionError: '2: Use peacock feathers to make a fly' not found in ['1: Use
peacock feathers to make a fly']
```

The functional tests have warned us of a regression in our application: because we're now creating a new list for every single POST submission, we have broken the ability to add multiple items to a list. This is exactly what we have functional tests for!

And it correlates nicely with the last item on our to-do list:



One More View to Handle Adding Items to an Existing List

We need a URL and view to handle adding a new item to an existing list (*/lists/<list_id>/add_item*). We're getting pretty good at these now, so let's knock one together quickly:

lists/tests.py.

```
class NewItemTest(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(
            '/lists/%d/add_item' % (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_redirects_to_list_view(self):
        other_list = List.objects.create()
        correct_list = List.objects.create()
```

```
        response = self.client.post(
            '/lists/%d/add_item' % (correct_list.id,),
            data={'item_text': 'A new item for an existing list'}
        )

        self.assertRedirects(response, '/lists/%d/' % (correct_list.id,))
```

We get:

```
AssertionError: 0 != 1
[...]
AssertionError: 301 != 302 : Response didn't redirect as expected: Response
code was 301 (expected 302)
```

Beware of Greedy Regular Expressions!

That's a little strange. We haven't actually specified a URL for */lists/1/add_item* yet, so our expected failure is `404 != 302`. Why are we getting a 301?

This was a bit of a puzzler, but it's because we've used a very "greedy" regular expression in our URL:

```
    url(r'^lists/(.+)/$', 'lists.views.view_list', name='view_list'),
```

Django has some built-in code to issue a permanent redirect (301) whenever someone asks for a URL which is *almost* right, except for a missing slash. In this case, */lists/1/add_item/* would be a match for `lists/(.+)/`, with the `(.+)` capturing `1/add_item`. So Django "helpfully" guesses that we actually wanted the URL with a trailing slash.

We can fix that by making our URL pattern explicitly capture only numerical digits, by using the regular expression `\d`:

superlists/urls.py.

```
    url(r'^lists/(\d+)/$', 'lists.views.view_list', name='view_list'),
```

That gives:

```
AssertionError: 0 != 1
[...]
AssertionError: 404 != 302 : Response didn't redirect as expected: Response
code was 404 (expected 302)
```

The Last New URL

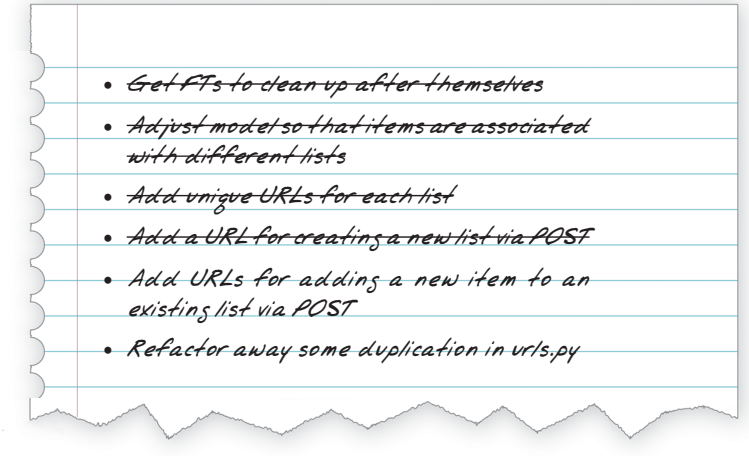
Now we've got our expected 404, let's add a new URL for adding new items to existing lists:

superlists/urls.py.

```
urlpatterns = patterns('',
    url(r'^$', 'lists.views.home_page', name='home'),
    url(r'^lists/(\d+)/$', 'lists.views.view_list', name='view_list'),
    url(r'^lists/(\d+)/add_item$', 'lists.views.add_item', name='add_item'),
    url(r'^lists/new$', 'lists.views.new_list', name='new_list'),
```

```
    # url(r'^admin/', include(admin.site.urls)),
)
```

Three very similar-looking URLs there. Let's make a note on our to-do list; they look like good candidates for a refactoring.



Back to the tests, we now get:

```
django.core.exceptions.ViewDoesNotExist: Could not import lists.views.add_item.
View does not exist in module lists.views.
```

The Last New View

Let's try:

lists/views.py

```
def add_item(request):
    pass
```

Aha:

```
TypeError: add_item() takes 1 positional argument but 2 were given
```

lists/views.py

```
def add_item(request, list_id):
    pass
```

And then:

```
ValueError: The view lists.views.add_item didn't return an HttpResponse object.
It returned None instead.
```

We can copy the `redirect` from `new_list` and the `List.objects.get` from `view_list`:

lists/views.py.

```
def add_item(request, list_id):
    list_ = List.objects.get(id=list_id)
    return redirect('/lists/%d/' % (list_.id,))
```

That takes us to:

```
    self.assertEqual(Item.objects.count(), 1)
AssertionError: 0 != 1
```

Finally we make it save our new list item:

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('/lists/%d/' % (list_.id,))
```

And we're back to passing tests.

```
Ran 9 tests in 0.050s

OK
```

But How to Use That URL in the Form?

Now we just need to use this URL in our *list.html* template. Open it up and adjust the form tag…

lists/templates/list.html.

```
<form method="POST" action="but what should we put here?">
```

... oh. To get the URL for adding to the current list, the template needs to know what list it's rendering, as well as what the items are. We want to be able to do something like this:

lists/templates/list.html.

```
<form method="POST" action="/lists/{{ list.id }}/add_item">
```

For that to work, the view will have to pass the list to the template. Let's create a new unit test in `ListViewTest`:

lists/tests.py (ch06l041).

```
def test_passes_correct_list_to_template(self):
    other_list = List.objects.create()
    correct_list = List.objects.create()
    response = self.client.get('/lists/%d/' % (correct_list.id,))
    self.assertEqual(response.context['list'], correct_list)
```

`response.context` represents the context we're going to pass into the render function —the Django test client puts it on the `response` object for us, to help with testing. That gives us:

```
KeyError: 'list'
```

because we're not passing `list` into the template. It actually gives us an opportunity to simplify a little:

lists/views.py.

```
def view_list(request, list_id):
    list_ = List.objects.get(id=list_id)
    return render(request, 'list.html', {'list': list_})
```

That, of course, will break because the template is expecting `items`:

```
AssertionError: False is not true : Couldn't find 'itemey 1' in response
```

But we can fix it in *list.html*, as well as adjusting the form's POST action:

lists/templates/list.html (ch06l043).

```
    <form method="POST" action="/lists/{{ list.id }}/add_item">

    [...]

        {% for item in list.item_set.all %}
            <tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>
        {% endfor %}
```

`.item_set` is called a "reverse lookup"—it's one of Django's incredibly useful bits of ORM that lets you look up an object's related items from a different table…

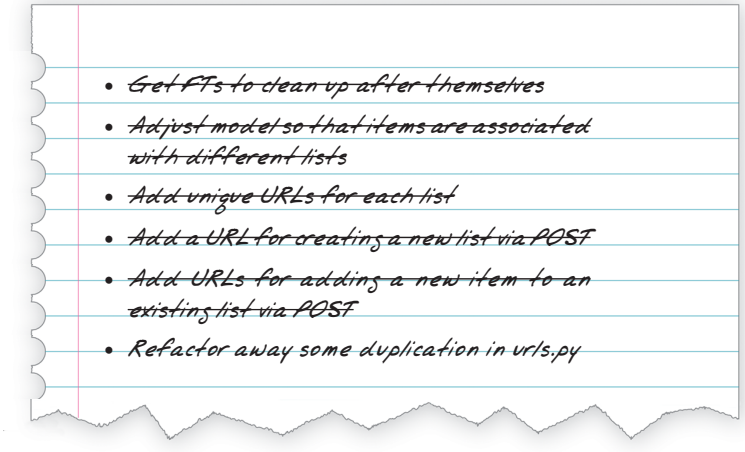
So that gets the unit tests to pass:

```
Ran 10 tests in 0.060s

OK
```

How about the FT?

```
$ python3 manage.py test functional_tests
Creating test database for alias 'default'...
.
 ----------------------------------------------------------------------
Ran 1 test in 5.824s

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

Yes! And a quick check on our to-do list:



Irritatingly, the Testing Goat is a stickler for tying up loose ends too, so we've got to do this one final thing.

Before we start, we'll do a commit—always make sure you've got a commit of a working state before embarking on a refactor:

```
$ git diff
$ git commit -am "new URL + view for adding to existing lists. FT passes :-)"
```

A Final Refactor Using URL includes

superlists/urls.py is really meant for URLs that apply to your entire site. For URLs that only apply to the `lists` app, Django encourages us to use a separate *lists/urls.py*, to make the app more self-contained. The simplest way to make one is to use a copy of the existing *urls.py*:

```
$ cp superlists/urls.py lists/
```

Then we replace three lines in *superlists/urls.py* with an `include`. Notice that `include` can take a part of a URL regex as a prefix, which will be applied to all the included URLs (this is the bit where we reduce duplication, as well as giving our code a better structure):

superlists/urls.py.

```
urlpatterns = patterns('',
    url(r'^$', 'lists.views.home_page', name='home'),
    url(r'^lists/', include('lists.urls')),
    # url(r'^admin/', include(admin.site.urls)),
)
```

And *lists/urls.py* we can trim down to only include the latter part of our three URLs, and none of the other stuff from the parent *urls.py*:

lists/urls.py (ch06l045).

```
from django.conf.urls import patterns, url

urlpatterns = patterns('',
    url(r'^(\d+)/$', 'lists.views.view_list', name='view_list'),
    url(r'^(\d+)/add_item$', 'lists.views.add_item', name='add_item'),
    url(r'^new$', 'lists.views.new_list', name='new_list'),
)
```

Rerun the unit tests to check everything worked. When I did it, I couldn't quite believe I did it correctly on the first go. It always pays to be skeptical of your own abilities, so I deliberately changed one of the URLs slightly, just to check if it broke a test. It did. We're covered.

Feel free to try it yourself! Remember to change it back, check the tests all pass again, and then commit:

```
$ git status
$ git add lists/urls.py
$ git add superlists/urls.py
$ git diff --staged
$ git commit
```

Phew. A marathon chapter. But we covered a number of important topics, starting with test isolation, and then some thinking about design. We covered some rules of thumb like "YAGNI" and "three strikes then refactor". But, most importantly, we saw how to adapt an existing site step by step, going from working state to working state, in order to iterate towards a new design.

I'd say we're pretty close to being able to ship this site, as the very first beta of the superlists website that's going to take over the world. Maybe it needs a little prettification first … let's look at what we need to do to deploy it in the next couple of chapters.

Useful TDD Concepts and Rules Of Thumb

Test Isolation and Global State
: Different tests shouldn't affect one another. This means we need to reset any permanent state at the end of each test. Django's test runner helps us do this by creating a test database, which it wipes clean in between each test. (See also Chapter 19.)

Working State to Working State (aka The Testing Goat vs. Refactoring Cat)
: Our natural urge is often to dive in and fix everything at once … but if we're not careful, we'll end up like Refactoring Cat, in a situation with loads of changes to our code and nothing working. The Testing Goat encourages us to take one step at a time, and go from working state to working state.

YAGNI
: You ain't gonna need it! Avoid the temptation to write code that you think *might* be useful, just because it suggests itself at the time. Chances are, you won't use it, or you won't have anticipated your future requirements correctly. See Chapter 18 for one methodology that helps us avoid this trap.

PART II
Web Development Sine Qua Nons

Real developers ship.

— Jeff Atwood

If this were just a guide to TDD in a normal programming field, we might be able to congratulate ourselves about now. After all, we've got some solid basics of TDD and Django under our belts; we've got all we need to start building a website.

But, real developers ship, and in order to ship, we're going to have to tackle some of the trickier but unavoidable aspects of web development: static files, form data validation, the dreaded JavaScript, but most hairy of all, deployment to a production server.

At every stage, TDD can help us to get these things right too.

In this section, I'm still trying to keep the learning curve relatively soft, but we will meet several major new concepts and technologies. I'll only be able to dip lightly into each one—I hope to demonstrate enough of each to get you started when you get to your own project, but you will also need to do your own reading around when you start to apply these topics in "real life".

For example, if you weren't familiar with Django before starting on the book, you may find that taking a little time to run through the official Django tutorial at this point would complement what you've learned so far nicely, and will leave you more confident with the Django stuff over the next few chapters, so you can focus on the core concepts.

Oh, but there's lots of fun stuff coming up! Just you wait!

CHAPTER 7

Prettification: Layout and Styling, and What to Test About It

We're starting to think about releasing the first version of our site, but we're a bit embarrassed by how ugly it looks at the moment. In this chapter, we'll cover some of the basics of styling, including integrating an HTML/CSS framework called Bootstrap. We'll learn how static files work in Django, and what we need to do about testing them.

What to Functionally Test About Layout and Style

Our site is undeniably a bit unattractive at the moment (Figure 7-1).

If you spin up your dev server with `manage.py runserver`, you may run into a database error "table lists_item has no column named list_id". You need to update your local database to reflect the changes we made in *models.py*. Use `manage.py migrate`.

We can't be adding to Python's reputation for being ugly, so let's do a tiny bit of polishing. Here's a few things we might want:

- A nice large input field for adding new and existing lists
- A large, attention-grabbing, centered box to put it in

How do we apply TDD to these things? Most people will tell you you shouldn't test aesthetics, and they're right. It's a bit like testing a constant, in that tests usually wouldn't add any value.

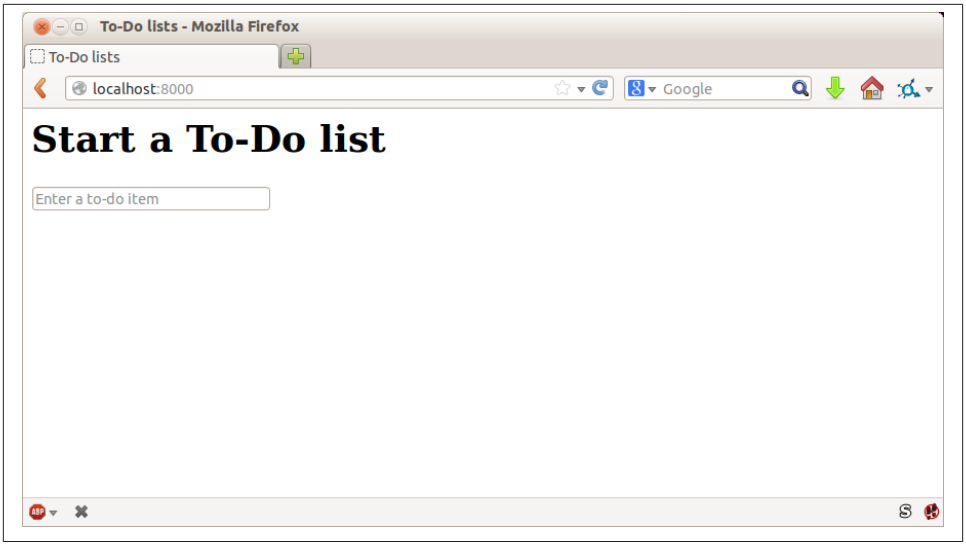


Figure 7-1. Our homepage, looking a little ugly…

But we can test the implementation of our aesthetics—just enough to reassure ourselves that things are working. For example, we're going to use Cascading Style Sheets (CSS) for our styling, and they are loaded as static files. Static files can be a bit tricky to configure (especially, as we'll see later, when you move off your own PC and onto a hosting site), so we'll want some kind of simple "smoke test" that the CSS has loaded. We don't have to test fonts and colours and every single pixel, but we can do a quick check that the main input box is aligned the way we want it on each page, and that will give us confidence that the rest of the styling for that page is probably loaded too.

We start with a new test method inside our functional test:

functional_tests/tests.py (ch07l001).

```
class NewVisitorTest(LiveServerTestCase):
    [...]


    def test_layout_and_styling(self):
        # Edith goes to the home page
        self.browser.get(self.live_server_url)
        self.browser.set_window_size(1024, 768)

        # She notices the input box is nicely centered
        inputbox = self.browser.find_element_by_id('id_new_item')
        self.assertAlmostEqual(
            inputbox.location['x'] + inputbox.size['width'] / 2,
            512,
            delta=5
        )
```

A few new things here. We start by setting the window size to a fixed size. We then find the input element, look at its size and location, and do a little maths to check whether it seems to be positioned in the middle of the page. `assertAlmostEqual` helps us to deal with rounding errors by letting us specify that we want our arithmetic to work to within plus or minus five pixels.

If we run the functional tests, we get:

```
$ python3 manage.py test functional_tests
Creating test database for alias 'default'...
.F
======================================================================
FAIL: test_layout_and_styling (functional_tests.tests.NewVisitorTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/workspace/superlists/functional_tests/tests.py", line 104, in
test_layout_and_styling
    delta=5
AssertionError: 110.0 != 512 within 5 delta

 ----------------------------------------------------------------------
Ran 2 tests in 9.188s

FAILED (failures=1)
Destroying test database for alias 'default'...
```

That's the expected failure. Still, this kind of FT is easy to get wrong, so let's use a quick-and-dirty "cheat" solution, to check that the FT also passes when the input box is centered. We'll delete this code again almost as soon as we've used it to check the FT:

lists/templates/home.html (ch07l002).

```
<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>
```

That passes, which means the FT works. Let's extend it to make sure that the input box is also center-aligned on the page for a new list:

functional_tests/tests.py (ch07l003).

```
        # She starts a new list and sees the input is nicely
        # centered there too
        inputbox.send_keys('testing\n')
        inputbox = self.browser.find_element_by_id('id_new_item')
        self.assertAlmostEqual(
            inputbox.location['x'] + inputbox.size['width'] / 2,
            512,
            delta=5
        )
```

That gives us another test failure:

```
  File "/workspace/superlists/functional_tests/tests.py", line 114, in
test_layout_and_styling
    delta=5
AssertionError: 110.0 != 512 within 5 delta
```

Let's commit just the FT:

```
$ git add functional_tests/tests.py
$ git commit -m "first steps of FT for layout + styling"
```

Now it feels like we're justified in finding a "proper" solution to our need for some better styling for our site. We can back out our hacky `<p style="text-align: center">`:

```
$ git reset --hard
```

`git reset --hard` is the "take off and nuke the site from orbit" Git command, so be careful with it—it blows away all your uncommitted changes. Unlike almost everything else you can do with Git, there's no way of going back after this one.

Prettification: Using a CSS Framework

Design is hard, and doubly so now that we have to deal with mobile, tablets, and so forth. That's why many programmers, particularly lazy ones like me, are turning to CSS frameworks to solve some of those problems for them. There are lots of frameworks out there, but one of the earliest and most popular is Twitter's Bootstrap. Let's use that.

You can find bootstrap at *http://getbootstrap.com/*.

We'll download it and put it in a new folder called *static* inside the `lists` app:[1]

```
$ wget -O bootstrap.zip https://github.com/twbs/bootstrap/releases/download/\
v3.1.0/bootstrap-3.1.0-dist.zip
$ unzip bootstrap.zip
$ mkdir lists/static
$ mv dist lists/static/bootstrap
$ rm bootstrap.zip
```

Bootstrap comes with a plain, uncustomised installation in the *dist* folder. We're going to use that for now, but you should really never do this for a real site—vanilla Bootstrap is instantly recognisable, and a big signal to anyone in the know that you couldn't be bothered to style your site. Learn how to use LESS and change the font, if nothing else! There is info in Bootstrap's docs, or there's a good guide here.

Our *lists* folder will end up looking like this:

1. On Windows, you may not have `wget` and `unzip`, but I'm sure you can figure out how to download Bootstrap, unzip it, and put the contents of the *dist* folder into the *lists/static/bootstrap* folder.

```
$ 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
│       │   ├── glyphicons-halflings-regular.eot
│       │   ├── glyphicons-halflings-regular.svg
│       │   ├── glyphicons-halflings-regular.ttf
│       │   └── glyphicons-halflings-regular.woff
│       └── js
│           ├── bootstrap.js
│           └── bootstrap.min.js
├── templates
│   ├── home.html
│   └── list.html
├── tests.py
├── urls.py
└── views.py
```

If we have a look at the "Getting Started" section of the Bootstrap documentation, you'll see it wants our HTML template to include something like this:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Bootstrap 101 Template</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <!-- Bootstrap -->
    <link href="css/bootstrap.min.css" rel="stylesheet" media="screen">
  </head>
  <body>
    <h1>Hello, world!</h1>
    <script src="http://code.jquery.com/jquery.js"></script>
    <script src="js/bootstrap.min.js"></script>
  </body>
</html>
```

We already have two HTML templates. We don't want to be adding a whole load of boilerplate code to each, so now feels like the right time to apply the "Don't repeat

yourself" rule, and bring all the common parts together. Thankfully, the Django template language makes that easy using something called template inheritance.

Django Template Inheritance

Let's have a little review of what the differences are between *home.html* and *list.html*:

```
$ diff lists/templates/home.html lists/templates/list.html
7,8c7,8
<         <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">
11a12,18
>
>         <table id="id_list_table">
>             {% for item in list.item_set.all %}
>                 <tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>
>             {% endfor %}
>         </table>
>
```

They have different header texts, and their forms use different URLs. On top of that, *list.html* has the additional `<table>` element.

Now that we're clear on what's in common and what's not, we can make the two templates inherit from a common "superclass" template. We'll start by making a copy of *home.html*:

```
$ cp lists/templates/home.html lists/templates/base.html
```

We make this into a base template which just contains the common boilerplate, and mark out the "blocks", places where child templates can customise it:

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>
```

The base template defines a series of areas called "blocks", which will be places that other templates can hook in and add their own content. Let's see how that works in practice, by changing *home.html* so that it "inherits from" *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 %}
```

You can see that lots of the boilerplate HTML disappears, and we just concentrate on the bits we want to customise. We do the same for *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 %}
```

That's a refactor of the way our templates work. We rerun the FTs to make sure we haven't broken anything…

```
AssertionError: 110.0 != 512 within 5 delta
```

Sure enough, they're still getting to exactly where they were before. That's worthy of a commit:

```
$ git diff -b
# the -b means ignore whitespace, useful since we've changed some html indenting
$ git status
$ git add lists/templates # leave static, for now
$ git commit -m"refactor templates to use a base template"
```

Integrating Bootstrap

Now it's much easier to integrate the boilerplate code that Bootstrap wants—we won't add the JavaScript yet, just the CSS:

lists/templates/base.html (ch07l006).

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>To-Do lists</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link href="css/bootstrap.min.css" rel="stylesheet" media="screen">
</head>
[...]
```

Rows and Columns

Finally, let's actually use some of the Bootstrap magic! You'll have to read the documentation yourself, but should be able to use a combination of the grid system and the `text-center` class to get what we want:

lists/templates/base.html (ch07l007).

```
<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>
```

(If you've never seen an HTML tag broken up over several lines, that `<input>` may be a little shocking. It is definitely valid, but you don't have to use it if you find it offensive. ;)

Take the time to browse through the Bootstrap documentation, if you've never seen it before. It's a shopping trolley brimming full of useful tools to use in your site.

Does that work?

```
AssertionError: 110.0 != 512 within 5 delta
```

Hmm. No. Why isn't our CSS loading?

Static Files in Django

Django, and indeed any web server, needs to know two things to deal with static files:

1. How to tell when a URL request is for a static file, as opposed to for some HTML that's going to be served via a view function
2. Where to find the static file the user wants

In other words, static files are a mapping from URLs to files on disk.

For item 1, Django lets us define a URL "prefix" to say that any URLs which start with that prefix should be treated as requests for static files. By default, the prefix is `/static/`. It's defined in *settings.py*:

superlists/settings.py.

```
[...]

# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/1.7/howto/static-files/

STATIC_URL = '/static/'
```

The rest of the settings we will add to this section are all to do with item 2: finding the actual static files on disk.

While we're using the Django development server (`manage.py runserver`), we can rely on Django to magically find static files for us—it'll just look in any subfolder of one of our apps called *static*.

You now see why we put all the Bootstrap static files into *lists/static*. So why are they not working at the moment? It's because we're not using the `/static/` URL prefix. Have another look at the link to the CSS in *base.html*:

lists/templates/base.html.

```
<link href="css/bootstrap.min.css" rel="stylesheet" media="screen">
```

To get this to work, we need to change it to:

lists/templates/base.html.

```
<link href="/static/bootstrap/css/bootstrap.min.css" rel="stylesheet" media="screen">
```

When `runserver` sees the request, it knows that it's for a static file because it begins with `/static/`. It then tries to find a file called `bootstrap/css/bootstrap.min.css`, looking in each of our app folders for subfolders called *static*, and it should find it at *lists/static/bootstrap/css/bootstrap.min.css*.

So if you take a look manually, you should see it works, as in Figure 7-2.

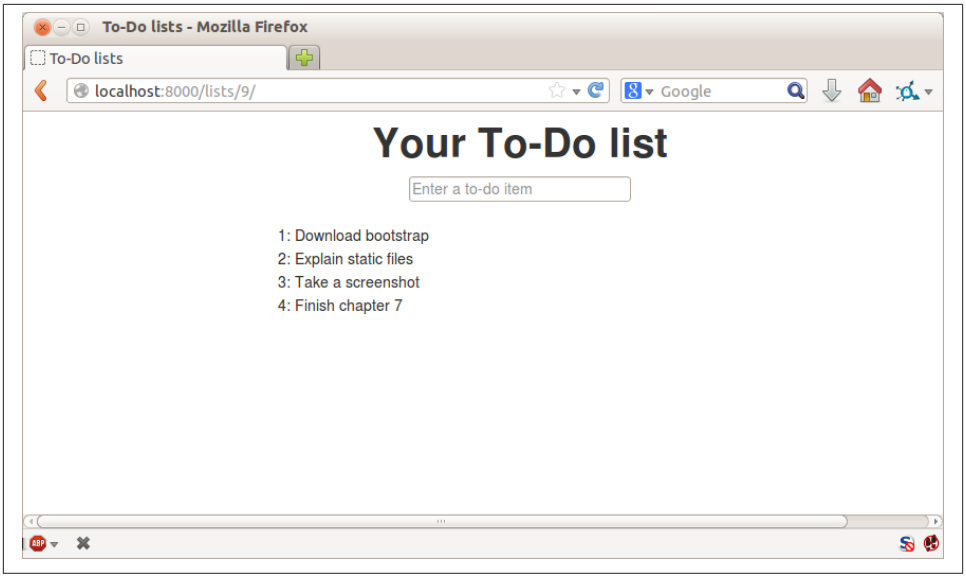


Figure 7-2. Our site starts to look a little better…

Switching to StaticLiveServerCase

If you run the FT though, it won't pass:

```
AssertionError: 110.0 != 512 within 5 delta
```

That's because, although `runserver` automagically finds static files, `LiveServerTestCase` doesn't. Never fear though, the Django developers have made a more magical test class called `StaticLiveServerCase` (see the docs).

Let's switch to that:

functional_tests/tests.py.

```
@@ -1,8 +1,8 @@
-from django.test import LiveServerTestCase
+from django.contrib.staticfiles.testing import StaticLiveServerCase
 from selenium import webdriver
 from selenium.webdriver.common.keys import Keys

-class NewVisitorTest(LiveServerTestCase):
+class NewVisitorTest(StaticLiveServerCase):
```

And now it will now find the new CSS, which will get our test to pass:

```
$ python3 manage.py test functional_tests
Creating test database for alias 'default'...
..
 ---------------------------------------------------------------------
Ran 2 tests in 9.764s
```

At this point, Windows users may see some (harmless, but distracting) error messages that say `socket.error: [WinError 10054] An existing connection was forcibly closed by the remote host`. Add a `self.browser.refresh()` just before the `self.browser.quit()` in `tearDown` to get rid of them. The issue is being tracked in this bug on the Django tracker.

Hooray!

Using Bootstrap Components to Improve the Look of the Site

Let's see if we can do even better, using some of the other tools in Bootstrap's panoply.

Jumbotron!

Bootstrap has a class called `jumbotron` for things that are meant to be particularly prominent on the page. Let's use that to embiggen the main page header and the input form:

lists/templates/base.html (ch07l009).

```
<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 %}">
            [...]
```

When hacking about with design and layout, it's best to have a window open that we can hit refresh on, frequently. Use `python3 manage.py runserver` to spin up the dev server, and then browse to *http://localhost:8000* to see your work as we go.

Large Inputs

The jumbotron is a good start, but now the input box has tiny text compared to everything else. Thankfully, Bootstrap's form control classes offer an option to set an input to be "large":

lists/templates/base.html (ch07l010).

```
<input name="item_text" id="id_new_item"
       class="form-control input-lg"
       placeholder="Enter a to-do item"
/>
```

Table Styling

The table text also looks too small compared to the rest of the page now. Adding the Bootstrap `table` class improves things:

lists/templates/list.html (ch07l011).

```
        <table id="id_list_table" class="table">
```

Using Our Own CSS

Finally I'd like to just offset the input from the title text slightly. There's no ready-made fix for that in Bootstrap, so we'll make one ourselves. That will require specifying our own CSS file:

lists/templates/base.html.

```
<head>
    <title>To-Do lists</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link href="/static/bootstrap/css/bootstrap.min.css" rel="stylesheet" media="screen">
    <link href="/static/base.css" rel="stylesheet" media="screen">
</head>
```

We create a new file at *lists/static/base.css*, with our new CSS rule. We'll use the `id` of the input element, `id_new_item`, to find it and give it some styling:

lists/static/base.css.

```
#id_new_item {
    margin-top: 2ex;
}
```

All that took me a few goes, but I'm reasonably happy with it now (Figure 7-3).

If you want to go further with customising Bootstrap, you need to get into compiling LESS. I *definitely* recommend taking the time to do that some day. LESS and other pseudo-CSS-alikes like SCSS are a great improvement on plain old CSS, and a useful tool even if you don't use Bootstrap. I won't cover it in this book, but you can find resources on the Internets. Here's one, for example.

A last run of the functional tests, to see if everything still works OK?

```
$ python3 manage.py test functional_tests
Creating test database for alias 'default'...
..
 ---------------------------------------------------------------------
Ran 2 tests in 10.084s

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

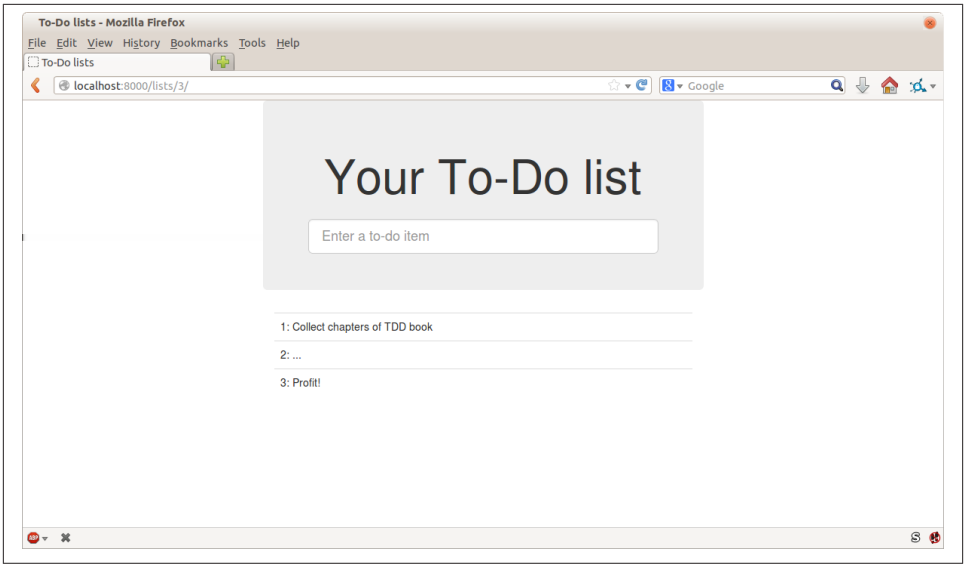


Figure 7-3. The lists page, with all big chunks…

That's it! Definitely time for a commit:

```
$ git status # changes tests.py, base.html, list.html + untracked lists/static
$ git add .
$ git status # will now show all the bootstrap additions
$ git commit -m"Use Bootstrap to improve layout"
```

What We Glossed Over: collectstatic and Other Static Directories

We saw earlier that the Django dev server will magically find all your static files inside app folders, and serve them for you. That's fine during development, but when you're running on a real web server, you don't want Django serving your static content—using Python to serve raw files is slow and inefficient, and a web server like Apache or Nginx can do this all for you. You might even decide to upload all your static files to a CDN, instead of hosting them yourself.

For these reasons, you want to be able to gather up all your static files from inside their various app folders, and copy them into a single location, ready for deployment. This is what the `collectstatic` command is for.

The destination, the place where the collected static files go, is defined in *settings.py* as `STATIC_ROOT`. In the next chapter we'll be doing some deployment, so let's actually experiment with that now. We'll change its value to a folder just outside our repo—I'm going to make it a folder just next to the main source folder:

```
workspace
├── superlists
│   ├── lists
│   │   ├── models.py
│   │
│   ├── manage.py
│   ├── superlists
│
├── static
│   ├── base.css
│   ├── etc...
```

The logic is that the static files folder shouldn't be a part of your repository—we don't want to put it under source control, because it's a duplicate of all the files that are inside *lists/static*.

Here's a neat way of specifying that folder, making it relative to the location of the *settings.py* file:

superlists/settings.py (ch07l018).

```
# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/1.7/howto/static-files/

STATIC_URL = '/static/'
STATIC_ROOT = os.path.abspath(os.path.join(BASE_DIR, '../static'))
```

Take a look at the top of the settings file, and you'll see how that `BASE_DIR` variable is helpfully defined for us. Now let's try running `collectstatic`:

```
$ python3 manage.py collectstatic

You have requested to collect static files at the destination
location as specified in your settings:

/workspace/static

This will overwrite existing files!
Are you sure you want to do this?

Type 'yes' to continue, or 'no' to cancel:
yes

[...]
Copying '/workspace/superlists/lists/static/bootstrap/fonts/glyphicons-halfling
s-regular.svg'

74 static files copied to '/workspace/static'.
```

And if we look in *../static*, we'll find all our CSS files:

```
$ tree ../static/
../static/
├── admin
│   ├── css
```

```
|   |   ├── base.css

[...]

|       └── urlify.js
├── base.css
└── bootstrap
    ├── css
    |   ├── bootstrap.css
    |   ├── bootstrap.min.css
    |   ├── bootstrap-theme.css
    |   └── bootstrap-theme.min.css
    ├── fonts
    |   ├── glyphicons-halflings-regular.eot
    |   ├── glyphicons-halflings-regular.svg
    |   ├── glyphicons-halflings-regular.ttf
    |   └── glyphicons-halflings-regular.woff
    └── js
        ├── bootstrap.js
        └── bootstrap.min.js

10 directories, 74 files
```

`collectstatic` has also picked up all the CSS for the admin site. It's one of Django's powerful features, and we'll find out all about it one day, but we're not ready to use that yet, so let's disable it for now:

superlists/settings.py.

```
INSTALLED_APPS = (
    #'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'lists',
)
```

And we try again:

```
$ rm -rf ../static/
$ python3 manage.py collectstatic --noinput
Copying '/workspace/superlists/lists/static/base.css'
Copying '/workspace/superlists/lists/static/bootstrap/js/bootstrap.min.js'
Copying '/workspace/superlists/lists/static/bootstrap/js/bootstrap.js'
[...]

13 static files copied to '/workspace/static'.
```

Much better.

Anyway, now we know how to collect all the static files into a single folder, where it's easy for a web server to find them. We'll find out all about that, including how to test it, in the next chapter!

For now let's save our changes to *settings.py*:

```
$ git diff # should show changes in settings.py*
$ git commit -am"set STATIC_ROOT in settings and disable admin"
```

A Few Things That Didn't Make It

Inevitably this was only a whirlwind tour of styling and CSS, and there were several topics that I'd hoped to cover in more depth that didn't make it. Here's a few candidates for further study:

- Customising bootstrap with LESS
- The `{% static %}` template tag, for more DRY and less hard-coded URLs
- Client-side packaging tools, like bower

Recap: On Testing Design and Layout

The short answer is: you shouldn't write tests for design and layout. It's too much like testing a constant, and any tests you write are likely to be brittle.

With that said, the *implementation* of design and layout involves something quite tricky: CSS and static files. As a result, it is valuable to have some kind of minimal "smoke test" which checks that your static files and CSS are working. As we'll see in the next chapter, it can help pick up problems when you deploy your code to production.

Similarly, if a particular piece of styling required a lot of client-side JavaScript code to get it to work (dynamic resizing is one I've spent a bit of time on), you'll definitely want some tests for that.

So be aware that this is a dangerous area. Try and write the minimal tests that will give you confidence that your design and layout is working, without testing *what* it actually is. Try and leave yourself in a position where you can freely make changes to the design and layout, without having to go back and adjust tests all the time.

CHAPTER 8
Testing Deployment Using a Staging Site

Is all fun and game until you are need of put it in production.

— Devops Borat

It's time to deploy the first version of our site and make it public. They say that if you wait until you feel ready to ship, then you've waited too long.

Is our site usable? Is it better than nothing? Can we make lists on it? Yes, yes, yes.

No, you can't log in yet. No, you can't mark tasks as completed. But do we really need any of that stuff? Not really—and you can never be sure what your users are *actually* going to do with your site once they get their hands on it. We think our users want to use the site for to-do lists, but maybe they actually want to use it to make "top 10 best fly-fishing spots" lists, for which you don't need any kind of "mark completed" function. We won't know until we put it out there.

In this chapter we're going to go through and actually deploy our site to a real, live web server.

You might be tempted to skip this chapter—there's lots of daunting stuff in it, and maybe you think this isn't what you signed up for. But I *strongly* urge you to give it a go. This is one of the chapters I'm most pleased with, and it's one that people often write to me saying they were really glad they stuck through it.

If you've never done a server deployment before, it will demystify a whole world for you, and there's nothing like the feeling of seeing your site live on the actual Internet. Give it a buzzword name like "DevOps" if that's what it takes to convince you it's worth it.

Why not ping me a note once your site is live on the web, and send me the URL? It always gives me a warm and fuzzy feeling … *obeythetestinggoat@gmail.com*.

TDD and the Danger Areas of Deployment

Deploying a site to a live web server can be a tricky topic. Oft-heard is the forlorn cry —"*but it works on my machine!*".

Some of the danger areas of deployment include:

Static files (CSS, JavaScript, images, etc.)
: Web servers usually need special configuration for serving these.

The database
: There can be permissions and path issues, and we need to be careful about preserving data between deploys.

Dependencies
: We need to make sure that the packages our software relies on are installed on the server, and have the correct versions.

But there are solutions to all of these. In order:

- Using a *staging site*, on the same infrastructure as the production site, can help us test out our deployments and get things right before we go to the "real" site.
- We can also *run our functional tests against the staging site*. That will reassure us that we have the right code and packages on the server, and since we now have a "smoke test" for our site layout, we'll know that the CSS is loaded correctly.
- *Virtualenvs* are a useful tool for managing packages and dependencies on a machine that might be running more than one Python application.
- And finally, *automation, automation, automation*. By using an automated script to deploy new versions, and by using the same script to deploy to staging and production, we can reassure ourselves that staging is as much like live as possible.[1]

Over the next few pages I'm going to go through *a* deployment procedure. It isn't meant to be the *perfect* deployment procedure, so please don't take it as being best practice, or a recommendation—it's meant to be an illustration, to show the kinds of issues involved in deployment and where testing fits in.

1. What I'm calling a "staging" server, some people would call a "development" server, and some others would also like to distinguish "preproduction" servers. Whatever we call it, the point is to have somewhere we can try our code out in an environment that's as similar as possible to the real production server.

Chapter Overview

There's lots of stuff in this chapter, so here's an overview to help you keep your bearings:

1. Adapt our FTs so they can run against a staging server.
2. Spin up a server, install all the required software on it, and point our staging and live domains at it.
3. Upload our code to the server using Git.
4. Try and get a quick & dirty version of our site running on the staging domain using the Django dev server.
5. Learn how to use a virtualenv to manage our project's Python dependencies on the server.
6. As we go, we'll keep running our FT, to tell us what's working and what's not.
7. Move from our quick & dirty version to a production-ready configuration, using Gunicorn, Upstart, and domain sockets.
8. Once we have a working config, we'll write a script to automate the process we've just been through manually, so that we can deploy our site automatically in future.
9. Finally we'll use this script to deploy the production version of our site on its real domain.

As Always, Start with a Test

Let's adapt our functional tests slightly so that it can be run against a staging site. We'll do it by slightly hacking an argument that is normally used to change the address which the test's temporary server gets run on:

functional_tests/tests.py (ch08l001).

```
import sys
[...]

class NewVisitorTest(StaticLiveServerCase):

    @classmethod
    def setUpClass(cls):  #❶
        for arg in sys.argv:  #❷
            if 'liveserver' in arg:  #❸
                cls.server_url = 'http://' + arg.split('=')[1]  #❹
                return  #❺
        super().setUpClass()
        cls.server_url = cls.live_server_url

    @classmethod
    def tearDownClass(cls):
```

```
        if cls.server_url == cls.live_server_url:
            super().tearDownClass()


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

OK, when I said slightly hacking, I meant seriously hacking. Do you remember I said that `LiveServerTestCase` had certain limitations? Well, one is that it always assumes you want to use its own test server. I still want to be able to do that sometimes, but I also want to be able to selectively tell it not to bother, and to use a real server instead.

❶ `setUpClass` is a similar method to `setUp`, also provided by `unittest`, which is used to do test setup for the whole class—that means it only gets executed once, rather than before every test method. This is where `LiveServerTestCase`/`StaticLiveServerCase` usually starts up its test server.

❷ ❸ We look for the *liveserver* command-line argument (which is found in `sys.argv`).

❹ ❺ If we find it, we tell our test class to skip the normal `setUpClass`, and just store away our staging server URL in a variable called `server_url` instead.

This means we also need to change the three places we used to use `self.live_server_url`:

functional_tests/tests.py (ch08l002).

```
    def test_can_start_a_list_and_retrieve_it_later(self):
        # Edith has heard about a cool new online to-do app. She goes
        # to check out its homepage
        self.browser.get(self.server_url)
        [...]
        # Francis visits the home page.  There is no sign of Edith's
        # list
        self.browser.get(self.server_url)
        [...]

    def test_layout_and_styling(self):
        # Edith goes to the home page
        self.browser.get(self.server_url)
```

We test that our little hack hasn't broken anything by running the functional tests "normally":

```
$ python3 manage.py test functional_tests
[...]
Ran 2 tests in 8.544s

OK
```

And now we can try them against our staging server URL. I'm hosting my staging server at *superlists-staging.ottg.eu*:

```
$ python3 manage.py test functional_tests --liveserver=superlists-staging.ottg.eu
Creating test database for alias 'default'...
FE
======================================================================
FAIL: test_can_start_a_list_and_retrieve_it_later
(functional_tests.tests.NewVisitorTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/workspace/superlists/functional_tests/tests.py", line 42, 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_layout_and_styling (functional_tests.tests.NewVisitorTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File
"/workspace/superlists/functional_tests/tests.py", line 114, in
test_layout_and_styling
    inputbox = self.browser.find_element_by_id('id_new_item')
[...]
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"id","selector":"id_new_item"}' ; Stacktrace:
 ----------------------------------------------------------------------
Ran 2 tests in 16.480s

FAILED (failures=2)
Destroying test database for alias 'default'...
```

You can see that both tests are failing, as expected, since I haven't actually set up my staging site yet. In fact, you can see from the first traceback that the test is actually ending up on the home page of my domain registrar.

The FT seems to be testing the right things though, so let's commit:

```
$ git diff # should show changes to functional_tests.py
$ git commit -am "Hack FT runner to be able to test staging"
```

Getting a Domain Name

We're going to need a couple of domain names at this point in the book—they can both be subdomains of a single domain. I'm going to use *superlists.ottg.eu* and *superlists-staging.ottg.eu*. If you don't already own a domain, this is the time to register one! Again, this is something I really want you to *actually* do. If you've never registered a domain before, just pick any old registrar and buy a cheap one—it should only cost you $5 or so, and you can even find free ones. I promise seeing your site on a "real" web site will be a thrill.

Manually Provisioning a Server to Host Our Site

We can separate out "deployment" into two tasks:

- *Provisioning* a new server to be able to host the code
- *Deploying* a new version of the code to an existing server

Some people like to use a brand new server for every deployment—it's what we do at PythonAnywhere. That's only necessary for larger, more complex sites though, or major changes to an existing site. For a simple site like ours, it makes sense to separate the two tasks. And, although we eventually want both to be completely automated, we can probably live with a manual provisioning system for now.

As you go through this chapter, you should be aware that provisioning is something that varies a lot, and that as a result there are few universal best practices for deployment. So, rather than trying to remember the specifics of what I'm doing here, you should be trying to understand the rationale, so that you can apply the same kind of thinking in the specific future circumstances you encounter.

Choosing Where to Host Our Site

There are loads of different solutions out there these days, but they broadly fall into two camps:

- Running your own (possibly virtual) server
- Using a Platform-As-A-Service (PaaS) offering like Heroku, DotCloud, OpenShift, or PythonAnywhere

Particularly for small sites, a PaaS offers a lot of advantages, and I would definitely recommend looking into them. We're not going to use a PaaS in this book however, for several reasons. Firstly, I have a conflict of interest, in that I think PythonAnywhere is the best, but then again I would say that because I work there. Secondly, all the PaaS offerings are quite different, and the procedures to deploy to each vary a lot—learning about one doesn't necessarily tell you about the others. Any one of them might change their process radically, or simply go out of business by the time you get to read this book.

Instead, we'll learn just a tiny bit of good old-fashioned server admin, including SSH and web server config. They're unlikely to ever go away, and knowing a bit about them will get you some respect from all the grizzled dinosaurs out there.

What I have done is to try and set up a server in such a way that it's a lot like the environment you get from a PaaS, so you should be able to apply the lessons we learn in the deployment section, no matter what provisioning solution you choose.

Spinning Up a Server

I'm not going to dictate how you do this—whether you choose Amazon AWS, Rackspace, Digital Ocean, your own server in your own data centre or a Raspberry Pi in a cupboard behind the stairs, any solution should be fine, as long as:

- Your server is running Ubuntu (13.04 or later).
- You have root access to it.
- It's on the public Internet.
- You can SSH into it.

I'm recommending Ubuntu as a distro because it has Python 3.4 and it has some specific ways of configuring Nginx, which I'm going to make use of next. If you know what you're doing, you can probably get away with using something else, but you're on your own.

Some people get to this chapter, and are tempted to skip the domain bit, and the "getting a real server" bit, and just use a VM on their own PC. Don't do this. It's *not* the same, and you'll have more difficulty following the instructions, which are complicated enough as it is. If you're worried about cost, dig around and you'll find free options for both. Email me if you need further pointers, I'm always happy to help.

User Accounts, SSH, and Privileges

In these instructions, I'm assuming that you have a nonroot user account set up that has "sudo" privileges, so whenever we need to do something that requires root access, we use sudo, and I'm explicit about that in the various instructions below. If you need to create a nonroot user, here's how:

```
# these commands must be run as root
root@server:$ useradd -m -s /bin/bash elspeth # add user named elspeth
# -m creates a home folder, -s sets elspeth to use bash by default
root@server:$ usermod -a -G sudo elspeth # add elspeth to the sudoers group
root@server:$ passwd elspeth # set password for elspeth
root@server:$ su - elspeth # switch-user to being elspeth!
elspeth@server:$
```

Name your own user whatever you like! I also recommend learning up how to use private key authentication rather than passwords for SSH. It's a matter of taking the public key from your own PC, and appending it to *~/.ssh/authorized_keys* in the user account on the server. You probably went through a similar procedure if you signed up for Bitbucket or Github.

There are some good instructions here (note that `ssh-keygen` *is* available as part of Git-Bash on Windows).

Look out for that `elspeth@server` in the command-line listings in this chapter. It indicates commands that must be run on the server, as opposed to commands you run on your own PC.

Installing Nginx

We'll need a web server, and all the cool kids are using Nginx these days, so we will too. Having fought with Apache for many years, I can tell you it's a blessed relief in terms of the readability of its config files, if nothing else!

Installing Nginx on my server was a matter of doing an `apt-get`, and I could then see the default Nginx "Hello World" screen:

```
elspeth@server:$ sudo apt-get install nginx
elspeth@server:$ sudo service nginx start
```

(You may need to do an `apt-get update` and/or an `apt-get upgrade` first.)

You should be able to go to the IP address of your server, and see the "Welcome to nginx" page at this point, as in Figure 8-1.

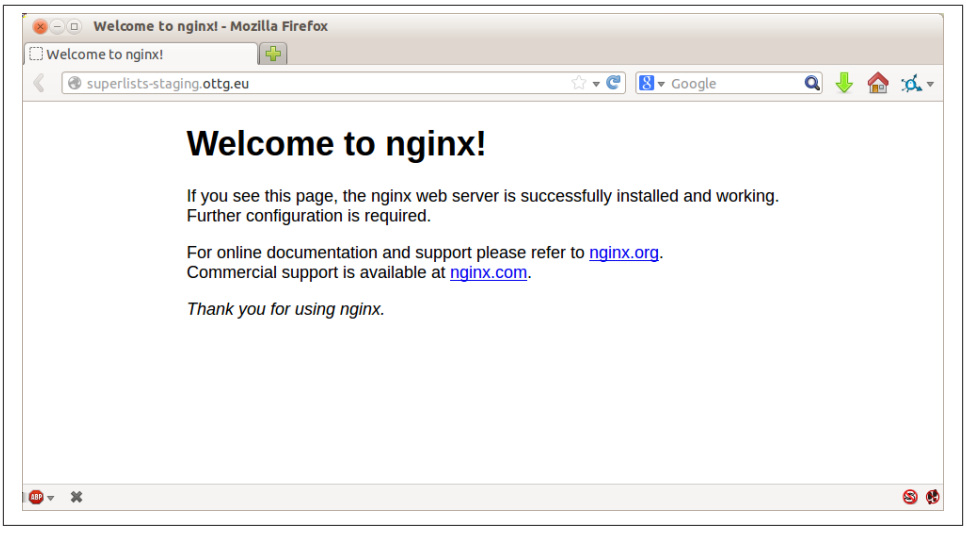


Figure 8-1. Nginx—it works!

If you don't see it, it may be because your firewall does not open port 80 to the world. On AWS for example, you may need to configure the "security group" for your server to open port 80.

While we've got root access, let's make sure the server has the key pieces of software we need at the system level: Python, Git, pip, and virtualenv.

```
elspeth@server:$ sudo apt-get install git python3 python3-pip
elspeth@server:$ sudo pip3 install virtualenv
```

Configuring Domains for Staging and Live

We don't want to be messing about with IP addresses all the time, so we should point our staging and live domains to the server. At my registrar, the control screens looked a bit like Figure 8-2.

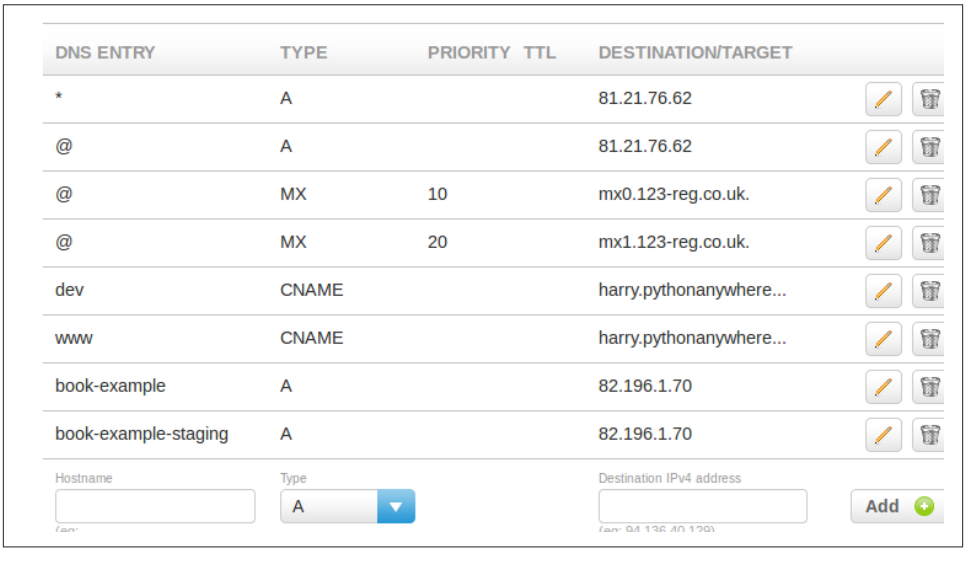


Figure 8-2. Domain setup

In the DNS system, pointing a domain at a specific IP address is called an "A-Record". All registrars are slightly different, but a bit of clicking around should get you to the right screen in yours.

Using the FT to Confirm the Domain Works and Nginx Is Running

To confirm this works, we can rerun our functional tests and see that their failure messages have changed slightly—one of them in particular should now mention Nginx:

```
$ python3 manage.py test functional_tests --liveserver=superlists-staging.ottg.eu
[...]
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"id","selector":"id_new_item"}' ; Stacktrace:
[...]
AssertionError: 'To-Do' not found in 'Welcome to nginx!'
```

Progress!

Deploying Our Code Manually

The next step is to get a copy of the staging site up and running, just to check whether we can get Nginx and Django to talk to each other. As we do so, we're starting to move into doing "deployment" rather than provisioning, so we should be thinking about how we can automate the process, as we go.

One rule of thumb for distinguishing provisioning from deployment is that you tend to need root permissions for the former, but we don't for the latter.

We need a directory for the source to live in. Let's assume we have a home folder for a nonroot user; in my case it would be at */home/elspeth* (this is likely to be the setup on any shared hosting system, but you should always run your web apps as a nonroot user, in any case). I'm going to set up my sites like this:

```
/home/elspeth
├── sites
│   ├── www.live.my-website.com
│   │   ├── database
│   │   │   └── db.sqlite3
│   │   ├── source
│   │   │   ├── manage.py
│   │   │   ├── superlists
│   │   │   ├── etc...
│   │   │
│   │   ├── static
│   │   │   ├── base.css
│   │   │   ├── etc...
│   │   │
│   │   └── virtualenv
│   │       ├── lib
│   │       ├── etc...
│   │
│   ├── www.staging.my-website.com
│   │   ├── database
│   │   ├── etc...
```

Each site (staging, live, or any other website) has its own folder. Within that we have a separate folder for the source code, the database, and the static files. The logic is that, while the source code might change from one version of the site to the next, the database will stay the same. The static folder is in the same relative location, *../static*, that we set up at the end of the last chapter. Finally, the virtualenv gets its own subfolder too. What's a virtualenv, I hear you ask? We'll find out shortly.

Adjusting the Database Location

First let's change the location of our database in *settings.py*, and make sure we can get that working on our local PC. Using `os.path.abspath` prevents any later confusion about the current working directory:

superlists/settings.py (ch08l003).

```
# Build paths inside the project like this: os.path.join(BASE_DIR, ...)
import os
BASE_DIR = os.path.abspath(os.path.dirname(os.path.dirname(__file__)))
[...]

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, '../database/db.sqlite3'),
    }
}
[...]

STATIC_ROOT = os.path.join(BASE_DIR, '../static')
```

Now let's try it locally:

```
$ mkdir ../database
$ python3 manage.py migrate --noinput
Creating tables ...
[...]
$ ls ../database/
db.sqlite3
```

That seems to work. Let's commit it:

```
$ git diff # should show changes in settings.py
$ git commit -am "move sqlite database outside of main source tree"
```

To get our code onto the server, we'll use Git and go via one of the code sharing sites. If you haven't already, push your code up to GitHub, BitBucket, or similar. They all have excellent instructions for beginners on how to do that.

Here's some bash commands that will set this all up. If you're not familiar with it, note the `export` command which lets me set up a "local variable" in 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/$SITENAME/virtualenv
# you should replace the URL in the next line with the URL for your own repo
elspeth@server:$ git clone https://github.com/hjwp/book-example.git \
~/sites/$SITENAME/source
Resolving deltas: 100% [...]
```

A bash variable defined using `export` only lasts as long as that console session. If you log out of the server and log back in again, you'll need to redefine it. It's devious because Bash won't error, it will just substitute the empty string for the variable, which will lead to weird results … if in doubt, do a quick **`echo $SITENAME`**.

Now we've got the site installed, let's just try running the dev server—this is a smoke test, to see if all the moving parts are connected:

```
elspeth@server:$ $ cd ~/sites/$SITENAME/source
$ python3 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. Django isn't installed on the server.

Creating a Virtualenv

We could install it at this point, but that would leave us with a problem: if we ever wanted to upgrade Django when a new version comes out, it would be impossible to test the staging site with a different version from live. Similarly, if there are other users on the server, we'd all be forced to use the same version of Django.

The solution is a "virtualenv"—a neat way of having different versions of Python packages installed in different places, in their own "virtual environments".

Let's try it out locally, on our own PC first:

```
$ pip3 install virtualenv # will need a sudo on linux/macos.
```

We'll follow the same folder structure as we're planning for the server:

```
$ virtualenv --python=python3 ../virtualenv
$ ls ../virtualenv/
bin  include  lib
```

That will create a folder at *../virtualenv* which will contain its own copy of Python and `pip`, as well as a location to install Python packages to. It's a self-contained "virtual" Python environment. To start using it, we run a script called `activate`, which will change the system path and the Python path in such a way as to use the virtualenv's executables and packages:

```
$ which python3
/usr/bin/python3
$ source ../virtualenv/bin/activate
$ which python  # note switch to virtualenv Python
/workspace/virtualenv/bin/python
(virtualenv)$ python3 manage.py test lists
[...]
ImportError: No module named 'django'
```

It's not required, but you might want to look into a tool called `virtualenvwrapper` for managing virtualenvs on your own PC.

Virtualenvs on Windows

On Windows, things are slightly different. There are two main things to watch out for:

- The *virtualenv/bin* folder is called *virtualenv/Scripts*, so you should substitute that in as appropriate.
- When using Git-Bash, do not try and run *activate.bat*—it is written for the DOS shell. Use `source ..\virtualenv\Scripts\activate`. The `source` is important.

We're seeing that `ImportError: No module named django` because Django isn't installed inside the virtualenv. So, we can install it, and see that it ends up inside the virtualenv's *site-packages* folder:

```
(virtualenv)$ pip install django==1.7
[...]
Successfully installed django
Cleaning up...
(virtualenv)$ python3 manage.py test lists
[...]
OK
$ ls ../virtualenv/lib/python3.4/site-packages/
django                      pip                      setuptools
Django-1.7-py3.4.egg-info   pip-1.4.1-py3.4.egg-info  setuptools-0.9.8-py3.4.egg-info
easy_install.py             pkg_resources.py
_markerlib                  __pycache__
```

To "save" the list of packages we need in our virtualenv, and be able to re-create it later, we create a *requirements.txt* file, using `pip freeze`, and add that to our repository:

```
(virtualenv)$ pip freeze > requirements.txt
(virtualenv)$ deactivate
$ cat requirements.txt
Django==1.7
```

```
$ git add requirements.txt
$ git commit -m"Add requirements.txt for virtualenv"
```

While Django 1.7 isn't quite out yet, you can use `pip install https://github.com/django/django/archive/stable/1.7.x.zip`, and just add the URL instead of the `"Django==1.7"` to *requirements.txt*; pip is magic enough to figure that out. Test it by doing a **`pip install -r requirements.txt`** locally, and you should see pip concluding everything is already installed.

And now we do a `git push` to send our updates up to our code-sharing site:

```
$ git push
```

And we can pull those changes down to the server, create a virtualenv on the server, and use *requirements.txt* along with `pip install -r` to make the server virtualenv just like our local one:

```
elspeth@server:$ git pull  # may ask you to do some git config first
elspeth@server:$ virtualenv --python=python3 ../virtualenv/
elspeth@server:$ ../virtualenv/bin/pip install -r requirements.txt
Downloading/unpacking Django==1.7 (from -r requirements.txt (line 1))
[...]
Successfully installed Django
Cleaning up...
elspeth@server:$ ../virtualenv/bin/python3 manage.py runserver
Validating models...
0 errors found
[...]
```

That looks like it's running happily. We can Ctrl-C it for now.

Notice you don't have to use the `activate` to use the virtualenv. Directly specifying the path to the virtualenv copies of `python` or `pip` works too. We'll use the direct paths on the server.

Most people like to create a virtualenv for a project as soon as they start it. I only waited until now because I wanted to keep the first few chapters as simple as possible.

Simple Nginx Configuration

Next we create an Nginx config file to tell it to send requests for our staging site along to Django. A minimal config looks like this:

server: /etc/nginx/sites-available/superlists-staging.ottg.eu.

```
server {
    listen 80;
    server_name superlists-staging.ottg.eu;

    location / {
        proxy_pass http://localhost:8000;
    }
}
```

This config says it will only work for our staging domain, and will "proxy" all requests to the local port 8000 where it expects to find Django waiting to respond to requests.

I saved[2] this to a file called *superlists-staging.ottg.eu* inside */etc/nginx/sites-available* folder, and then added it to the enabled sites for the server by creating a symlink to it:

```
elspeth@server:$ echo $SITENAME # check this still has our site in
superlists-staging.ottg.eu
elspeth@server:$ sudo ln -s ../sites-available/$SITENAME \
/etc/nginx/sites-enabled/$SITENAME
elspeth@server:$ ls -l /etc/nginx/sites-enabled # check our symlink is there
```

That's the Debian/Ubuntu preferred way of saving Nginx configurations—the real config file in *sites-available*, and a symlink in *sites-enabled*; the idea is that it makes it easier to switch sites on or off.

We also may as well remove the default "Welcome to nginx" config, to avoid any confusion:

```
elspeth@server:$ sudo rm /etc/nginx/sites-enabled/default
```

And now to test it:

```
elspeth@server:$ sudo service nginx reload
elspeth@server:$ ../virtualenv/bin/python3 manage.py runserver
```

I also had to edit */etc/nginx/nginx.conf* and uncomment a line saying `server_names_hash_bucket_size 64;` to get my long domain name to work. You may not have this problem; Nginx will warn you when you do a `reload` if it has any trouble with its config files.

A quick visual inspection confirms—the site is up (Figure 8-3)!

2. Not sure how to edit a file on the server? There's always vi, which I'll keep encouraging you to learn a bit of. Alternatively, try the relatively beginner-friendly `nano`. Note you'll also need to use `sudo` because the file is in a system folder.

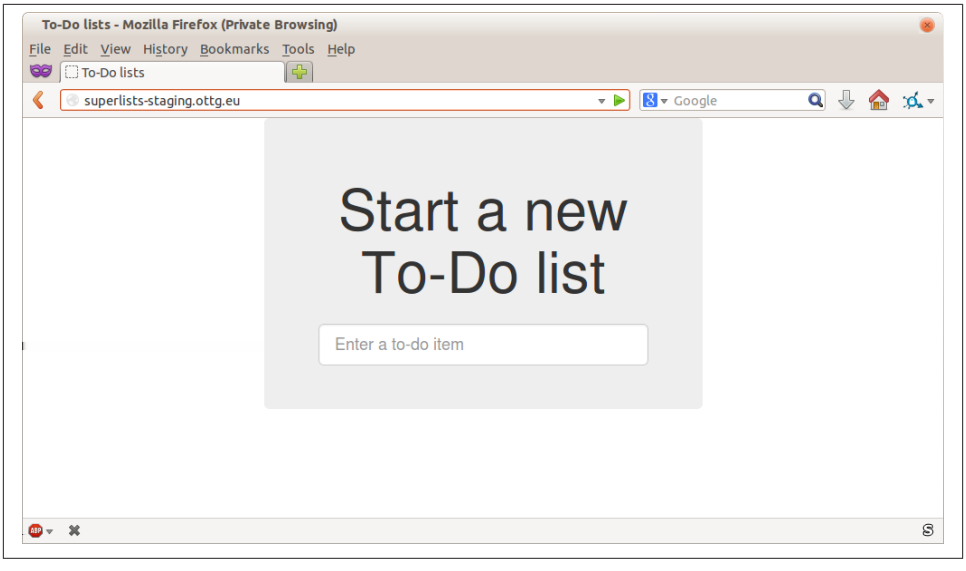


Figure 8-3. The staging site is up!

Let's see what our functional tests say:

```
$ python3 manage.py test functional_tests --liveserver=superlists-staging.ottg.eu
[...]
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
[...]
AssertionError: 0.0 != 512 within 3 delta
```

The tests are failing as soon as they try and submit a new item, because we haven't set up the database. You'll probably have spotted the yellow Django debug page (Figure 8-4) telling us as much as the tests went through, or if you tried it manually.

The tests saved us from potential embarrassment there. The site *looked* fine when we loaded its front page. If we'd been a little hasty, we might have thought we were done, and it would have been the first users that discovered that nasty Django DEBUG page. Okay, slight exaggeration for effect, maybe we *would* have checked, but what happens as the site gets bigger and more complex? You can't check everything. The tests can.

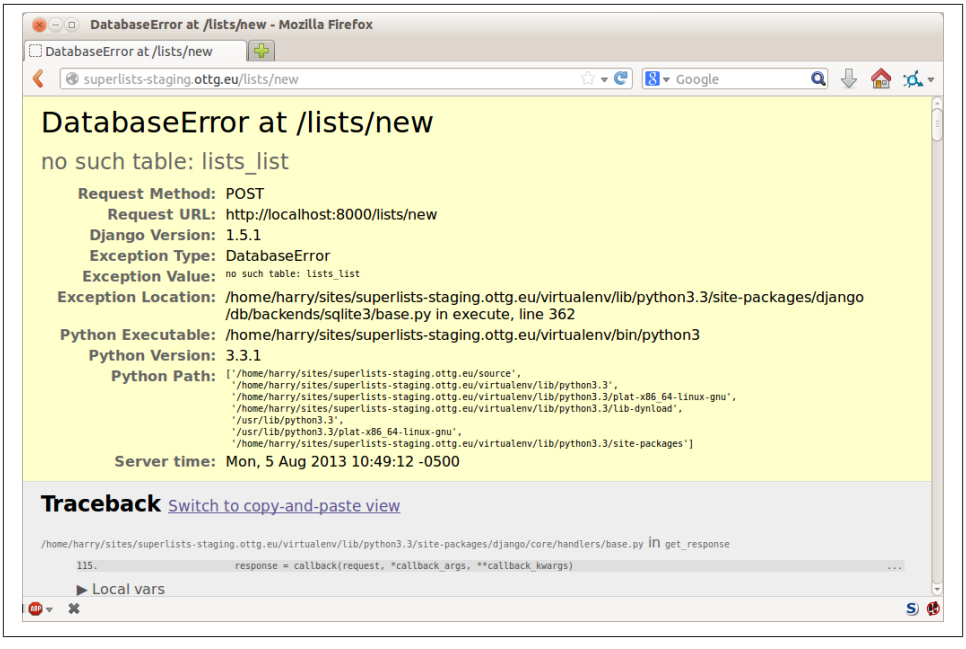


Figure 8-4. But the database isn't

Creating the Database with migrate

We run `migrate` using the `--noinput` argument to suppress the two little "are you sure" prompts:

```
elspeth@server:$ ../virtualenv/bin/python3 manage.py migrate --noinput
Creating tables ...
[...]
elspeth@server:$ ls ../database/
db.sqlite3
elspeth@server:$ ../virtualenv/bin/python3 manage.py runserver
```

Let's try the FTs again:

```
$ python3 manage.py test functional_tests --liveserver=superlists-staging.ottg.eu
Creating test database for alias 'default'...
..
 ----------------------------------------------------------------------
Ran 2 tests in 10.718s

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

It's great to see the site up and running! We might reward ourselves with a well-earned tea break at this point, before moving on to the next section…

If you see a "502 - Bad Gateway", it's probably because you forgot to restart the dev server with `manage.py runserver` after the `migrate`.

Getting to a Production-Ready Deployment

We're at least reassured that the basic piping works, but we really can't be using the Django dev server in production. We also can't be relying on manually starting it up with `runserver`.

Switching to Gunicorn

Do you know why the Django mascot is a pony? The story is that Django comes with so many things you want: an ORM, all sorts of middleware, the admin site … "What else do you want, a pony?" Well, Gunicorn stands for "Green Unicorn", which I guess is what you'd want next if you already had a pony…

```
elspeth@server:$ ../virtualenv/bin/pip install gunicorn
```

Gunicorn will need to know a path to a WSGI server, which is usually a function called `application`. Django provides one in *superlists/wsgi.py*:

```
elspeth@server:$ ../virtualenv/bin/gunicorn superlists.wsgi:application
2013-05-27 16:22:01 [10592] [INFO] Starting gunicorn 0.18.0
2013-05-27 16:22:01 [10592] [INFO] Listening at: http://127.0.0.1:8000 (10592)
[...]
```

If you now take a look at the site, you'll find the CSS is all broken, as in Figure 8-5.

And if we run the functional tests, you'll see they confirm that something is wrong. The test for adding list items passes happily, but the test for layout + styling fails. Good job tests!

```
$ python3 manage.py test functional_tests --liveserver=superlists-staging.ottg.eu
[...]
AssertionError: 125.0 != 512 within 3 delta
FAILED (failures=1)
```

The reason that the CSS is broken is that although the Django dev server will serve static files magically for you, Gunicorn doesn't. Now is the time to tell Nginx to do it instead.

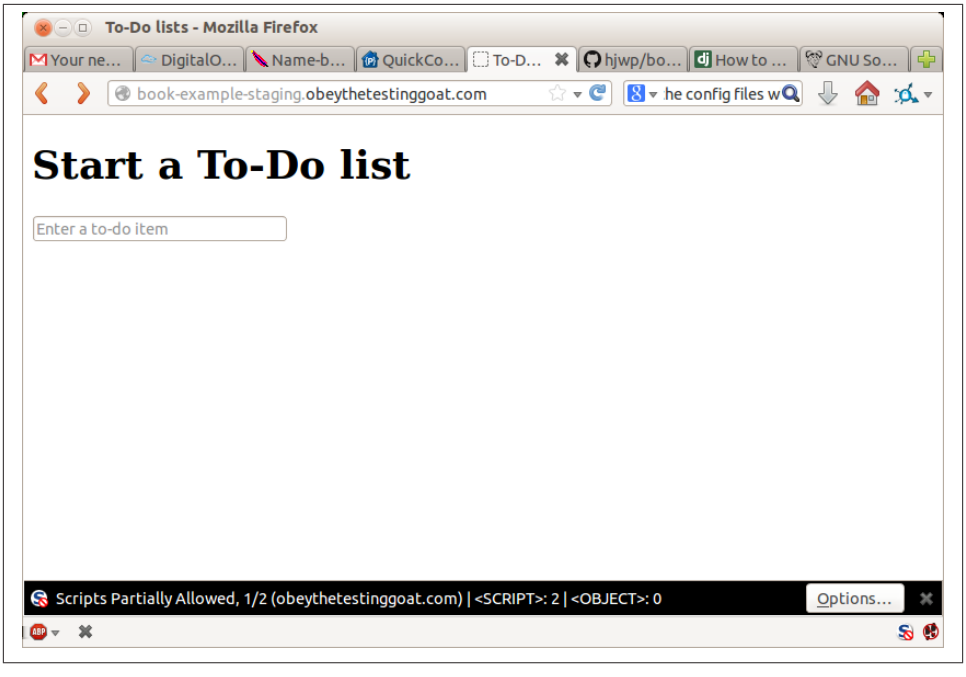


Figure 8-5. Broken CSS

Getting Nginx to Serve Static Files

First we run `collectstatic` to copy all the static files to a folder where Nginx can find them:

```
elspeth@server:$ ../virtualenv/bin/python3 manage.py collectstatic --noinput
elspeth@server:$ ls ../static/
base.css  bootstrap
```

Note that, again, instead of using the virtualenv `activate` command, we can use the direct path to the virtualenv's copy of Python instead.

Now we tell Nginx to start serving those static files for us:

server: /etc/nginx/sites-available/superlists-staging.ottg.eu.

```
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;
```

```
    }
}
```

Reload Nginx and restart Gunicorn…

```
elspeth@server:$ sudo service nginx reload
elspeth@server:$ ../virtualenv/bin/gunicorn superlists.wsgi:application
```

And if we take another look at the site, things are looking much healthier. We can rerun our FTs:

```
$ python3 manage.py test functional_tests --liveserver=superlists-staging.ottg.eu
Creating test database for alias 'default'...
..
 ---------------------------------------------------------------------
Ran 2 tests in 10.718s

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

Switching to Using Unix Sockets

When we want to serve both staging and live, we can't have both servers trying to use port 8000. We could decide to allocate different ports, but that's a bit arbitrary, and it would be dangerously easy to get it wrong and start the staging server on the live port, or vice versa.

A better solution is to use Unix domain sockets—they're like files on disk, but can be used by Nginx and Gunicorn to talk to each other. We'll put our sockets in */tmp*. Let's change the proxy settings in 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` is used to make sure Gunicorn and Django know what domain it's running on. We need that for the `ALLOWED_HOSTS` security feature, which we're about to switch on.

Now we restart Gunicorn, but this time telling it to listen on a socket instead of on the default port:

```
elspeth@server:$ sudo service nginx reload
elspeth@server:$ ../virtualenv/bin/gunicorn --bind \
    unix:/tmp/superlists-staging.ottg.eu.socket superlists.wsgi:application
```

And again, we rerun the functional test again, to make sure things still pass:

```
$ python3 manage.py test functional_tests --liveserver=superlists-staging.ottg.eu
OK
```

A couple more steps!

Switching DEBUG to False and Setting ALLOWED_HOSTS

Django's DEBUG mode is all very well for hacking about on your own server, but leaving those pages full of tracebacks available isn't secure.

You'll find the `DEBUG` setting at the top of *settings.py*. When we set this to `False`, we also need to set another setting called `ALLOWED_HOSTS`. This was added as a security feature in Django 1.5. Unfortunately it doesn't have a helpful comment in the default *settings.py*, but we can add one ourselves. Do this on the server:

server: superlists/settings.py.

```
# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = False

TEMPLATE_DEBUG = DEBUG

# Needed when DEBUG=False
ALLOWED_HOSTS = ['superlists-staging.ottg.eu']
[...]
```

And, once again, we restart Gunicorn and run the FT to check things still work.

Don't commit these changes on the server. At the moment this is just a hack to get things working, not a change we want to keep in our repo. In general, to keep things simple, I'm only going to do Git commits from the local PC, using `git push` and `git pull` when I need to sync them up to the server.

Using Upstart to Make Sure Gunicorn Starts on Boot

Our final step is to make sure that the server starts up Gunicorn automatically on boot, and reloads it automatically if it crashes. On Ubuntu, the way to do this is using Upstart:

server: /etc/init/gunicorn-superlists-staging.ottg.eu.conf.

```
description "Gunicorn server for superlists-staging.ottg.eu"

start on net-device-up  ❶
stop on shutdown

respawn  ❷

setuid elspeth ❸
chdir /home/elspeth/sites/superlists-staging.ottg.eu/source  ❹

exec ../virtualenv/bin/gunicorn \  ❺
```

```
    --bind unix:/tmp/superlists-staging.ottg.eu.socket \
    superlists.wsgi:application
```

Upstart is joyously simple to configure (especially if you've ever had the dubious pleasure of writing an `init.d` script), and is fairly self-explanatory.

❶ `start on net-device-up` makes sure Gunicorn only runs once the server has connected up to the Internet.

❷ `respawn` will restart the process automatically if it crashes.

❸ `setuid` makes the process run as the "elspeth" user.

❹ `chdir` sets the working directory.

❺ `exec` is the actual process to execute.

Upstart scripts live in */etc/init*, and their names must end in *.conf*.

Now we can start Gunicorn with the `start` command:

```
elspeth@server:$ sudo start gunicorn-superlists-staging.ottg.eu
```

And we can rerun the FTs to see that everything still works. You can even test that the site comes back up if you reboot the server!

Saving Our Changes: Adding Gunicorn to Our requirements.txt

Back in the *local* copy of your repo, we should add Gunicorn to the list of packages we need in our virtualenvs:

```
$ source ../virtualenv/bin/activate # if necessary
(virtualenv)$ pip install gunicorn
(virtualenv)$ pip freeze > requirements.txt
(virtualenv)$ deactivate
$ git commit -am "Add gunicorn to virtualenv requirements"
$ git push
```

On Windows, at the time of writing, Gunicorn would pip install quite happily, but it wouldn't actually work if you tried to use it. Thankfully we only ever run it on the server, so that's not a problem. And, Windows support is being discussed…

Automating

Let's recap our provisioning and deployment procedures:

Provisioning

1. Assume we have a user account and home folder

2. `apt-get nginx git python-pip`
3. `pip install virtualenv`
4. Add Nginx config for virtual host
5. Add Upstart job for Gunicorn

Deployment

1. Create directory structure in *~/sites*
2. Pull down source code into folder named *source*
3. Start virtualenv in *../virtualenv*
4. `pip install -r requirements.txt`
5. `manage.py migrate` for database
6. `collectstatic` for static files
7. Set DEBUG = False and ALLOWED_HOSTS in *settings.py*
8. Restart Gunicorn job
9. Run FTs to check everything works

Assuming we're not ready to entirely automate our provisioning process, how should we save the results of our investigation so far? I would say that the Nginx and Upstart config files should probably be saved somewhere, in a way that makes it easy to reuse them later. Let's save them in a new subfolder in our repo:

```
$ mkdir deploy_tools
```

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;
    }
}
```

deploy_tools/gunicorn-upstart.template.conf.

```
description "Gunicorn server for SITENAME"

start on net-device-up
stop on shutdown

respawn
```

```
setuid elspeth
chdir /home/elspeth/sites/SITENAME/source

exec ../virtualenv/bin/gunicorn \
    --bind unix:/tmp/SITENAME.socket \
    superlists.wsgi:application
```

Then it's easy for us to use those two files to generate a new site, by doing a find & replace on `SITENAME`.

For the rest, just keeping a few notes is OK. Why not keep them in a file in the repo too?

deploy_tools/provisioning_notes.md.

```
Provisioning a new site
=======================

## Required packages:

* nginx
* Python 3
* Git
* pip
* virtualenv

eg, on Ubuntu:

    sudo apt-get install nginx git python3 python3-pip
    sudo pip3 install virtualenv

## Nginx Virtual Host config

* see nginx.template.conf
* replace SITENAME with, eg, staging.my-domain.com

## Upstart Job

* see gunicorn-upstart.template.conf
* replace SITENAME with, eg, staging.my-domain.com

## Folder structure:
Assume we have a user account at /home/username

/home/username
└── sites
    └── SITENAME
         ├── database
         ├── source
         ├── static
         └── virtualenv
```

We can do a commit for those:

```
$ git add deploy_tools
$ git status # see three new files
$ git commit -m "Notes and template config files for provisioning"
```

Our source tree will now look something like this:

```
$ tree -I __pycache__
.
├── deploy_tools
│   ├── gunicorn-upstart.template.conf
│   ├── nginx.template.conf
│   └── provisioning_notes.md
├── functional_tests
│   ├── __init__.py
│   ├── [...]
├── lists
│   ├── __init__.py
│   ├── models.py
│   ├── static
│   │   ├── base.css
│   │   ├── [...]
│   ├── templates
│   │   ├── base.html
│   │   ├── [...]
├── manage.py
├── requirements.txt
└── superlists
    ├── [...]
```

Test-Driving Server Configuration and Deployment

Tests take some of the uncertainty out of deployment
: As developers, server administration is always "fun", by which I mean, a process full of uncertainty and surprises. My aim during this chapter was to show a functional test suite can take some of the uncertainty out of the process.

Typical pain points—database, static files, dependencies, custom settings
: The things that you need to keep an eye out on any deployment include your database configuration, static files, software dependencies, and custom settings that differ between development and production. You'll need to think through each of these for your own deployments.

Tests allow us to experiment
: Whenever we make a change to our server configuration, we can rerun the test suite, and be confident that everything works as well as it did before. It allows us to experiment with our setup with less fear.

"Saving Your Progress"

Being able to run our FTs against a staging server can be very reassuring. But, in most cases, you don't want to run your FTs against your "real" server. In order to "save our work", and reassure ourselves that the production server will work just as well as the real server, we need to make our deployment process repeatable.

Automation is the answer, and it's the topic of the next chapter.

CHAPTER 9
Automating Deployment with Fabric

Automate, automate, automate.

— Cay Horstman

Automating deployment is critical for our staging tests to mean anything. By making sure the deployment procedure is repeatable, we give ourselves assurances that everything will go well when we deploy to production.

Fabric is a tool which lets you automate commands that you want to run on servers. You can install fabric systemwide—it's not part of the core functionality of our site, so it doesn't need to go into our virtualenv and *requirements.txt*. So, on your local PC:

```
$ pip2 install fabric
```

At the time of writing, Fabric had not been ported to Python 3, so we have to use the Python 2 version. Thankfully, the Fabric code is totally separate from the rest of our codebase, so it's not a problem.

Installing Fabric on Windows

Fabric depends on pycrypto, which is a package that needs compiling. Compiling on Windows is a rather fraught process; it's often quicker to try and get hold of precompiled binaries put out there by some kindly soul. In this case the excellent Michael Foord[1] has provided some Windows binaries. (Don't forget to giggle at the mention of absurd US munitions export controls.)

So the instructions, for Windows, are:

1. Author of the Mock library and maintainer of `unittest`; if the Python testing world has a rock star, it is he.

1. Download and install pycrypto from the previous URL.
2. pip install Fabric.

Another amazing source of precompiled Python packages for Windows is maintained by Christoph Gohlke.

The usual setup is to have a file called *fabfile.py*, which will contain one or more functions that can later be invoked from a command-line tool called `fab`, like this:

```
fab function_name,host=SERVER_ADDRESS
```

That will invoke the function called `function_name`, passing in a connection to the server at `SERVER_ADDRESS`. There are many other options for specifying usernames and passwords, which you can find out about using `fab --help`.

Breakdown of a Fabric Script for Our Deployment

The best way to see how it works is with an example. Here's one I made earlier, automating all the deployment steps we've been going through. The main function is called `deploy`; that's the one we'll invoke from the command line. It uses several helper functions. `env.host` will contain the server address that we've passed in:

deploy_tools/fabfile.py.

```
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 = '/home/%s/sites/%s' % (env.user, 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)
    _update_virtualenv(source_folder)
    _update_static_files(source_folder)
    _update_database(source_folder)
```

❶ You'll want to update the `REPO_URL` variable with the URL of your own Git repo on its code sharing site.

❷ `env.host` will contain the address of the server we've specified at the command line, eg, *superlists.ottg.eu*.

❸ `env.user` will contain the username you're using to log in to the server.

Hopefully each of those helper functions have fairly self-descriptive names. Because any function in a fabfile can theoretically be invoked from the command line, I've used the convention of a leading underscore to indicate that they're not meant to be part of the "public API" of the fabfile. Here they are in chronological order.

Here's how we build our directory structure, in a way that doesn't fall down if it already exists:

deploy_tools/fabfile.py.

```
def _create_directory_structure_if_necessary(site_folder):
    for subfolder in ('database', 'static', 'virtualenv', 'source'):
        run('mkdir -p %s/%s' % (site_folder, subfolder))  #❶❷
```

❶ `run` is the most common Fabric command. It says "run this shell command on the server".

❷ `mkdir -p` is a useful flavor of `mkdir`, which is better in two ways: it can create directories several levels deep, and it only creates them if necessary. So, `mkdir -p /tmp/foo/bar` will create the directory *bar* but also its parent directory *foo* if it needs to. It also won't complain if *bar* already exists.[2]

Next we want to pull down our source code:

deploy_tools/fabfile.py.

```
def _get_latest_source(source_folder):
    if exists(source_folder + '/.git'):  #❶
        run('cd %s && git fetch' % (source_folder,))  #❷❸
    else:
        run('git clone %s %s' % (REPO_URL, source_folder))  #❹
    current_commit = local("git log -n 1 --format=%H", capture=True)  #❺
    run('cd %s && git reset --hard %s' % (source_folder, current_commit))  #❻
```

❶ `exists` checks whether a directory or file already exists on the server. We look for the *.git* hidden folder to check whether the repo has already been cloned in that folder.

❷ Many commands start with a `cd` in order to set the current working directory. Fabric doesn't have any state, so it doesn't remember what directory you're in from one `run` to the next.[3]

❸ `git fetch` inside an existing repository pulls down all the latest commits from the Web.

❹ Alternatively we use `git clone` with the repo URL to bring down a fresh source tree.

2. If you're wondering why we're building up paths manually with `%s` instead of the `os.path.join` command we saw earlier, it's because `path.join` will use backslashes if you run the script from Windows, but we definitely want forward slashes on the server
3. There is a Fabric "cd" command, but I figured it was one thing too many to add in this chapter.

❺ Fabric's `local` command runs a command on your local machine—it's just a wrapper around `subprocess.Popen` really, but it's quite convenient. Here we capture the output from that `git log` invocation to get the hash of the current commit that's in your local tree. That means the server will end up with whatever code is currently checked out on your machine (as long as you've pushed it up to the server).

❻ We `reset --hard` to that commit, which will blow away any current changes in the server's code directory.

For this script to work, you need to have done a `git push` of your current local commit, so that the server can pull it down and `reset` to it. If you see an error saying `Could not parse object`, try doing a `git push`.

Next we update our settings file, to set the `ALLOWED_HOSTS` and `DEBUG`, and to create a new secret key:

deploy_tools/fabfile.py.

```
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 =.+$',
        'ALLOWED_HOSTS = ["%s"]' % (site_name,)  #❷
    )
    secret_key_file = source_folder + '/superlists/secret_key.py'
    if not exists(secret_key_file):  #❸
        chars = 'abcdefghijklmnopqrstuvwxyz0123456789!@#$%^&*(-_=+)'
        key = ''.join(random.SystemRandom().choice(chars) for _ in range(50))
        append(secret_key_file, "SECRET_KEY = '%s'" % (key,))
    append(settings_path, '\nfrom .secret_key import SECRET_KEY')  #❹❺
```

❶ The Fabric `sed` command does a string substitution in a file; here it's changing DEBUG from `True` to `False`.

❷ And here it is adjusting `ALLOWED_HOSTS`, using a regex to match the right line.

❸ Django uses `SECRET_KEY` for some of its crypto—cookies and CSRF protection. It's good practice to make sure the secret key on the server is different from the one in your (possibly public) source code repo. This code will generate a new key to import into settings, if there isn't one there already (once you have a secret key, it should stay the same between deploys). Find out more in the Django docs.

❹ `append` just adds a line to the end of a file. (It's clever enough not to bother if the line is already there, but not clever enough to automatically add a newline if the file doesn't end in one. Hence the back-n.)

❺ I'm using a *relative import* (`from .secret key` instead of `from secret_key`) to be absolutely sure we're importing the local module, rather than one from somewhere else on `sys.path`. I'll talk a bit more about relative imports in the next chapter.

Other people, such as the eminent authors of the excellent Two Scoops of Django, suggest using environment variables to set things like secret keys; you should use whatever you feel is most secure in your environment.

Next we create or update the virtualenv:

deploy_tools/fabfile.py.

```
def _update_virtualenv(source_folder):
    virtualenv_folder = source_folder + '/../virtualenv'
    if not exists(virtualenv_folder + '/bin/pip'): #❶
        run('virtualenv --python=python3 %s' % (virtualenv_folder,))
    run('%s/bin/pip install -r %s/requirements.txt' % ( #❷
            virtualenv_folder, source_folder
    ))
```

❶ We look inside the virtualenv folder for the `pip` executable as a way of checking whether it already exists.

❷ Then we use `pip install -r` like we did earlier.

Updating static files is a single command:

deploy_tools/fabfile.py.

```
def _update_static_files(source_folder):
    run('cd %s && ../virtualenv/bin/python3 manage.py collectstatic --noinput' % ( # ❶
        source_folder,
    ))
```

❶ We use the virtualenv binaries folder whenever we need to run a Django *manage.py* command, to make sure we get the virtualenv version of Django, not the system one.

Finally, we update the database with `manage.py migrate`:

deploy_tools/fabfile.py.

```
def _update_database(source_folder):
    run('cd %s && ../virtualenv/bin/python3 manage.py migrate --noinput' % (
        source_folder,
    ))
```

Trying It Out

We can try this command out on our existing staging site—the script should work for an existing site as well as for a new one. If you like words with Latin roots, you might describe it as idempotent, which means it does nothing if run twice…

```
$ 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.
```

Awesome. I love making computers spew out pages and pages of output like that (in fact I find it hard to stop myself from making little '70s computer *<brrp, brrrp, brrrp>* noises like Mother in *Alien*). If we look through it we can see it is doing our bidding: the `mkdir -p` commands go through happily, even though the directories already exist. Next `git pull` pulls down the couple of commits we just made. The `sed` and `echo >>`

modify our *settings.py*. Then `pip3 install -r requirements.txt`, completes happily, noting that the existing virtualenv already has all the packages we need. `collectstatic` also notices that the static files are all already there, and finally the `migrate` completes without a hitch.

Fabric Configuration

If you are using an SSH key to log in, are storing it in the default location, and are using the same username on the server as locally, then Fabric should "just work". If you aren't there are several tweaks you may need to apply in order to get the `fab` command to do your bidding. They revolve around the username, the location of the SSH key to use, or the password.

You can pass these in to Fabric at the command line. Check out:

```
$ fab --help
```

Or see the Fabric documentation for more info.

Deploying to Live

So, let's try using it for our live site!

```
$ 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 = '\\''mqu(ffwid5vleol%ke^jil*x1mkj-4
```

```
[superlists.ottg.eu] run: echo 'from .secret_key import SECRET_KEY' >> "$(echo /
[superlists.ottg.eu] run: virtualenv --python=python3 /home/elspeth/sites/superl
[superlists.ottg.eu] out: Already using interpreter /usr/bin/python3
[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==1.7 (from -r /home/elspe
[superlists.ottg.eu] out:   Downloading Django-1.7.tar.gz (8.0MB):
[...]
[superlists.ottg.eu] out:   Downloading Django-1.7.tar.gz (8.0MB): 100%  8.0MB
[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:     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. You can see the script follows a slightly different path, doing a `git clone` to bring down a brand new repo instead of a `git pull`. It also needs to set up a new virtualenv from scratch, including a fresh install of pip and Django. The `collect static` actually creates new files this time, and the `migrate` seems to have worked too.

Nginx and Gunicorn Config Using sed

What else do we need to do to get our live site into production? We refer to our provisioning notes, which tell us to use the template files to create our Nginx virtual host and the Upstart script. How about a little Unix command-line magic?

```
elspeth@server:$ sed "s/SITENAME/superlists.ottg.eu/g" \
    deploy_tools/nginx.template.conf | sudo tee \
    /etc/nginx/sites-available/superlists.ottg.eu
```

`sed` ("stream editor") takes a stream of text and performs edits on it. It's no accident that the fabric string substitution command has the same name. In this case we ask it to substitute the string *SITENAME* for the address of our site, with the `s/replaceme/withthis/g` syntax. We pipe (`|`) the output of that to a root-user process (`sudo`), which uses `tee` to write what's piped to it to a file, in this case the Nginx sites-available virtualhost config file.

We can now activate that file:

```
elspeth@server:$ sudo ln -s ../sites-available/superlists.ottg.eu \
    /etc/nginx/sites-enabled/superlists.ottg.eu
```

Then we write the upstart script:

```
elspeth@server: sed "s/SITENAME/superlists.ottg.eu/g" \
    deploy_tools/gunicorn-upstart.template.conf | sudo tee \
    /etc/init/gunicorn-superlists.ottg.eu.conf
```

Finally we start both services:

```
elspeth@server:$ sudo service nginx reload
elspeth@server:$ sudo start gunicorn-superlists.ottg.eu
```

And we take a look at our site. It works, hooray!

Let's add the fabfile to our repo:

```
$ git add deploy_tools/fabfile.py
$ git commit -m "Add a fabfile for automated deploys"
```

Git Tag the Release

One final bit of admin. In order to preserve a historical marker, we'll use Git tags to mark the state of the codebase that reflects what's currently live on the server:

```
$ git tag LIVE
$ export TAG=`date +DEPLOYED-%F/%H%M`  # this generates a timestamp
$ echo $TAG # should show "DEPLOYED-" and then the timestamp
$ git tag $TAG
$ git push origin LIVE $TAG # pushes the tags up
```

Now it's easy, at any time, to check what the difference is between our current codebase and what's live on the servers. This will come in useful in a few chapters, when we look at database migrations. Have a look at the tag in the history:

```
$ git log --graph --oneline --decorate
```

Anyway, you now have a live website! Tell all your friends! Tell your mum, if no one else is interested! And, in the next chapter, it's back to coding again.

Further Reading

There's no such thing as the One True Way in deployment, and I'm no grizzled expert in any case. I've tried to set you off on a reasonably sane path, but there's plenty of things you could do differently, and lots, lots more to learn besides. Here are some resources I used for inspiration:

- Solid Python Deployments for Everybody by Hynek Schlawack
- Git-based fabric deployments are awesome by Dan Bravender
- The deployment chapter of Two Scoops of Django by Dan Greenfield and Audrey Roy
- The 12-factor App by the Heroku team

For some ideas on how you might go about automating the provisioning step, and an alternative to Fabric called Ansible, go check out Appendix C.

Automated Deployments

Fabric
: Fabric lets you run commands on servers from inside Python scripts. This is a great tool for automating server admin tasks.

Idempotency
: If your deployment script is deploying to existing servers, you need to design them so that they work against a fresh installation *and* against a server that's already configured.

Keep config files under source control
: Make sure your only copy of a config file isn't on the server! They are critical to your application, and should be under version control like anything else.

Automating provisioning
: Ultimately, *everything* should be automated, and that includes spinning up brand new servers and ensuring they have all the right software installed. This will involve interacting with the API of your hosting provider.

Configuration management tools
: Fabric is very flexible, but its logic is still based on scripting. More advanced tools take a more "declarative" approach, and can make your life even easier. Ansible and Vagrant are two worth checking out (see Appendix C), but there are many more (Chef, Puppet, Salt, Juju…).

CHAPTER 10
Input Validation and Test Organisation

Over the next few chapters we'll talk about testing and implementing validation of user inputs. We'll also take the opportunity to do a little tidying up—both in our application code, and also in our tests.

Validation FT: Preventing Blank Items

As our first few users start using the site, we've noticed they sometimes make mistakes that mess up their lists, like accidentally submitting blank list items, or accidentally inputting two identical items to a list. Computers are meant to help stop us from making silly mistakes, so let's see if we can get our site to help.

Here's the outline of an FT:

functional_tests/tests.py (ch10l001).

```
    def test_cannot_add_empty_list_items(self):
        # Edith goes to the home page and accidentally tries to submit
        # an empty list item. She hits Enter on the empty input box

        # The home page refreshes, and there is an error message saying
        # that list items cannot be blank

        # She tries again with some text for the item, which now works

        # Perversely, she now decides to submit a second blank list item

        # She receives a similar warning on the list page

        # And she can correct it by filling some text in
        self.fail('write me!')
```

That's all very well, but before we go any further—our functional tests file is beginning to get a little crowded. Let's split it out into several files, in which each has a single test method.

Remember that functional tests are closely linked to "user stories". If you were using some sort of project management tool like an issue tracker, you might make it so that each file matched one issue or ticket, and its filename contained the ticket ID. Or, if you prefer to think about things in terms of "features", where one feature may have several user stories, then you might have one file and class for the feature, and several methods for each of its user stories.

We'll also have one base test class which they can all inherit from. Here's how to get there step by step.

Skipping a Test

It's always nice, when doing refactoring, to have a fully passing test suite. We've just written a test with a deliberate failure. Let's temporarily switch it off, using a decorator called "skip" from `unittest`:

functional_tests/tests.py (ch10l001-1).

```
from unittest import skip
[...]

    @skip
    def test_cannot_add_empty_list_items(self):
```

This tells the test runner to ignore this test. You can see it works—if we rerun the tests, it'll say it passes:

```
$ python3 manage.py test functional_tests
[...]
Ran 3 tests in 11.577s
OK
```

Skips are dangerous—you need to remember to remove them before you commit your changes back to the repo. This is why line-by-line reviews of each of your diffs are a good idea!

Don't Forget the "Refactor" in "Red, Green, Refactor"

A criticism that's sometimes levelled at TDD is that it leads to badly architected code, as the developer just focuses on getting tests to pass rather than stopping to think about how the whole system should be designed. I think it's slightly unfair.

TDD is no silver bullet. You still have to spend time thinking about good design. But what often happens is that people forget the "Refactor" in "Red, Green, Refactor". The methodology allows you to throw together any old code to get your tests to pass, but it *also* asks you to then spend some time refactoring it to improve its design.

Often, however, the best ideas for how to refactor code don't occur to you straight away. They may occur to you days, weeks, even months after you wrote a piece of code, when you're working on something totally unrelated and you happen to see some old code again with fresh eyes. But if you're halfway through something else, should you stop to refactor the old code?

The answer is that it depends. In the case at the beginning of the chapter, we haven't even started writing our new code. We know we are in a working state, so we can justify putting a skip on our new FT (to get back to fully passing tests) and do a bit of refactoring straight away.

Later in the chapter we'll spot other bits of code we want to alter. In those cases, rather than taking the risk of refactoring an application that's not in a working state, we'll make a note of the thing we want to change on our scratchpad and wait until we're back to a fully passing test suite before refactoring.

Splitting Functional Tests out into Many Files

We start putting each test into its own class, still in the same file:

functional_tests/tests.py (ch10l002).

```
class FunctionalTest(StaticLiveServerCase):

    @classmethod
    def setUpClass(cls):
        [...]
    @classmethod
    def tearDownClass(cls):
        [...]
    def setUp(self):
        [...]
    def tearDown(self):
        [...]
    def check_for_row_in_list_table(self, row_text):
        [...]


class NewVisitorTest(FunctionalTest):

    def test_can_start_a_list_and_retrieve_it_later(self):
        [...]



class LayoutAndStylingTest(FunctionalTest):

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

```
class ItemValidationTest(FunctionalTest):

    @skip
    def test_cannot_add_empty_list_items(self):
        [...]
```

At this point we can rerun the FTs and see they all still work:

```
Ran 3 tests in 11.577s

OK
```

That's labouring it a little bit, and we could probably get away doing this stuff in fewer steps, but, as I keep saying, practising the step-by-step method on the easy cases makes it that much easier when we have a complex case.

Now we switch from a single tests file to using one for each class, and one "base" file to contain the base class all the tests will inherit from. We'll make four copies of *tests.py*, naming them appropriately, and then delete the parts we don't need from each:

```
$ 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 can be cut down to just the `FunctionalTest` class. We leave the helper method on the base class, because we suspect we're about to reuse it in our new FT:

functional_tests/base.py (ch10l003).

```
from django.contrib.staticfiles.testing import StaticLiveServerCase
from selenium import webdriver
import sys

class FunctionalTest(StaticLiveServerCase):

    @classmethod
    def setUpClass(cls):
        [...]
    def tearDownClass(cls):
        [...]
    def setUp(self):
        [...]
    def tearDown(self):
        [...]
    def check_for_row_in_list_table(self, row_text):
        [...]
```

Keeping helper methods in a base `FunctionalTest` class is one useful way of preventing duplication in FTs. Later in the book (in Chapter 21) we'll use the "Page pattern", which is related, but prefers composition over inheritance.

Our first FT is now in its own file, and should be just one class and one test method:

functional_tests/test_simple_list_creation.py (ch10l004).

```
from .base import FunctionalTest
from selenium import webdriver
from selenium.webdriver.common.keys import Keys

class NewVisitorTest(FunctionalTest):

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

I used a relative import (`from .base`). Some people like to use them a lot in Django code (eg, your views might import models using `from .models import List`, instead of `from list.models`). Ultimately this is a matter of personal preference. I prefer to use relative imports only when I'm super-super sure that the relative position of the thing I'm importing won't change. That applies in this case because I know for sure all the tests will sit next to *base.py*, which they inherit from.

The layout and styling FT should now be one file and one class:

functional_tests/test_layout_and_styling.py (ch10l005).

```
from .base import FunctionalTest

class LayoutAndStylingTest(FunctionalTest):
        [...]
```

Lastly our new validation test is in a file of its own too:

functional_tests/test_list_item_validation.py (ch10l006).

```
from unittest import skip
from .base import FunctionalTest

class ItemValidationTest(FunctionalTest):

    @skip
    def test_cannot_add_empty_list_items(self):
        [...]
```

And we can test everything worked by rerunning `manage.py test functional_tests`, and checking once again that all three tests are run:

```
Ran 3 tests in 11.577s

OK
```

Now we can remove our skip:

functional_tests/test_list_item_validation.py (ch10l007).

```
class ItemValidationTest(FunctionalTest):

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

Running a Single Test File

As a side bonus, we're now able to run an individual test file, like this:

```
$ python3 manage.py test functional_tests.test_list_item_validation
[...]
AssertionError: write me!
```

Brilliant, no need to sit around waiting for all the FTs when we're only interested in a single one. Although we need to remember to run all of them now and again, to check for regressions. Later in the book we'll see how to give that task over to an automated Continuous Integration loop. For now let's commit!

```
$ git status
$ git add functional_tests
$ git commit -m"Moved Fts into their own individual files"
```

Fleshing Out the FT

Now let's start implementing the test, or at least the beginning of it:

functional_tests/test_list_item_validation.py (ch10l008).

```
def test_cannot_add_empty_list_items(self):
    # Edith goes to the home page and accidentally tries to submit
    # an empty list item. She hits Enter on the empty input box
    self.browser.get(self.server_url)
    self.browser.find_element_by_id('id_new_item').send_keys('\n')

    # The home page refreshes, and there is an error message saying
    # that list items cannot be blank
    error = self.browser.find_element_by_css_selector('.has-error') #❶
    self.assertEqual(error.text, "You can't have an empty list item")

    # She tries again with some text for the item, which now works
    self.browser.find_element_by_id('id_new_item').send_keys('Buy milk\n')
    self.check_for_row_in_list_table('1: Buy milk') #❷

    # Perversely, she now decides to submit a second blank list item
    self.browser.find_element_by_id('id_new_item').send_keys('\n')

    # She receives a similar warning on the list page
    self.check_for_row_in_list_table('1: Buy milk')
    error = self.browser.find_element_by_css_selector('.has-error')
    self.assertEqual(error.text, "You can't have an empty list item")

    # And she can correct it by filling some text in
    self.browser.find_element_by_id('id_new_item').send_keys('Make tea\n')
    self.check_for_row_in_list_table('1: Buy milk')
    self.check_for_row_in_list_table('2: Make tea')
```

A couple of things to note about this test:

❶ We specify we're going to use a CSS class from Bootstrap called `.has-error` to mark our error text. We'll see that Bootstrap has some useful styling for those

❷ As predicted, we are reusing the `check_for_row_in_list_table` helper function when we want to confirm that list item submission *does* work.

The technique of keeping helper methods in a parent class is meant to prevent duplication across your functional test code. The day we decide to change the implementation of how our list table works, we want to make sure we only have to change our FT code in one place, not in dozens of places across loads of FTs…

And we're off!

```
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"css selector","selector":".has-error"}' ; Stacktrace:
```

I'll let you do your own "first-cut FT" commit.

Using Model-Layer Validation

There are two levels at which you can do validation in Django. One is at the model level, and the other is higher up at the forms level. I like to use the lower level whenever possible, partially because I'm a bit too fond of databases and database integrity rules, and partially because it's safer—you can sometimes forget which form you use to validate input, but you're always going to use the same database.

Refactoring Unit Tests into Several Files

We're going to want to add another test for our model, but before we do so, it's time to tidy up our unit tests in a similar way to the functional tests. A difference will be that, because the `lists` app contains real application code as well as tests, we'll separate out the tests into their own folder:

```
$ mkdir lists/tests
$ touch lists/tests/__init__.py
$ git mv lists/tests.py lists/tests/test_all.py
$ git status
$ git add lists/tests
$ python3 manage.py test lists
[...]
Ran 10 tests in 0.034s

OK
$ git commit -m"Move unit tests into a folder with single file"
```

If you get a message saying "Ran 0 tests", you probably forgot to add the dunderinit—it needs to be there or else the tests folder isn't a valid Python module…[1]

Now we turn *test_all.py* into two files, one called *test_views.py*, which only contains view tests, and one called *test_models.py*:

```
$ git mv lists/tests/test_all.py lists/tests/test_views.py
$ cp lists/tests/test_views.py lists/tests/test_models.py
```

We strip *test_models.py* down to being just the one test—it means it needs far fewer imports:

lists/tests/test_models.py (ch10l009).

```
from django.test import TestCase
from lists.models import Item, List


class ListAndItemModelsTest(TestCase):
        [...]
```

Whereas *test_views.py* just loses one class:

lists/tests/test_views.py (ch10l010).

```
--- 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):
[...]
```

We rerun the tests to check everything is still there:

```
$ python3 manage.py test lists
[...]
Ran 10 tests in 0.040s

OK
```

Great!

```
$ git add lists/tests
$ git commit -m "Split out unit tests into two files"
```

1. "Dunder" is shorthand for double-underscore, so "dunderinit" means *__init__.py*.

Some people like to make their unit tests into a tests folder straight away, as soon as they start a project, with the addition of another file, *test_forms.py*. That's a perfectly good idea; I just thought I'd wait until it became necessary, to avoid doing too much housekeeping all in the first chapter!

Unit Testing Model Validation and the self.assertRaises Context Manager

Let's add a new test method to `ListAndItemModelsTest`, which tries to create a blank list item:

lists/tests/test_models.py (ch10l012-1).

```
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()
```

If you're new to Python, you may never have seen the `with` statement. It's used with what are called "context managers", which wrap a block of code, usually with some kind of setup, cleanup, or error-handling code. There's a good write-up in the Python 2.5 release notes.

This is a new unit testing technique: when we want to check that doing something will raise an error, we can use the `self.assertRaises` context manager. We could have used something like this instead:

```
try:
    item.save()
    self.fail('The save should have raised an exception')
except ValidationError:
    pass
```

But the `with` formulation is neater. Now, we can try running the test, and see it fail:

```
    item.save()
AssertionError: ValidationError not raised
```

A Django Quirk: Model Save Doesn't Run Validation

And now we discover one of Django's little quirks. *This test should already pass*. If you take a look at the docs for the Django model fields, you'll see that `TextField` actually defaults to `blank=False`, which means that it *should* disallow empty values.

So why is the test not failing? Well, for slightly counterintuitive historical reasons, Django models don't run full validation on save. As we'll see later, any constraints that are actually implemented in the database will raise errors on save, but SQLite doesn't support enforcing emptiness constraints on text columns, and so our save method is letting this invalid value through silently.

There's a way of checking whether the constraint will happen at the database level or not: if it was at the database level, we would need a migration to apply the constraint. But, Django knows that SQLite doesn't support this type of constraint, so if we try and run `makemigrations`, it will report there's nothing to do:

```
$ python3 manage.py makemigrations
No changes detected
```

Django does have a method to manually run full validation however, called `full_clean`. Let's hack it in to see it work:

lists/tests/test_models.py.

```
        with self.assertRaises(ValidationError):
            item.save()
            item.full_clean()
```

That gets the test to pass:

```
OK
```

That taught us a little about Django validation, and the test is there to warn us if we ever forget our requirement and set `blank=True` on the `text` field (try it!).

Surfacing Model Validation Errors in the View

Let's try and enforce our model validation in the views layer and bring it up through into our templates, so the user can see them. Here's how we can optionally display an error in our HTML—we check whether the template has been passed an error variable, and if so, we display it next to the form:

lists/templates/base.html (ch10l013).

```
<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>
```

Take a look at the Bootstrap docs for more info on form controls.

Passing this error to the template is the job of the view function. Let's take a look at the unit tests in the `NewListTest` class. I'm going to use two slightly different error-handling patterns here.

In the first case, our URL and view for new lists will optionally render the same template as the home page, but with the addition of an error message. Here's a unit test for that:

lists/tests/test_views.py (ch10l014).

```
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)
```

As we're writing this test, we might get slightly offended by the */lists/new* URL, which we're manually entering as a string. We've got a lot of URLs hardcoded in our tests, in our views, and in our templates, which violates the DRY principle. I don't mind a bit of duplication in tests, but we should definitely be on the lookout for hardcoded URLs in our views and templates, and make a note to refactor them out. But we won't do them straight away, because right now our application is in a broken state. We want to get back to a working state first.

Back to our test, which is failing because the view is currently returning a 302 redirect, rather than a "normal" 200 response:

```
AssertionError: 302 != 200
```

Let's try calling `full_clean()` in the 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('/lists/%d/' % (list_.id,))
```

As we're looking at the view code, we find a good candidate for a hardcoded URL to get rid of. Let's add that to our scratchpad:

Now the model validation raises an exception, which comes up through our view:

```
[...]
  File "/workspace/superlists/lists/views.py", line 11, in new_list
    item.full_clean()
[...]
django.core.exceptions.ValidationError: {'text': ['This field cannot be
blank.']}
```

So we try our first approach: using a `try/except` to detect errors. Obeying the Testing Goat, we start with just the `try/except` and nothing else. The tests should tell us what to code next…

lists/views.py (ch10l015).

```
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('/lists/%d/' % (list_.id,))
```

That gets us back to the 302 != 200:

```
AssertionError: 302 != 200
```

Let's return a rendered template then, which should take care of the template check as well:

lists/views.py (ch10l016).

```
    except ValidationError:
        return render(request, 'home.html')
```

And the tests now tell us to put the error message into the template:

```
AssertionError: False is not true : Couldn't find 'You can't have an empty list
item' in response
```

We do that by passing a new template variable in:

lists/views.py (ch10l017).

```
    except ValidationError:
        error = "You can't have an empty list item"
        return render(request, 'home.html', {"error": error})
```

Hmm, it looks like that didn't quite work:

```
AssertionError: False is not true : Couldn't find 'You can't have an empty list
item' in response
```

A little print-based debug…

lists/tests/test_views.py.

```
expected_error = "You can't have an empty list item"
print(response.content.decode())
self.assertContains(response, expected_error)
```

…will show us the cause: Django has HTML-escaped the apostrophe:

```
[...]
<span class="help-block">You can&#39;t have an
empty list item</span>
```

We could hack something like this into our test:

```
    expected_error = "You can&#39;t have an empty list item"
```

But using Django's helper function is probably a better idea:

lists/tests/test_views.py (ch10l019).

```
from django.utils.html import escape
[...]

        expected_error = escape("You can't have an empty list item")
        self.assertContains(response, expected_error)
```

That passes!

```
Ran 12 tests in 0.047s

OK
```

Checking Invalid Input Isn't Saved to the Database

Before we go further though, did you notice a little logic error we've allowed to creep into our implementation? We're currently creating an object, even if validation fails:

lists/views.py.

```
    item = Item.objects.create(text=request.POST['item_text'], list=list_)
    try:
        item.full_clean()
    except ValidationError:
        [...]
```

Let's add a new unit test to make sure that empty list items don't get saved:

lists/tests/test_views.py (ch10l020-1).

```
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)
```

That gives:

```
[...]
Traceback (most recent call last):
  File "/workspace/superlists/lists/tests/test_views.py", line 57, in
test_invalid_list_items_arent_saved
    self.assertEqual(List.objects.count(), 0)
AssertionError: 1 != 0
```

We fix it like this:

lists/views.py (ch10l020-2).

```
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('/lists/%d/' % (list_.id,))
```

Do the FTs pass?

```
$ python3 manage.py test functional_tests.test_list_item_validation
[...]
  File "/workspace/superlists/functional_tests/test_list_item_validation.py",
line 26, in test_cannot_add_empty_list_items
[...]
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"css selector","selector":".has-error"}' ; Stacktrace:
```

Not quite, but they did get a little further. Checking the `line 26`, we can see that we've got past the first part of the test, and are now onto the second check—that submitting a second empty item also shows an error.

We've got some working code though, so let's have a commit:

```
$ git commit -am"Adjust new list view to do model validation"
```

Django Pattern: Processing POST Requests in the Same View as Renders the Form

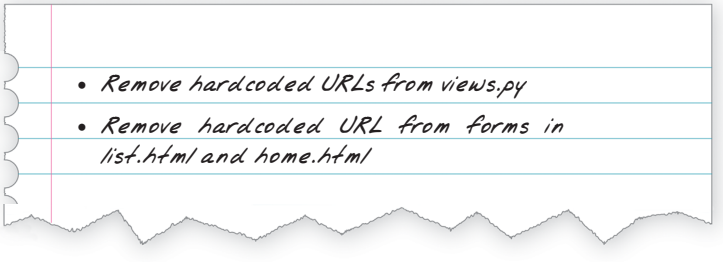
This time we'll use a slightly different approach, one that's actually a very common pattern in Django, which is to use the same view to process POST requests as to render the form that they come from. Whilst this doesn't fit the REST-ful URL model quite as well, it has the important advantage that the same URL can display a form, and display any errors encountered in processing the user's input.

The current situation is that we have one view and URL for displaying a list, and one view and URL for processing additions to that list. We're going to combine them into one. So, in *list.html*, our form will have a different target:

lists/templates/list.html (ch10l020).

```
{% block form_action %}/lists/{{ list.id }}/{% endblock %}
```

Incidentally, that's another hardcoded URL. Let's add it to our to-do list, and while we're thinking about it, there's one in *home.html* too:



This will immediately break our original functional test, because the `view_list` page doesn't know how to process POST requests yet:

```
$ python3 manage.py test functional_tests
[...]
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"css selector","selector":".has-error"}' ; Stacktrace:
[...]
AssertionError: '2: Use peacock feathers to make a fly' not found in ['1: Buy
peacock feathers']
```

In this section we're performing a refactor at the application level. We execute our application-level refactor by changing or adding unit tests, and then adjusting our code. We use the functional tests to tell us when our refactor is complete and things are back to working as before. Have another look at the diagram from the end of Chapter 4 if you need to get your bearings.

Refactor: Transferring the new_item Functionality into view_list

Let's take all the old tests from `NewItemTest`, the ones that are about saving POST requests to existing lists, and move them into `ListViewTest`. As we do so, we also make them point at the base list URL, instead of *…/new_item*:

lists/tests/test_views.py (ch10l021).

```
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(
            '/lists/%d/' % (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(
            '/lists/%d/' % (correct_list.id,),
            data={'item_text': 'A new item for an existing list'}
        )
        self.assertRedirects(response, '/lists/%d/' % (correct_list.id,))
```

Note that the `NewItemTest` class disappears completely. I've also changed the name of the redirect test to make it explicit that it only applies to POST requests.

That gives:

```
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
```

We change the `view_list` function to handle two types of request:

lists/views.py (ch10l022-1).

```
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('/lists/%d/' % (list_.id,))
    return render(request, 'list.html', {'list': list_})
```

That gets us passing tests:

```
Ran 13 tests in 0.047s

OK
```

Now we can delete the `add_item` view, since it's no longer needed … oops, a couple of unexpected failures:

```
[...]
django.core.exceptions.ViewDoesNotExist: Could not import lists.views.add_item.
View does not exist in module lists.views.
[...]
FAILED (errors=4)
```

It's because we've deleted the view, but it's still being referred to in *urls.py*. We remove it from there:

lists/urls.py (ch10l023).

```
urlpatterns = patterns('',
    url(r'^(\d+)/$', 'lists.views.view_list', name='view_list'),
    url(r'^new$', 'lists.views.new_list', name='new_list'),
)
```

And that gets us to the `OK`. Let's try a full FT run:

```
$ python3 manage.py test functional_tests
[...]

Ran 3 tests in 15.276s

FAILED (errors=1)
```

We're back to the one failure in our new functional test. Our refactor of the `add_item` functionality is complete. We should commit there:

```
$ git commit -am"Refactor list view to handle new item POSTs"
```

So did I break the rule about never refactoring against failing tests? In this case, it's allowed, because the refactor is required to get our new functionality to work. You should definitely never refactor against failing *unit* tests. But in my book it's OK for the FT for the current story you're working on to be failing. If you prefer a clean test run, you could add a skip to the current FT.

Enforcing Model Validation in view_list

We still want the addition of items to existing lists to be subject to our model validation rules. Let's write a new unit test for that; it's very similar to the one for the home page, with just a couple of tweaks:

lists/tests/test_views.py (ch10l024).

```
class ListViewTest(TestCase):
    [...]

    def test_validation_errors_end_up_on_lists_page(self):
        list_ = List.objects.create()
        response = self.client.post(
            '/lists/%d/' % (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)
```

That should fail, because our view currently does not do any validation, and just redirects for all POSTs:

```
    self.assertEqual(response.status_code, 200)
AssertionError: 302 != 200
```

Here's an implementation:

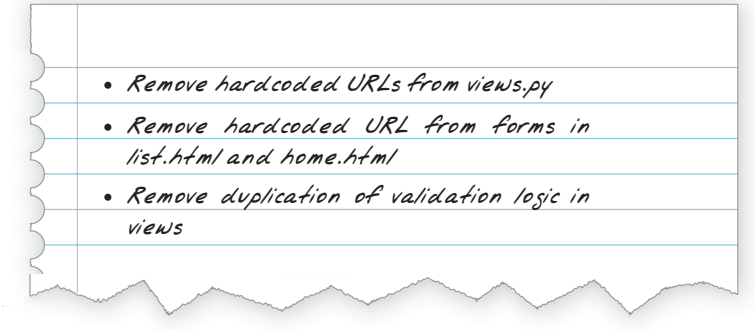
lists/views.py (ch10l025).

```
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('/lists/%d/' % (list_.id,))
        except ValidationError:
            error = "You can't have an empty list item"

    return render(request, 'list.html', {'list': list_, 'error': error})
```

It's not deeply satisfying is it? There's definitely some duplication of code here, that `try/except` occurs twice in *views.py*, and in general things are feeling clunky.

```
Ran 14 tests in 0.047s

OK
```

Let's wait a bit before we do more refactoring though, because we know we're about to do some slightly different validation coding for duplicate items. We'll just add it to our scratchpad for now:



One of the reasons that the "three strikes and refactor" rule exists is that, if you wait until you have three use cases, each might be slightly different, and it gives you a better view for what the common functionality is. If you refactor too early, you may find that the third use case doesn't quite fit with your refactored code…

At least our functional tests are back to passing:

```
$ python3 manage.py test functional_tests
[...]
OK
```

We're back to a working state, so we can take a look at some of the items on our scratchpad. This would be a good time for a commit. And possibly a tea break.

```
$ git commit -am"enforce model validation in list view"
```

Refactor: Removing Hardcoded URLs

Do you remember those `name=` parameters in *urls.py*? We just copied them across from the default example Django gave us, and I've been giving them some reasonably descriptive names. Now we find out what they're for.

lists/urls.py.

```
    url(r'^(\d+)/$', 'lists.views.view_list', name='view_list'),
    url(r'^new$', 'lists.views.new_list', name='new_list'),
```

The {% url %} Template Tag

We can replace the hardcoded URL in *home.html* with a Django template tag which refers to the URL's "name":

lists/templates/home.html (ch10l026-1).

```
{% block form_action %}{% url 'new_list' %}{% endblock %}
```

We check that doesn't break the unit tests:

```
$ python3 manage.py test lists
OK
```

Let's do the other template. This one is more interesting, because we pass it a parameter:

lists/templates/list.html (ch10l026-2).

```
{% block form_action %}{% url 'view_list' list.id %}{% endblock %}
```

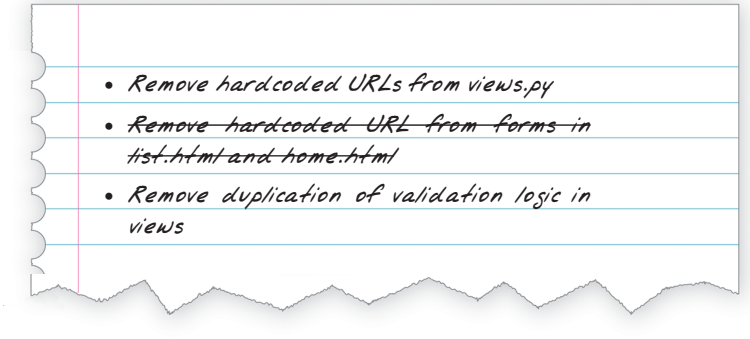
Check out the Django docs on reverse URL resolution for more info.

We run the tests again, and check they all pass:

```
$ python3 manage.py test lists
OK
$ python3 manage.py test functional_tests
OK
```

Excellent:

```
$ git commit -am"Refactor hard-coded URLs out of templates"
```



Using get_absolute_url for Redirects

Now let's tackle *views.py*. One way of doing it is just like in the template, passing in the name of the URL and a positional argument:

lists/views.py (ch10l026-3).

```
def new_list(request):
    [...]
    return redirect('view_list', list_.id)
```

That would get the unit and functional tests passing, but the `redirect` function can do even better magic than that! In Django, because model objects are often associated with a particular URL, you can define a special function called `get_absolute_url` which says what page displays the item. It's useful in this case, but it's also useful in the Django admin (which I don't cover in the book, but you'll soon discover for yourself): it will let you jump from looking at an object in the admin view to looking at the object on the live site. I'd always recommend defining a `get_absolute_url` for a model whenever there is one that makes sense; it takes no time at all.

All it takes is a super-simple unit test in *test_models.py*:

lists/tests/test_models.py (ch10l026-4).

```
    def test_get_absolute_url(self):
        list_ = List.objects.create()
        self.assertEqual(list_.get_absolute_url(), '/lists/%d/' % (list_.id,))
```

Which gives:

```
AttributeError: 'List' object has no attribute 'get_absolute_url'
```

And the implementation is to use Django's `reverse` function, which essentially does the reverse of what Django normally does with *urls.py* (see docs):

lists/models.py (ch10l026-5).

```
from django.core.urlresolvers import reverse


class List(models.Model):

    def get_absolute_url(self):
        return reverse('view_list', args=[self.id])
```

And now we can use it in the view—the `redirect` function just takes the object we want to redirect to, and it uses `get_absolute_url` under the hood automagically!

lists/views.py (ch10l026-6).

```
def new_list(request):
    [...]
    return redirect(list_)
```

There's more info in the Django docs. Quick check that the unit tests still pass:

```
OK
```

Then we do the same to `view_list`:

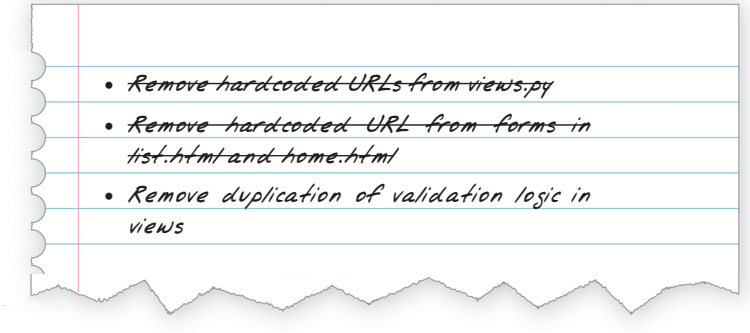
lists/views.py (ch10l026-7).

```
def view_list(request, list_id):
    [...]

            item.save()
            return redirect(list_)
        except ValidationError:
            error = "You can't have an empty list item"
```

And a full unit test and functional test run to assure ourselves that everything still works:

```
$ python3 manage.py test lists
OK
$ python3 manage.py test functional_tests
OK
```

Cross off our to-dos:



Let's do a commit:

```
$ git commit -am"Use get_absolute_url on List model to DRY urls in views"
```

That final to-do item will be the subject of the next chapter…

Tips on Organising Tests and Refactoring

Use a tests folder

Just as you use multiple files to hold your application code, you should split your tests out into multiple files.

- Use a folder called *tests*, adding a *__init__.py* which imports all test classes.
- For functional tests, group them into tests for a particular feature or user story.
- For unit tests, you want a separate test file for each tested source code file. For Django, that's typically *test_models.py*, *test_views.py*, and *test_forms.py*.
- Have at least a placeholder test for *every* function and class.

Don't forget the "Refactor" in "Red, Green, Refactor"
: The whole point of having tests is to allow you to refactor your code! Use them, and make your code as clean as you can.

Don't refactor against failing tests
: - In general!
  - But the FT you're currently working on doesn't count.
  - You can occasionally put a skip on a test which is testing something you haven't written yet.
  - More commonly, make a note of the refactor you want to do, finish what you're working on, and do the refactor a little later, when you're back to a working state.
  - Don't forget to remove any skips before you commit your code! You should always review your diffs line by line to catch things like this.

CHAPTER 11

A Simple Form

At the end of the last chapter, we were left with the thought that there was too much duplication of code in the validation handling bits of our views. Django encourages you to use form classes to do the work of validating user input, and choosing what error messages to display. Let's see how that works.

As we go through the chapter, we'll also spend a bit of time tidying up our unit tests, and making sure each of them only tests one thing at a time.

Moving Validation Logic into a Form

In Django, a complex view is a code smell. Could some of that logic be pushed out to a form? Or to some custom methods on the model class? Or maybe even to a non-Django module that represents your business logic?

Forms have several superpowers in Django:

- They can process user input and validate it for errors.
- They can be used in templates to render HTML input elements, and error messages too.
- And, as we'll see later, some of them can even save data to the database for you.

You don't have to use all three form superpowers in every form. You may prefer to roll your own HTML, or do your own saving. But they are an excellent place to keep validation logic.

Exploring the Forms API with a Unit Test

Let's do a little experimenting with forms by using a unit test. My plan is to iterate towards a complete solution, and hopefully introduce forms gradually enough that they'll make sense if you've never seen them before.

First we add a new file for our form unit tests, and we start with a test that just looks at the form HTML:

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()` renders the form as HTML. This unit test is using a `self.fail` for some exploratory coding. You could just as easily use a `manage.py shell` session, although you'd need to keep reloading your code for each change.

Let's make a minimal form. It inherits from the base `Form` class, and has a single field called `item_text`:

lists/forms.py.

```
from django import forms

class ItemForm(forms.Form):
    item_text = forms.CharField()
```

We now see a failure message which tells us what the auto-generated form HTML will look like:

```
        self.fail(form.as_p())
AssertionError: <p><label for="id_item_text">Item text:</label> <input
id="id_item_text" name="item_text" type="text" /></p>
```

It's already pretty close to what we have in *base.html*. We're missing the placeholder attribute and the Bootstrap CSS classes. Let's make our unit test into a test for that:

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())
```

That gives us a fail which justifies some real coding. How can we customise the input for a form field? Using a "widget". Here it is with just the placeholder:

lists/forms.py.

```
class ItemForm(forms.Form):
    item_text = forms.CharField(
        widget=forms.fields.TextInput(attrs={
            'placeholder': 'Enter a to-do item',
        }),
    )
```

That gives:

```
AssertionError: 'class="form-control input-lg"' not found in '<p><label
for="id_item_text">Item text:</label> <input id="id_item_text" name="item_text"
placeholder="Enter a to-do item" type="text" /></p>'
```

And then:

lists/forms.py.

```
        widget=forms.fields.TextInput(attrs={
            'placeholder': 'Enter a to-do item',
            'class': 'form-control input-lg',
        }),
```

Doing this sort of widget customisation would get tedious if we had a much larger, more complex form. Check out django-crispy-forms and django-floppyforms for some help.

Development-Driven Tests: Using Unit Tests for Exploratory Coding

Does this feel a bit like development-driven tests? That's OK, now and again.

When you're exploring a new API, you're absolutely allowed to mess about with it for a while before you get back to rigorous TDD. You might use the interactive console, or write some exploratory code (but you have to promise the Testing Goat that you'll throw it away and rewrite it properly later).

Here we're actually using a unit test as a way of experimenting with the forms API. It's actually a pretty good way of learning how it works.

Switching to a Django ModelForm

What's next? We want our form to reuse the validation code that we've already defined on our model. Django provides a special class which can auto-generate a form for a model, called `ModelForm`. As you'll see, it's configured using a special attribute called `Meta`:

lists/forms.py.

```
from django import forms
```

```
from lists.models import Item

class ItemForm(forms.models.ModelForm):

    class Meta:
        model = Item
        fields = ('text',)
```

In `Meta` we specify which model the form is for, and which fields we want it to use.

`ModelForms` do all sorts of smart stuff, like assigning sensible HTML form input types to different types of field, and applying default validation. Check out the docs for more info.

We now have some different-looking form HTML:

```
AssertionError: 'placeholder="Enter a to-do item"' not found in '<p><label
for="id_text">Text:</label> <textarea cols="40" id="id_text" name="text"
rows="10">\r\n</textarea></p>'
```

It's lost our placeholder and CSS class. But you can also see that it's using `name="text"` instead of `name="item_text"`. We can probably live with that. But it's using a `textarea` instead of a normal input, and that's not the UI we want for our app. Thankfully, you can override widgets for `ModelForm` fields, similarly to the way we did it with the normal form:

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',
            }),
        }
```

That gets the test passing.

Testing and Customising Form Validation

Now let's see if the `ModelForm` has picked up the same validation rules which we defined on the model. We'll also learn how to pass data into the form, as if it came from the user:

lists/tests/test_forms.py (ch11l008).

```
    def test_form_validation_for_blank_items(self):
        form = ItemForm(data={'text': ''})
        form.save()
```

That gives us:

```
ValueError: The Item could not be created because the data didn't validate.
```

Good, the form won't allow you to save if you give it an empty item text.

Now let's see if we can get it to use the specific error message that we want. The API for checking form validation *before* we try and save any data is a function called `is_valid`:

lists/tests/test_forms.py (ch11l009).

```
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"]
    )
```

Calling `form.is_valid()` returns `True` or `False`, but it also has the side effect of validating the input data, and populating the `errors` attribute. It's a dictionary mapping the names of fields to lists of errors for those fields (it's possible for a field to have more than one error).

That gives us:

```
AssertionError: ['This field is required.'] != ["You can't have an empty list
item"]
```

Django already has a default error message that we could present to the user—you might use it if you were in a hurry to build your web app, but we care enough to make our message special. Customising it means changing `error_messages`, another `Meta` variable:

lists/forms.py (ch11l010).

```
    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
```

You know what would be even better than messing about with all these error strings? Having a constant:

lists/forms.py (ch11l011).

```
EMPTY_LIST_ERROR = "You can't have an empty list item"
[...]

        error_messages = {
```

```
            'text': {'required': EMPTY_LIST_ERROR}
        }
```

Rerun the tests to see they pass … OK. Now we change the test:

lists/tests/test_forms.py (ch11l012).

```
from lists.forms import EMPTY_LIST_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_LIST_ERROR])
```

And the tests still pass:

```
OK
```

Great. Totes committable:

```
$ git status # should show lists/forms.py and tests/test_forms.py
$ git add lists
$ git commit -m "new form for list items"
```

Using the Form in Our Views

I had originally thought to extend this form to capture uniqueness validation as well as empty-item validation. But there's a sort of corollary to the "deploy as early as possible" lean methodology, which is "merge code as early as possible". In other words: while building this bit of forms code, it would be easy to go on for ages, adding more and more functionality to the form—I should know, because that's exactly what I did during the drafting of this chapter, and I ended up doing all sorts of work making an all-singing, all-dancing form class before I realised it wouldn't really work for our most basic use case.

So, instead, try and use your new bit of code as soon as possible. This makes sure you never have unused bits of code lying around, and that you start checking your code against "the real world" as soon as possible.

We have a form class which can render some HTML and do validation of at least one kind of error—let's start using it! We should be able to use it in our *base.html* template, and so in all of our views.

Using the Form in a View with a GET Request

Let's start in our unit tests for the home view. We'll replace the old-style `test_home_page_returns_correct_html` and `test_root_url_resolves_to_home_page_view` with a set of tests that use the Django test client. We leave the old tests in at first, to check that our new tests are equivalent:

lists/tests/test_views.py (ch11l013).

```
from lists.forms import ItemForm

class HomePageTest(TestCase):

    def test_root_url_resolves_to_home_page_view(self):
        [...]

    def test_home_page_returns_correct_html(self):
        request = HttpRequest()
        [...]


    def test_home_page_renders_home_template(self):
        response = self.client.get('/')
        self.assertTemplateUsed(response, 'home.html') #❶


    def test_home_page_uses_item_form(self):
        response = self.client.get('/')
        self.assertIsInstance(response.context['form'], ItemForm) #❷
```

❶ We'll use the helper method `assertTemplateUsed` to replace our old manual test of the template.

❷ We use `assertIsInstance` to check that our view uses the right kind of form.

That gives us:

```
KeyError: 'form'
```

So we use the form in our home page view:

lists/views.py (ch11l014).

```
[...]
from lists.forms import ItemForm
from lists.models import Item, List

def home_page(request):
    return render(request, 'home.html', {'form': ItemForm()})
```

OK, now let's try using it in the template—we replace the old `<input ..>` with `{{ 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 }}` renders just the HTML input for the `text` field of the form.

Now the old test is out of date:

```
    self.assertEqual(response.content.decode(), expected_html)
AssertionError: '<!DO[596 chars]     <input class="form-control input-lg"
id="[342 chars]l>\n' != '<!DO[596 chars]     \n                    \n
[233 chars]l>\n'
```

That error message is impossible to read though. Let's clarify its message a little:

lists/tests/test_views.py (ch11l016).

```
class HomePageTest(TestCase):
    maxDiff = None #❶
    [...]
    def test_home_page_returns_correct_html(self):
        request = HttpRequest()
        response = home_page(request)
        expected_html = render_to_string('home.html')
        self.assertMultiLineEqual(response.content.decode(), expected_html) #❷
```

❷ `assertMultiLineEqual` is useful for comparing long strings; it gives you a diff-style output, but it truncates long diffs by default…

❶ …so that's why we also need to set `maxDiff = None` on the test class.

Sure enough, it's because our `render_to_string` call doesn't know about the form:

```
[...]
                    <form method="POST" action="/lists/new">
-                       <input class="form-control input-lg" id="id_text"
name="text" placeholder="Enter a to-do item" type="text" />
+
[...]
```

But we can fix that:

lists/tests/test_views.py.

```
def test_home_page_returns_correct_html(self):
    request = HttpRequest()
    response = home_page(request)
    expected_html = render_to_string('home.html', {'form': ItemForm()})
    self.assertMultiLineEqual(response.content.decode(), expected_html)
```

And that gets us back to passing. We've now reassured ourselves enough that the behaviour has stayed the same, so it's now OK to delete the two old tests. The `assertTemplateUsed` and `response.context` checks from the new test are sufficient for testing a basic view with a GET request.

That leaves us with just two tests in `HomePageTest`:

lists/tests/test_views.py (ch11l017).

```
class HomePageTest(TestCase):

    def test_home_page_renders_home_template(self):
        [...]

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

A Big Find and Replace

One thing we have done, though, is changed our form—it no longer uses the same `id` and `name` attributes. You'll see if we run our functional tests that they fail the first time they try and find the input box:

```
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"id","selector":"id_new_item"}' ; Stacktrace:
```

We'll need to fix this, and it's going to involve a big find and replace. Before we do that, let's do a commit, to keep the rename separate from the logic change:

```
$ git diff # review changes in home.html, views.py and its tests
$ git commit -am "use new form in home_page, simplify tests. NB breaks stuff"
```

Let's fix the functional tests. A quick `grep` shows us there are several places where we're using `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('\n')
[...]
```

That's a good call for a refactor. Let's make a new helper method in *base.py*:

functional_tests/base.py (ch11l018).

```
class FunctionalTest(StaticLiveServerCase):
    [...]
    def get_item_input_box(self):
        return self.browser.find_element_by_id('id_text')
```

And then we use it throughout—I had to make three changes in *test_simple_list_creation.py*, two in *test_layout_and_styling.py*, and four in *test_list_item_validation.py*, eg:

functional_tests/test_simple_list_creation.py.

```
        # She is invited to enter a to-do item straight away
        inputbox = self.get_item_input_box()
```

Or:

functional_tests/test_list_item_validation.py.

```
        # an empty list item. She hits Enter on the empty input box
        self.browser.get(self.server_url)
        self.get_item_input_box().send_keys('\n')
```

I won't show you every single one, I'm sure you can manage this for yourself! You can redo the `grep` to check you've caught them all.

We're past the first step, but now we have to bring the rest of the application code in line with the change. We need to find any occurrences of the old `id` (`id_new_item`) and `name` (`item_text`) and replace them too, with `id_text` and `text`, respectively:

```
$ grep -r id_new_item lists/
lists/static/base.css:#id_new_item {
```

That's one change, and similarly for the `name`:

```
$ grep -Ir 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'],
lists/tests/test_views.py:            data={'item_text': 'A new list item'}
lists/tests/test_views.py:            data={'item_text': 'A new list item'}
lists/tests/test_views.py:        response = self.client.post('/lists/new',
data={'item_text': ''})
[...]
```

Once we're done, we rerun the unit tests to check everything still works:

```
$ python3 manage.py test lists
Creating test database for alias 'default'...
.................
 ----------------------------------------------------------------------
Ran 17 tests in 0.126s

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

And the functional tests too:

```
$ python3 manage.py test functional_tests
[...]
  File "/workspace/superlists/functional_tests/test_simple_list_creation.py",
line 40, in test_can_start_a_list_and_retrieve_it_later
    return self.browser.find_element_by_id('id_text')
  File "/workspace/superlists/functional_tests/base.py", line 31, in
get_item_input_box
    return self.browser.find_element_by_id('id_text')
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"id","selector":"id_text"}' ; Stacktrace:
[...]
FAILED (errors=3)
```

Not quite! Let's look at where this is happening—if you check the line number from one of the failures, you'll see that each time after we've submitted a first item, the input box has disappeared from the lists page.

Checking *views.py* and the `new_list` view we can see it's because if we detect a validation error, we're not actually passing the form to the *home.html* template:

lists/views.py.

```
    except ValidationError:
        error = "You can't have an empty list item"
        return render(request, 'home.html', {"error": error})
```

We'll want to use the form in this view too. Before we make any more changes though, let's do a commit:

```
$ git status
$ git commit -am"rename all item input ids and names. still broken"
```

Using the Form in a View That Takes POST Requests

Now we want to adjust the unit tests for the `new_list` view, especially the one that deals with validation. Let's take a look at it now:

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)
```

Adapting the Unit Tests for the new_list View

For a start this test is testing too many things at once, so we've got an opportunity to clarify things here. We should split out two different assertions:

- If there's a validation error, we should render the home template, with a 200.
- If there's a validation error, the response should contain our error text.

And we can add a new one too:

- If there's a validation error, we should pass our form object to the template.

And while we're at it, we'll use a constant instead of a hardcoded string for that error message:

lists/tests/test_views.py (ch11l023).

```
from lists.forms import ItemForm, EMPTY_LIST_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_LIST_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)
```

Much better. Each test is now clearly testing one thing, and, with a bit of luck, just one will fail and tell us what to do:

```
$ python3 manage.py test lists
[...]
======================================================================
ERROR: test_for_invalid_input_passes_form_to_template
(lists.tests.test_views.NewListTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/workspace/superlists/lists/tests/test_views.py", line 55, 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)
```

Using the Form in the View

And here's how we use the form in the 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 We pass the `request.POST` data into the form's constructor.

❷ We use `form.is_valid()` to determine whether this is a good or a bad submission.

❸ In the invalid case, we pass the form down to the template, instead of our hardcoded error string.

That view is now looking much nicer! And all our tests pass, except one:

```
    self.assertContains(response, escape(EMPTY_LIST_ERROR))
[...]
AssertionError: False is not true : Couldn't find 'You can&#39;t have an empty
list item' in response
```

Using the Form to Display Errors in the Template

We're failing because we're not yet using the form to display errors in the template:

lists/templates/base.html (ch11l026).

```
<form method="POST" action="{% block form_action %}{% endblock %}">
    {{ form.text }}
    {% csrf_token %}
    {% if form.errors %}❶
        <div class="form-group has-error">
            <div class="help-block">{{ form.text.errors }}</div>❷
        </div>
    {% endif %}
</form>
```

❶ `form.errors` contains a list of all the errors for the form.

❷ `form.text.errors` is a list of just the errors for the `text` field.

What does that do to our tests?

```
FAIL: test_validation_errors_end_up_on_lists_page
(lists.tests.test_views.ListViewTest)
[...]
AssertionError: False is not true : Couldn't find 'You can&#39;t have an empty
list item' in response
```

An unexpected failure—it's actually in the tests for our final view, `view_list`. Because we've changed the way errors are displayed in *all* templates, we're no longer showing the error that we manually pass into the template.

That means we're going to need to rework `view_list` as well, before we can get back to a working state.

Using the Form in the Other View

This view handles both GET and POST requests. Let's start with checking the form is used in GET requests. We can have a new test for that:

lists/tests/test_views.py.

```
class ListViewTest(TestCase):
    [...]

    def test_displays_item_form(self):
        list_ = List.objects.create()
        response = self.client.get('/lists/%d/' % (list_.id,))
        self.assertIsInstance(response.context['form'], ItemForm)
        self.assertContains(response, 'name="text"')
```

That gives:

```
KeyError: 'form'
```

Here's a minimal implementation:

lists/views.py (ch11l028).

```
def view_list(request, list_id):
    [...]
    form = ItemForm()
    return render(request, 'list.html', {
        'list': list_, "form": form, "error": error
    })
```

A Helper Method for Several Short Tests

Next we want to use the form errors in the second view. We'll split our current single test for the invalid case (`test_validation_errors_end_up_on_lists_page`) into several separate ones:

lists/tests/test_views.py (ch11l030).

```
class ListViewTest(TestCase):
    [...]

    def post_invalid_input(self):
        list_ = List.objects.create()
        return self.client.post(
            '/lists/%d/' % (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_LIST_ERROR))
```

By making a little helper function, `post_invalid_input`, we can make four separate tests without duplicating lots of lines of code.

We've seen this several times now. It often feels more natural to write view tests as a single, monolithic block of assertions—the view should do this and this and this then return that with this. But breaking things out into multiple tests is definitely worthwhile; as we saw in previous chapters, it helps you isolate the exact problem you may have, when you later come and change your code and accidentally introduce a bug. Helper methods are one of the tools that lower the psychological barrier.

For example, now we can see there's just one failure, and it's a clear one:

```
FAIL: test_for_invalid_input_shows_error_on_page
(lists.tests.test_views.ListViewTest)
AssertionError: False is not true : Couldn't find 'You can&#39;t have an empty
list item' in response
```

Now let's see if we can properly rewrite the view to use our form. Here's a first cut:

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})
```

That gets the unit tests passing:

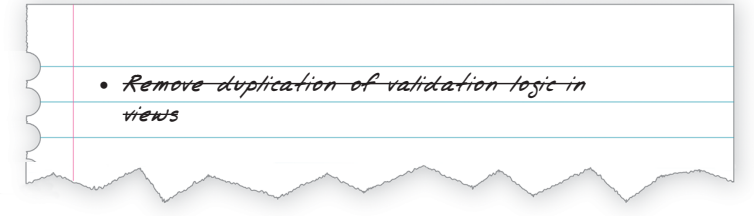
```
Ran 23 tests in 0.086s

OK
```

How about the FTs?

```
$ python3 manage.py test functional_tests
Creating test database for alias 'default'...
...
 ----------------------------------------------------------------------
Ran 3 tests in 12.154s

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

Woohoo! Can you feel that feeling of relief wash over you? We've just made a major change to our small app—that input field, with its name and ID, is absolutely critical to making everything work. We've touched seven or eight different files, doing a refactor

that's quite involved … this is the kind of thing that, without tests, would seriously worry me. In fact, I might well have decided that it wasn't worth messing with code that works … but, because we have a full tests suite, we can delve around in it, tidying things up, safe in the knowledge that the tests are there to spot any mistakes we make. It just makes it that much likelier that you're going to keep refactoring, keep tidying up, keep gardening, keep tending your code, keep everything neat and tidy and clean and smooth and precise and concise and functional and good.



Definitely time for a commit:

```
$ git diff
$ git commit -am"use form in all views, back to working state"
```

Using the Form's Own Save Method

There are a couple more things we can do to make our views even simpler. I've mentioned that forms are supposed to be able to save data to the database for us. Our case won't quite work out of the box, because the item needs to know what list to save to, but it's not hard to fix that.

We start, as always, with a test. Just to illustrate what the problem is, let's see what happens if we just try to call `form.save()`:

lists/tests/test_forms.py (ch11l032).

```
    def test_form_save_handles_saving_to_a_list(self):
        form = ItemForm(data={'text': 'do me'})
        new_item = form.save()
```

Django isn't happy, because an item needs to belong to a list:

```
django.db.utils.IntegrityError: NOT NULL constraint failed: lists_item.list_id
```

Our solution is to tell the form's save method what list it should save to:

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_)
```

We then make sure that the item is correctly saved to the database, with the right attributes:

```
TypeError: save() got an unexpected keyword argument 'for_list'
```

And here's how we can implement our custom save method:

lists/forms.py (ch11l034).

```
    def save(self, for_list):
        self.instance.list = for_list
        return super().save()
```

The `.instance` attribute on a form represents the database object that is being modified or created. And I only learned that as I was writing this chapter! There are other ways of getting this to work, including manually creating the object yourself, or using the `commit=False` argument to save, but this is the neatest I think. We'll explore a different way of making a form "know" what list it's for in the next chapter:

```
Ran 24 tests in 0.086s

OK
```

Finally we can refactor our views. `new_list` first:

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})
```

Rerun the test to check everything still passes:

```
Ran 24 tests in 0.086s

OK
```

And now `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})
```

And we still have full passes:

```
Ran 24 tests in 0.111s

OK
```

and

```
Ran 3 tests in 14.367s

OK
```

Great! Our two views are now looking very much like "normal" Django views: they take information from a user's request, combine it with some custom logic or information from the URL (`list_id`), pass it to a form for validation and possible saving, and then redirect or render a template.

Forms and validation are really important in Django, and in web programming in general, so let's see if we can't make a slightly more complicated one in the next chapter.

Tips

Thin views
: If you find yourself looking at complex views, and having to write a lot of tests for them, it's time to start thinking about whether that logic could be moved elsewhere: possibly to a form, like we've done here. Another possible place would be a custom method on the model class. And—once the complexity of the app demands it—out of Django-specific files and into your own classes and functions, that capture your core business logic.

Each test should test one thing
: The heuristic is to be suspicious if there's more than one assertion in a test. Sometimes two assertions are closely related, so they belong together. But often your first draft of a test ends up testing multiple behaviours, and it's worth rewriting it as several tests. Helper functions can keep them from getting too bloated.

CHAPTER 12

More Advanced Forms

Now let's look at some more advanced forms usage. We've helped our users to avoid blank list items, now let's help them avoid duplicate items.

This chapter goes into more intricate details of Django's form validation, and you can consider it optional if you already know all about customising Django forms. If you're still learning Django, there's good stuff in here. If you want to skip ahead, that's OK too. Make sure you take a quick look at the aside on developer stupidity, and the recap on testing views at the end.

Another FT for Duplicate Items

We add a second test method to `ItemValidationTest`:

functional_tests/test_list_item_validation.py (ch12l001).

```
    def test_cannot_add_duplicate_items(self):
        # Edith goes to the home page and starts a new list
        self.browser.get(self.server_url)
        self.get_item_input_box().send_keys('Buy wellies\n')
        self.check_for_row_in_list_table('1: Buy wellies')

        # She accidentally tries to enter a duplicate item
        self.get_item_input_box().send_keys('Buy wellies\n')

        # She sees a helpful error message
        self.check_for_row_in_list_table('1: Buy wellies')
        error = self.browser.find_element_by_css_selector('.has-error')
        self.assertEqual(error.text, "You've already got this in your list")
```

Why have two test methods rather than extending one, or having a new file and class? It's a judgement call. These two feel closely related; they're both about validation on the same input field, so it feels right to keep them in the same file. On the other hand, they're logically separate enough that it's practical to keep them in different methods:

```
$ python3 manage.py test functional_tests.test_list_item_validation
[...]
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"css selector","selector":".has-error"}' ; Stacktrace:

Ran 2 tests in 9.613s
```

OK, so we know the first of the two tests passes now. Is there a way to run just the failing one, I hear you ask? Why yes indeed:

```
$ python3 manage.py test functional_tests.\
test_list_item_validation.ItemValidationTest.test_cannot_add_duplicate_items
[...]
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"css selector","selector":".has-error"}' ; Stacktrace:
```

Preventing Duplicates at the Model Layer

Here's what we really wanted to do. It's a new test that checks that duplicate items in the same list raise an error:

lists/tests/test_models.py (ch09l028).

```
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()
```

And, while it occurs to us, we add another test to make sure we don't overdo it on our integrity constraints:

lists/tests/test_models.py (ch09l029).

```
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()  # should not raise
```

I always like to put a little comment for tests which are checking that a particular use case should *not* raise an error; otherwise it can be hard to see what's being tested.

```
AssertionError: ValidationError not raised
```

If we want to get it deliberately wrong, we can do this:

lists/models.py (ch09l030).

```
class Item(models.Model):
    text = models.TextField(default='', unique=True)
    list = models.ForeignKey(List, default=None)
```

That lets us check that our second test really does pick up on this problem:

```
Traceback (most recent call last):
  File "/workspace/superlists/lists/tests/test_models.py", line 62, in
```

```
test_CAN_save_same_item_to_different_lists
    item.full_clean()  # should not raise
    [...]
django.core.exceptions.ValidationError: {'text': ['Item with this Text already
exists.']}
```

An Aside on When to Test for Developer Stupidity

One of the judgement calls in testing is when you should write tests that sound like "check we haven't done something stupid". In general, you should be wary of these.

In this case, we've written a test to check that you can't save duplicate items to the same list. Now, the simplest way to get that test to pass, the way in which you'd write the least lines of code, would be to make it impossible to save *any* duplicate items. That justifies writing another test, despite the fact that it would be a "stupid" or "wrong" thing for us to code.

But you can't be writing tests for every possible way we could have coded something wrong. If you have a function that adds two numbers, you can write a couple of tests:

```
assert adder(1, 1) == 2
assert adder(2, 1) == 3
```

But you have the right to assume that the implementation isn't deliberately screwey or perverse:

```
def adder(a, b):
    # unlikely code!
    if a == 3:
        return 666
    else:
        return a + b
```

One way of putting it is that you should trust yourself not to do something *deliberately* stupid, but not something *accidentally* stupid.

Just like `ModelForms`, models have a `class Meta`, and that's where we can implement a constraint which says that that an item must be unique for a particular list, or in other words, that `text` and `list` must be unique together:

lists/models.py (ch09l031).

```
class Item(models.Model):
    text = models.TextField(default='')
    list = models.ForeignKey(List, default=None)

    class Meta:
        unique_together = ('list', 'text')
```

You might want to take a quick peek at the Django docs on model `Meta` attributes at this point.

A Little Digression on Queryset Ordering and String Representations

When we run the tests they reveal an unexpected failure:

```
======================================================================
FAIL: test_saving_and_retrieving_items
(lists.tests.test_models.ListAndItemModelsTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/workspace/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
[...]
```

Depending on your platform and its SQLite installation, you may not see this error. You can follow along anyway; the code and tests are interesting in their own right.

That's a bit of a puzzler. A bit of print-based debugging:

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')
```

Will show us…

```
.....Item the second
The first (ever) list item
F.....
```

It looks like our uniqueness constraint has messed with the default ordering of queries like `Item.objects.all()`. Although we already have a failing test, it's best to add a new test that explicitly tests for ordering:

lists/tests/test_models.py (ch09l032).

```
    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]
        )
```

That gives us a new failure, but it's not a very readable one:

```
AssertionError: [<Item: Item object>, <Item: Item object>, <Item: Item object>]
!= [<Item: Item object>, <Item: Item object>, <Item: Item object>]
```

We need a better string representation for our objects. Let's add another unit test:

Ordinarily you would be wary of adding more failing tests when you already have some—it makes reading test output that much more complicated, and just generally makes you nervous. Will we ever get back to a working state? In this case, they're all quite simple tests, so I'm not worried.

lists/tests/test_models.py (ch12l008).

```
def test_string_representation(self):
    item = Item(text='some text')
    self.assertEqual(str(item), 'some text')
```

That gives us:

```
AssertionError: 'Item object' != 'some text'
```

As well as the other two failures. Let's start fixing them all now:

lists/models.py (ch09l034).

```
class Item(models.Model):
    [...]

    def __str__(self):
        return self.text
```

in Python 2.x versions of Django, the string representation method used to be `__unicode__`. Like much string handling, this is simplified in Python 3. See the docs.

Now we're down to two failures, and the ordering test has a more readable failure message:

```
AssertionError: [<Item: 3>, <Item: i1>, <Item: item 2>] != [<Item: i1>, <Item:
item 2>, <Item: 3>]
```

We can fix that in the `class Meta`:

lists/models.py (ch09l035).

```
    class Meta:
        ordering = ('id',)
        unique_together = ('list', 'text')
```

Does that work?

```
AssertionError: [<Item: i1>, <Item: item 2>, <Item: 3>] != [<Item: i1>, <Item:
item 2>, <Item: 3>]
```

Urp? It has worked; you can see the items *are* in the same order, but the tests are confused. I keep running into this problem actually—Django querysets don't compare well with lists. We can fix it by converting the queryset to a list[1] in our test:

lists/tests/test_models.py (ch09l036).

```
        self.assertEqual(
            list(Item.objects.all()),
            [item1, item2, item3]
        )
```

That works; we get a fully passing test suite:

```
OK
```

Rewriting the Old Model Test

That long-winded model test did serendipitously help us find an unexpected bug, but now it's time to rewrite it. I wrote it in a very verbose style to introduce the Django ORM, but in fact, now that we have the explicit test for ordering, we can get the same coverage from a couple of much shorter tests. Delete `test_saving_and_retrieving_items` and replace with this:

lists/tests/test_models.py (ch12l010).

```
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())

    [...]
```

That's more than enough really—a check of the default values of attributes on a freshly initialized model object is enough to sanity-check that we've probably set some fields up in *models.py*. The "item is related to list" test is a real "belt and braces" test to make sure that our foreign key relationship works.

While we're at it, we can split this file out into tests for `Item` and tests for `List` (there's only one of the latter, `test_get_absolute_url`:

1. You could also check out `assertSequenceEqual` from `unittest`, and `assertQuerysetEqual` from Django's test tools, although I confess when I last looked at `assertQuerysetEqual` I was quite baffled…

lists/tests/test_models.py (ch12l011).

```
class ItemModelTest(TestCase):

    def test_default_text(self):
        [...]


class ListModelTest(TestCase):

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

That's neater and tidier:

```
$ python3 manage.py test lists
[...]
Ran 29 tests in 0.092s

OK
```

Some Integrity Errors Do Show Up on Save

A final aside before we move on. Do you remember I mentioned in Chapter 10 that some data integrity errors *are* picked up on save? It all depends on whether the integrity constraint is actually being enforced by the database.

Try running `makemigrations` and you'll see that Django wants to add the `unique_together` constraint to the database itself, rather than just having it as an application-layer constraint:

```
$ python3 manage.py makemigrations
Migrations for 'lists':
  0005_auto_20140414_2038.py:
    - Alter unique_together for item (1 constraints)
```

Now if we change our duplicates test to do a `.save` instead of a `.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()
```

It gives:

```
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
```

You can see that the error bubbles up from SQLite, and it's a different error to the one we want, an `IntegrityError` instead of a `ValidationError`.

Let's revert our changes to the test, and see them all passing again:

```
$ python3 manage.py test lists
[...]
Ran 29 tests in 0.092s
OK
```

And now it's time to commit our model-layer changes:

```
$ git status # should show changes to tests + models and new migration
# let's give our new migration a better name
$ 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"
```

Experimenting with Duplicate Item Validation at the Views Layer

Let's try running our FT, just to see where we are:

```
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"id","selector":"id_list_table"}' ; Stacktrace:
```

In case you didn't see it as it flew past, the site is 500ing.[2] A quick unit test at the view level ought to clear this up:

lists/tests/test_views.py (ch12l014).

```
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(
            '/lists/%d/' % (list1.id,),
            data={'text': 'textey'}
        )
```

2. It's showing a server error, code 500. Gotta get with the jargon!

```
        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)
```

Gives:

```
django.db.utils.IntegrityError: UNIQUE constraint failed: lists_item.list_id,
lists_item.text
```

We want to avoid integrity errors! Ideally, we want the call to `is_valid` to somehow notice the duplication error before we even try to save, but to do that, our form will need to know what list it's being used for, in advance.

Let's put a skip on that test for now:

lists/tests/test_views.py (ch12l015).

```
from unittest import skip
[...]

    @skip
    def test_duplicate_item_validation_errors_end_up_on_lists_page(self):
```

A More Complex Form to Handle Uniqueness Validation

The form to create a new list only needs to know one thing, the new item text. A form which validates that list items are unique needs to know both. Just like we overrode the save method on our `ItemForm`, this time we'll override the constructor on our new form class so that it knows what list it applies to.

We duplicate our tests for the previous form, tweaking them slightly:

lists/tests/test_forms.py (ch12l016).

```
from lists.forms import (
    DUPLICATE_ITEM_ERROR, EMPTY_LIST_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_LIST_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])
```

We can iterate through a few TDD cycles (I won't show them all, but I'm sure you'll do them, right? Remember, the Goat sees all.) until we get a form with a custom constructor, which just ignores its `for_list` argument:

lists/forms.py (ch09l071).

```
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)
```

Gives:

```
ValueError: ModelForm has no model class specified.
```

Now let's see if making it inherit from our existing form helps:

lists/forms.py (ch09l072).

```
class ExistingListItemForm(ItemForm):
    def __init__(self, for_list, *args, **kwargs):
        super().__init__(*args, **kwargs)
```

That takes us down to just one failure:

```
FAIL: test_form_validation_for_duplicate_items
(lists.tests.test_forms.ExistingListItemFormTest)
    self.assertFalse(form.is_valid())
AssertionError: True is not false
```

The next step requires a little knowledge of Django's internals, but you can read up on it in the Django docs on model validation and form validation.

Django uses a method called `validate_unique`, both on forms and models, and we can use both, in conjunction with the `instance` attribute:

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)
```

That's a bit of Django voodoo right there, but we basically take the validation error, adjust its error message, and then pass it back into the form. And we're there! A quick commit:

```
$ git diff
$ git commit -a
```

Using the Existing List Item Form in the List View

Now let's see if we can put this form to work in our view.

We remove the skip, and while we're at it, we can use our new constant. Tidy.

lists/tests/test_views.py (ch12l049).

```
from lists.forms import (
    DUPLICATE_ITEM_ERROR, EMPTY_LIST_ERROR,
    ExistingListItemForm, ItemForm,
)
[...]

    def test_duplicate_item_validation_errors_end_up_on_lists_page(self):
        [...]
        expected_error = escape(DUPLICATE_ITEM_ERROR)
```

That brings back out integrity error:

```
django.db.utils.IntegrityError: UNIQUE constraint failed: lists_item.list_id,
lists_item.text
```

Our fix for this is to switch to using the new form class. Before we implement it, let's find the tests where we check the form class, and adjust them:

lists/tests/test_views.py (ch12l050).

```
class ListViewTest(TestCase):
[...]

    def test_displays_item_form(self):
        list_ = List.objects.create()
        response = self.client.get('/lists/%d/' % (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)
```

That gives us:

```
AssertionError: <lists.forms.ItemForm object at 0x7f767e4b7f90> is not an
instance of <class 'lists.forms.ExistingListItemForm'>
```

So we can adjust the view:

lists/views.py (ch12l051).

```
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()
            [...]
```

And that *almost* fixes everything, except for an unexpected fail:

```
TypeError: save() missing 1 required positional argument: 'for_list'
```

Our custom save method from the parent `ItemForm` is no longer needed. Let's make a quick unit test for that:

lists/tests/test_forms.py (ch12l053).

```
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])
```

We can make our form call the grandparent save method:

lists/forms.py (ch12l054).

```
    def save(self):
        return forms.models.ModelForm.save(self)
```

Personal opinion here: I could have used `super`, but I prefer not to use `super` when it requires arguments, eg to get a grandparent method. I find Python 3's `super()` with no args awesome to get the immediate parent. Anything else is too error-prone, and I find it ugly besides. YMMV.

And we're there! All the unit tests pass:

```
$ python3 manage.py test lists
[...]
Ran 34 tests in 0.082s

OK
```

And so does our FT for validation:

```
$ python3 manage.py test functional_tests.test_list_item_validation
Creating test database for alias 'default'...
```

```
..
 ----------------------------------------------------------------------
Ran 2 tests in 12.048s

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

As a final check, we rerun *all* the FTs:

```
$ python3 manage.py test functional_tests
Creating test database for alias 'default'...
....
 ----------------------------------------------------------------------
Ran 4 tests in 19.048s

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

Hooray! Time for a final commit, and a wrap-up of what we've learned about testing views over the last few chapters.

Recap: What to Test in Views

Partial listing showing all view tests and assertions.

```
class ListViewTest(TestCase):
    def test_uses_list_template(self):
        response = self.client.get('/lists/%d/' % (list_.id,)) #1
        self.assertTemplateUsed(response, 'list.html') #2
    def test_passes_correct_list_to_template(self):
        self.assertEqual(response.context['list'], correct_list) #3
    def test_displays_item_form(self):
        self.assertIsInstance(response.context['form'], ExistingListItemForm) #4
        self.assertContains(response, 'name="text"')
    def test_displays_only_items_for_that_list(self):
        self.assertContains(response, 'itemey 1') #5
        self.assertContains(response, 'itemey 2') #6
        self.assertNotContains(response, 'other list item 1') #7
    def test_can_save_a_POST_request_to_an_existing_list(self):
        self.assertEqual(Item.objects.count(), 1) #8
        self.assertEqual(new_item.text, 'A new item for an existing list') #9
    def test_POST_redirects_to_list_view(self):
        self.assertRedirects(response, '/lists/%d/' % (correct_list.id,)) #10
    def test_for_invalid_input_nothing_saved_to_db(self):
        self.assertEqual(Item.objects.count(), 0) #11
    def test_for_invalid_input_renders_list_template(self):
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'list.html') #12
    def test_for_invalid_input_passes_form_to_template(self):
        self.assertIsInstance(response.context['form'], ExistingListItemForm) #13
    def test_for_invalid_input_shows_error_on_page(self):
        self.assertContains(response, escape(EMPTY_LIST_ERROR)) #14
    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)
```

❶ Use the Django test client.

❷ Check the template used. Then, check each item in the template context.

❸ Check any objects are the right ones, or querysets have the correct items.

❹ Check any forms are of the correct class.

❺ ❻ ❼ Test any template logic: any `for` or `if` should get a minimal test.

❽ ❾ ❿ ⓫ ⓬ For views that handle POST requests, make sure you test both the valid case and the invalid case.

⓭ ⓮ Sanity-check that your form is rendered, and its errors are displayed.

Why these points? Skip ahead to Appendix B, and I'll show how they are sufficient to ensure that our views are still correct if we refactor them to start using class-based views.

Next we'll try and make our data validation more friendly by using a bit of client-side code. Uh-oh, you know what that means…

CHAPTER 13

Dipping Our Toes, Very Tentatively, into JavaScript

If the Good Lord had wanted us to enjoy ourselves, he wouldn't have granted us his precious gift of relentless misery.

— John Calvin (as portrayed in Calvin and the Chipmunks)

Our new validation logic is good, but wouldn't it be nice if the error messages disappeared once the user started fixing the problem? For that we'd need a teeny-tiny bit of JavaScript.

We are utterly spoiled by programming every day in such a joyful language as Python. JavaScript is our punishment. So let's dip our toes in, very gingerly.

I'm going to assume you know the basics of JavaScript syntax. If you haven't read *JavaScript: The Good Parts*, go and get yourself a copy right away! It's not a very long book.

Starting with an FT

Let's add a new functional test to the `ItemValidationTest` class:

functional_tests/test_list_item_validation.py (ch14l001).

```
def test_error_messages_are_cleared_on_input(self):
    # Edith starts a new list in a way that causes a validation error:
    self.browser.get(self.server_url)
    self.get_item_input_box().send_keys('\n')
    error = self.browser.find_element_by_css_selector('.has-error')
    self.assertTrue(error.is_displayed()) #❶

    # She starts typing in the input box to clear the error
    self.get_item_input_box().send_keys('a')
```

```
        # She is pleased to see that the error message disappears
        error = self.browser.find_element_by_css_selector('.has-error')
        self.assertFalse(error.is_displayed()) #❷
```

❶ ❷ `is_displayed()` tells you whether an element is visible or not. We can't just rely on checking whether the element is present in the DOM, because now we're starting to hide elements.

That fails appropriately, but before we move on: three strikes and refactor! We've got several places where we find the error element using CSS. Let's move it to a helper function:

functional_tests/test_list_item_validation.py (ch14l002).

```
    def get_error_element(self):
        return self.browser.find_element_by_css_selector('.has-error')
```

I like to keep helper functions in the FT class that's using them, and only promote them to the base class when they're actually needed elsewhere. It stops the base class from getting too cluttered. YAGNI.

And we then make five replacements in *test_list_item_validation*, like this one for example:

functional_tests/test_list_item_validation.py (ch14l003).

```
        # She is pleased to see that the error message disappears
        error = self.get_error_element()
        self.assertFalse(error.is_displayed())
```

We have an expected failure:

```
$ python3 manage.py test functional_tests.test_list_item_validation
[...]
    self.assertFalse(error.is_displayed())
AssertionError: True is not false
```

And we can commit this as the first cut of our FT.

Setting Up a Basic JavaScript Test Runner

Choosing your testing tools in the Python and Django world is fairly straightforward. The standard library `unittest` module is perfectly adequate, and the Django test runner also makes a good default choice. There are some alternatives out there—nose is popular, and I've personally found pytest to be very impressive. But there is a clear default option, and it's just fine.[1]

1. Admittedly once you start looking for Python BDD tools, things are a little more confusing.

Not so in the JavaScript world! We use YUI at work, but I thought I'd go out and see whether there were any new tools out there. I was overwhelmed with options—jsUnit, Qunit, Mocha, Chutzpah, Karma, Testacular, Jasmine, and many more. And it doesn't end there either: as I had almost settled on one of them, Mocha,[2] I find out that I now need to choose an *assertion framework* and a *reporter*, and maybe a *mocking library*, and it never ends!

In the end I decided we should use QUnit because it's simple, and it works well with jQuery.

Make a directory called *tests* inside *lists/static*, and download the Qunit JavaScript and CSS files into it, stripping out version numbers if necessary (I got version 1.12). We'll also put a file called *tests.html* in there:

```
$ tree lists/static/tests/
lists/static/tests/
├── qunit.css
├── qunit.js
└── tests.html
```

The boilerplate for a QUnit HTML file looks like this, including a smoke test:

lists/static/tests/tests.html.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Javascript tests</title>
    <link rel="stylesheet" href="qunit.css">
</head>

<body>
    <div id="qunit"></div>
    <div id="qunit-fixture"></div>
    <script src="qunit.js"></script>
    <script>
/*global $, test, equal */

test("smoke test", function () {
    equal(1, 1, "Maths works!");
});

    </script>

</body>
</html>
```

Dissecting that, the important things to pick up are the fact that we pull in *qunit.js* using the first `<script>` tag, and then use the second one to write the main body of tests.

2. Purely because it features the NyanCat test runner.

Are you wondering about the `/*global` comment? I'm using a tool called `jslint`, which is a syntax-checker for Javascript that's integrated into my editor. The comment tells it what global variables are expected—it's not important to the code, so don't worry about it, but I would recommend taking a look at Javascript linters like `jslint` or `jshint` when you get a moment. They can be very useful for avoiding JavaScript "gotchas".

If you open up the file using your web browser (no need to run the dev server, just find the file on disk) you should see something like Figure 13-1.

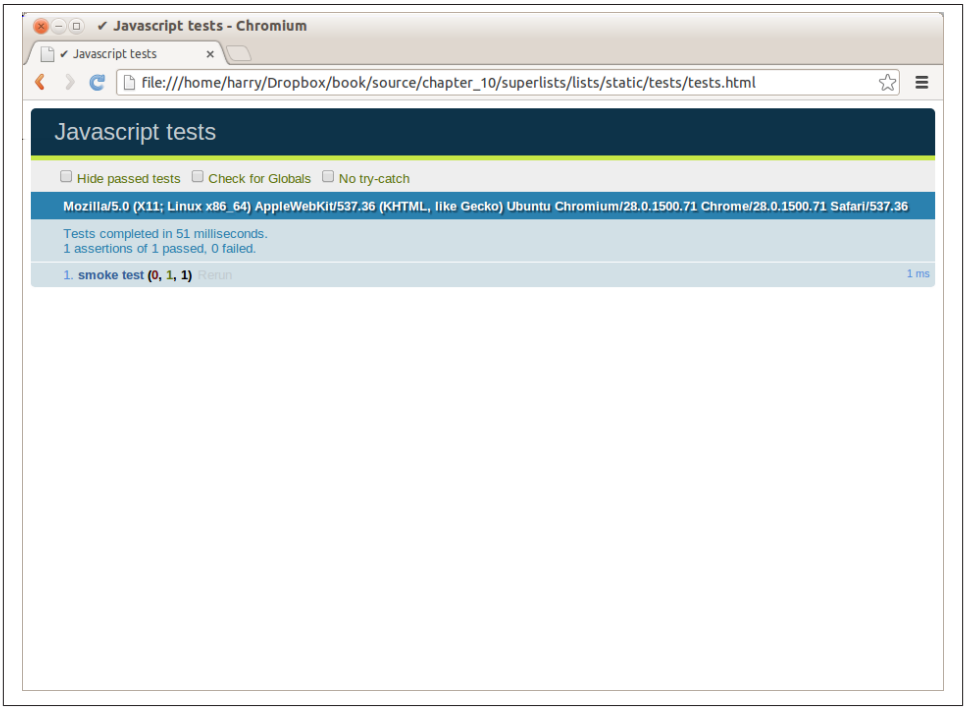


Figure 13-1. Basic QUnit screen

Looking at the test itself, we'll find many similarities with the Python tests we've been writing so far:

```
test("smoke test", function () { // ❶
    equal(1, 1, "Maths works!"); // ❷
});
```

❶ The `test` function defines a test case, a bit like `def test_something(self)` did in Python. Its first argument is a name for the test, and the second is a function for the body of the test.

❷ The `equal` function is an assertion; very much like `assertEqual`, it compares two arguments. Unlike in Python, though, the message is displayed both for failures and for passes, so it should be phrased as a positive rather than a negative.

Why not try changing those arguments to see a deliberate failure?

Using jQuery and the Fixtures Div

Let's get a bit more comfortable with what our testing framework can do, and start using a bit of jQuery

If you've never seen jQuery before, I'm going to try and explain it as we go, just enough so that you won't be totally lost; but this isn't a jQuery tutorial. You may find it helpful to spend an hour or two investigating jQuery at some point during this chapter.

Let's add jQuery to our scripts, and a few elements to use in our tests:

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="http://code.jquery.com/jquery.min.js"></script>
    <script src="qunit.js"></script>
    <script>
/*global $, test, equal */

test("smoke test", function () {
    equal($('.has-error').is(':visible'), true); //❷❸
    $('.has-error').hide(); //❹
    equal($('.has-error').is(':visible'), false); //❺
});

    </script>
```

❶ The `<form>` and its contents are there to represent what will be on the real list page.

❷ jQuery magic starts here! `$` is the jQuery Swiss Army knife. It's used to find bits of the DOM. Its first argument is a CSS selector; here, we're telling it to find all elements that have the class "error". It returns an object that represents one or more DOM elements. That, in turn, has various useful methods that allow us to manipulate or find out about those elements.

❸ One of which is `.is`, which can tell us whether an element matches a particular CSS property. Here we use `:visible` to check whether the element is displayed or hidden.

❹ We then use jQuery's `.hide()` method to hide the div. Behind the scenes, it dynamically sets a `style="display: none"` on the element.

❺ And finally we check that it's worked, with a second `equal` assertion.

If you refresh the browser, you should see that all passes:

Expected results from QUnit in the browser.

```
2 assertions of 2 passed, 0 failed.
1. smoke test (0, 2, 2)
```

Time to see how fixtures work. Let's just dupe up this test:

lists/static/tests/tests.html.

```
    <script>
/*global $, test, equal */

test("smoke test", function () {
    equal($('.has-error').is(':visible'), true);
    $('.has-error').hide();
    equal($('.has-error').is(':visible'), false);
});
test("smoke test 2", function () {
    equal($('.has-error').is(':visible'), true);
    $('.has-error').hide();
    equal($('.has-error').is(':visible'), false);
});

    </script>
```

Slightly unexpectedly, we find one of them fails—see Figure 13-2.

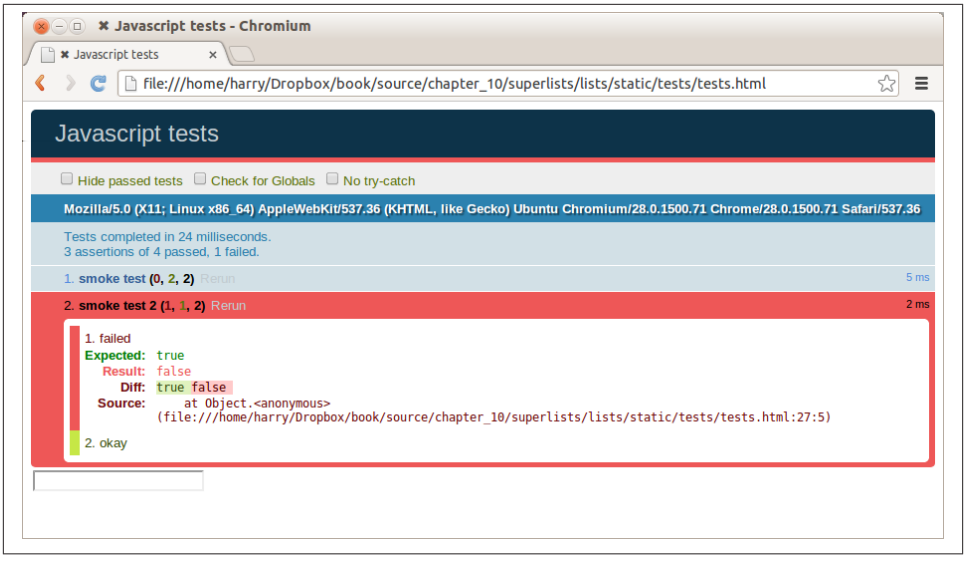


Figure 13-2. One of the two tests is failing

What's happening here is that the first test hides the error div, so when the second test runs, it starts out invisible.

QUnit tests do not run in a predictable order, so you can't rely on the first test running before the second one.

We need some way of tidying up between tests, a bit like `setUp` and `tearDown`, or like the Django test runner would reset the database between each test. The `qunit-fixture` div is what we're looking for. Move the form in there:

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="http://code.jquery.com/jquery.min.js"></script>
```

As you've probably guessed, jQuery resets the content of the fixtures div before each test, so that gets us back to two neatly passing tests:

```
4 assertions of 4 passed, 0 failed.
1. smoke test (0, 2, 2)
2. smoke test 2 (0, 2, 2)
```

Building a JavaScript Unit Test for Our Desired Functionality

Now that we're acquainted with our JavaScript testing tools, we can switch back to just one test, and start to write the real thing:

lists/static/tests/tests.html.

```
    <script>
/*global $, test, equal */

test("errors should be hidden on keypress", function () {
    $('input').trigger('keypress'); // ❶
    equal($('.has-error').is(':visible'), false);
});

    </script>
```

❶ The jQuery `.trigger` method is mainly used for testing. It says "fire off a JavScript DOM event on the element(s)". Here we use the *keypress* event, which is fired off by the browser behind the scenes whenever a user types something into a particular input element.

jQuery is hiding a lot of complexity behind the scenes here. Check out Quirksmode.org for a view on the hideous nest of differences between the different browsers' interpretation of events. The reason that jQuery is so popular is that it just makes all this stuff go away.

And that gives us:

```
0 assertions of 1 passed, 1 failed.
1. errors should be hidden on keypress (1, 0, 1)
    1. failed
        Expected: false
        Result: true
```

Let's say we want to keep our code in a standalone JavaScript file called *list.js*.

lists/static/tests/tests.html.

```
    <script src="qunit.js"></script>
    <script src="../list.js"></script>
    <script>
```

Here's the minimal code to get that test to pass:

lists/static/list.js.

```
$('.has-error').hide();
```

It has an obvious problem. We'd better add another test:

lists/static/tests/tests.html.

```
test("errors should be hidden on keypress", function () {
    $('input').trigger('keypress');
    equal($('.has-error').is(':visible'), false);
});

test("errors not be hidden unless there is a keypress", function () {
    equal($('.has-error').is(':visible'), true);
});
```

Now we get an expected failure:

```
1 assertions of 2 passed, 1 failed.
1. errors should be hidden on keypress (0, 1, 1)
2. errors not be hidden unless there is a keypress (1, 0, 1)
    1. failed
        Expected: true
        Result: false
        Diff: true false
[...]
```

And we can make a more realistic implementation:

lists/static/list.js.

```
$('input').on('keypress', function () { //❶
    $('.has-error').hide();
});
```

❶ This line says: find all the input elements, and for each of them, attach an event listener which reacts *on* keypress events. The event listener is the inline function, which hides all elements that have the class `.has-error`.

That gets our unit tests to pass:

```
2 assertions of 2 passed, 0 failed.
```

Grand, so let's pull in our script, and jQuery, on all our pages:

lists/templates/base.html (ch14l014).

```
</div>
<script src="http://code.jquery.com/jquery.min.js"></script>
<script src="/static/list.js"></script>
</body>

</html>
```

It's good practice to put your script-loads at the end of your body HTML, as it means the user doesn't have to wait for all your JavaScript to load before they can see something on the page. It also helps to make sure most of the DOM has loaded before any scripts run.

Aaaand we run our FT:

```
$ python3 manage.py test functional_tests.test_list_item_validation.\
ItemValidationTest.test_error_messages_are_cleared_on_input
[...]

Ran 1 test in 3.023s

OK
```

Hooray! That's a commit!

Javascript Testing in the TDD Cycle

You may be wondering how these JavaScript tests fit in with our "double loop" TDD cycle. The answer is that they play exactly the same role as our Python unit tests.

1. Write an FT and see it fail.
2. Figure out what kind of code you need next: Python or JavaScript?
3. Write a unit test in either language, and see it fail.
4. Write some code in either language, and make the test pass.
5. Rinse and repeat.

Want a little more practice with JavaScript? See if you can get our error messages to be hidden when the user clicks inside the input element, as well as just when they type in it. You should be able to FT it too.

Columbo Says: Onload Boilerplate and Namespacing

Oh, and one last thing. Whenever you have some JavaScript that interacts with the DOM, it's always good to wrap it in some "onload" boilerplate code to make sure that the page has fully loaded before it tries to do anything. Currently it works anyway, because we've placed the `<script>` tag right at the bottom of the page, but we shouldn't rely on that.

The jQuery `onload` boilerplate is quite minimal:

lists/static/list.js.

```
$(document).ready(function () {
    $('input').on('keypress', function () {
        $('.has-error').hide();
    });
});
```

In addition, we're using the magic `$` function from jQuery, but sometimes other JavaScript libraries try and use that too. It's just an alias for the less contested name `jQuery` though, so here's the standard way of getting more fine-grained control over the namespacing:

lists/static/list.js.

```
jQuery(document).ready(function ($) {
    $('input').on('keypress', function () {
        $('.has-error').hide();
    });
});
```

Read more in the jQuery `.ready()` docs.

We're almost ready to move on to Part III. The last step is to deploy our new code to our servers.

A Few Things That Didn't Make It

- The selector `$(input)` is *way* too greedy; it's assigning a handler to every input element on the page. Try the exercise to add a click handler and you'll realise why that's a problem. Make it more discerning!
- At the moment our test only checks that the JavaScript works on one page. It works because we're including it in *base.html*, but if we'd only added it to *home.html* the tests would still pass. It's a judgement call, but you could choose to write an extra test here.

JavaScript Testing Notes

- One of the great advantages of Selenium is that it allows you to test that your JavaScript really works, just as it tests your Python code.
- There are many JavaScript test running libraries out there. QUnit is closely tied to jQuery, which is the main reason I chose it.
- QUnit mainly expects you to "run" your tests using an actual web browser. This has the advantage that it's easy to create some HTML fixtures that match the kind of HTML your site actually contains, for tests to run against.
- I don't really mean it when I say that JavaScript is awful. It can actually be quite fun. But I'll say it again: make sure you've read *JavaScript: The Good Parts*.

CHAPTER 14

Deploying Our New Code

It's time to deploy our brilliant new validation code to our live servers. This will be a chance to see our automated deploy scripts in action for the second time.

At this point I want to say a huge thanks to Andrew Godwin and the whole Django team. Up until Django 1.7, I used to have a whole long section, entirely devoted to migrations. Migrations now "just work", so I was able to drop it altogether. Thanks for all the great work gang!

Staging Deploy

We start with the staging server:

```
$ cd deploy_tools
$ fab deploy:host=elspeth@superlists-staging.ottg.eu
Disconnecting from superlists-staging.ottg.eu... done.
```

Restart Gunicorn:

```
elspeth@server:$ sudo restart gunicorn-superlists.ottg.eu
```

And run the tests against staging:

```
$ python3 manage.py test functional_tests --liveserver=superlists-staging.ottg.eu
OK
```

Live Deploy

Assuming all is well, we then run our deploy against live:

```
$ fab deploy:host=elspeth@superlists.ottg.eu
```

What to Do If You See a Database Error

Because our migrations introduce a new integrity constraint, you may find that it fails to apply because some existing data violates that constraint.

At this point you have two choices:

- Delete the database on the server and try again. After all, it's only a toy project!
- Or, learn about data migrations. See Appendix D.

Wrap-Up: git tag the New Release

The last thing to do is to tag the release in our VCS—it's important that we're always able to keep track of what's live:

```
$ git tag -f LIVE  # needs the -f because we are replacing the old tag
$ export TAG=`date +DEPLOYED-%F/%H%M`
$ git tag $TAG
$ git push -f origin LIVE $TAG
```

Some people don't like to use `push -f` and update an existing tag, and will instead some kind of version number to tag their releases. Use whatever works for you.

And on that note, we can wrap up Part II, and move on to the more exciting topics that comprise Part III. Can't wait!

PART III
More Advanced Topics

"Oh my gosh, what? Another section? Harry, I'm exhausted, it's already been two hundred pages, I don't think I can handle a whole 'nother section of the book. Particularly not if it's called "Advanced" … maybe I can get away with just skipping it?"

Oh no you can't! This may be called the advanced section, but it's full of really important topics for TDD and web development. No way can you skip it. If anything, it's *even more important* than the first two sections.

We'll be talking about how to integrate third-party systems, and how to test them. Modern web development is all about reusing existing components. We'll cover mocking and test isolation, which is really a core part of TDD, and a technique you're going to need for all but the simplest of codebases. We'll talk about server-side debugging, and test fixtures, and how to set up a Continuous Integration environment. None of these things are take-it-or-leave-it optional luxury extras for your project, they're all vital!

Inevitably, the learning curve does get a little steeper in this section. You may find yourself having to read things a couple of times before they sink in, or you may find that things don't work first go, and that you need to do a bit of debugging on your own. But persist with it! The harder it is, the more rewarding it is. And I'm always happy to help if you're stuck, just drop me an email, *obeythetestinggoat@gmail.com*.

Come on, I promise the best is yet to come!

CHAPTER 15

User Authentication, Integrating Third-Party Plugins, and Mocking with JavaScript

Our beautiful lists site has been live for a few days, and our users are starting to come back to us with feedback. "We love the site", they say, "but we keep losing our lists. Manually remembering URLs is hard. It'd be great if it could remember what lists we'd started".

Remember Henry Ford and faster horses. Whenever you hear a user requirement, it's important to dig a little deeper and think—what is the real requirement here? And how can I make it involve a cool new technology I've been wanting to try out?

Clearly the requirement here is that people want to have some kind of user account on the site. So, without further ado, let's dive into authentication.

Naturally we're not going to mess about with remembering passwords ourselves—besides being *so* '90s, secure storage of user passwords is a security nightmare we'd rather leave to someone else. We'll use a federated authentication system instead.

(If you *insist* on storing your own passwords, Django's default auth module is ready and waiting for you. It's nice and straightforward, and I'll leave it to you to discover on your own.)

In this chapter, we're going to get pretty deep into a testing technique called "mocking". Personally, I know it took me a few weeks to really get my head around mocking, so don't worry if it's confusing at first. In this chapter we do a lot of mocking in JavaScript. In the next chapter we'll do some mocking with Python, which you might find a little easier to grasp. I would recommend reading both of them through together, and just letting the whole concept wash over you; then come back and do them again, and see if you understand all of the steps a little better on the second round.

Do let me know via *obeythetestinggoat@gmail.com* if you feel there's any particular sections where I don't explain things well, or where I'm going too fast.

Mozilla Persona (BrowserID)

But which federated authentication system to use? Oauth? Openid? "Login with Facebook"? Ugh. In my book those all have unacceptable creepy overtones; why should Google or Facebook know what sites you're logging into and when? Thankfully there are still some techno-hippy-idealists out there, and the lovely people at Mozilla have cooked up a privacy-friendly auth mechanism they call "Persona", or sometimes "BrowserID".

The theory goes that your web browser acts as a third party between the website that wants to check your ID, and the website that you will use as a guarantor of your ID. The latter may be Google or Facebook or whomever, but a clever protocol means that they never need know which website you were logging into or when.

Ultimately, Persona may never take off as an authentication platform, but the main lessons from the next couple of chapters should be relevant no matter what third-party auth system you want to integrate:

- Don't test other people's code or APIs.
- But, test that you've integrated them correctly into your own code.
- Check that everything works from the point of view of the user.
- Test that your system degrades gracefully if the third party is down.

Exploratory Coding, aka "Spiking"

Before I wrote this chapter all I'd seen of Persona was a talk at PyCon by Dan Callahan, in which he promised it could be implemented in 30 lines of code, and magic'd his way through a demo—in other words, I knew it not at all.

In Chapter 10 and Chapter 11 we saw that you can use a unit test as a way of exploring a new API, but sometimes you just want to hack something together without any tests at all, just to see if it works, to learn it or get a feel for it. That's absolutely fine. When learning a new tool or exploring a new possible solution, it's often appropriate to leave the rigorous TDD process to one side, and build a little prototype without tests, or perhaps with very few tests. The goat doesn't mind looking the other way for a bit.

This kind of prototyping activity is often called a "spike", for reasons best known.

The first thing I did was take a look at an existing Django-Persona integration called Django-BrowserID, but unfortunately it didn't really support Python 3. I'm sure it will by the time you read this, but I was quietly relieved since I was rather looking forward to writing my own code for this!

It took me about three hours of hacking about, using a combination of code stolen from Dan's talk and the example code on the Persona site, but by the end I had something which just about works. I'll take you on a tour, and then we'll go through and "de-spike" the implementation.

You should go ahead and add this code to your own site too, and then you can have a play with it, try logging in with your own email address, and convince yourself that it really does work.

Starting a Branch for the Spike

Before embarking on a spike, it's a good idea to start a new branch, so you can still use your VCS without worrying about your spike commits getting mixed up with your production code:

```
$ git checkout -b persona-spike
```

Frontend and JavaScript Code

Let's start with the frontend. I was able to cut and paste code from the Persona site and Dan's slides with minimal modification:

lists/templates/base.html (ch15l001).

```
<script src="http://code.jquery.com/jquery.min.js"></script>
<script src="/static/list.js"></script>
<script src="https://login.persona.org/include.js"></script>
<script>
$(document).ready(function() {

var loginLink = document.getElementById('login');
if (loginLink) {
  loginLink.onclick = function() { navigator.id.request(); };
}

var logoutLink = document.getElementById('logout');
if (logoutLink) {
  logoutLink.onclick = function() { navigator.id.logout(); };
}

var currentUser = '{{ user.email }}' || null;
var csrf_token = '{{ csrf_token }}';
console.log(currentUser);

navigator.id.watch({
  loggedInUser: currentUser,
  onlogin: function(assertion) {
    $.post('/accounts/login', {assertion: assertion, csrfmiddlewaretoken: csrf_token})
    .done(function() { window.location.reload(); })
```

```
      .fail(function() { navigator.id.logout();});
    },
    onlogout: function() {
      $.post('/accounts/logout')
      .always(function() { window.location.reload(); });
    }
  });

  });
  </script>
```

The Persona JavaScript library gives us a special `navigator.id` object. We bind its `request` method to our link called "login" (which I've put in any old where at the top of the page), and similarly a "logout" link gets bound to a `logout` function:

lists/templates/base.html (ch15l002).

```
<body>
<div class="container">

    <div class="navbar">
        {% if user.email %}
            <p>Logged in as {{ user.email}}</p>
            <p><a id="logout" href="{% url 'logout' %}">Sign out</a></p>
        {% else %}
            <a href="#" id="login">Sign in</a>
        {% endif %}
        <p>User: {{user}}</p>
    </div>

    <div class="row">
    [...]
```

The Browser-ID Protocol

Persona will now pop up its authentication dialog box if users click the log in link. What happens next is the clever part of the Persona protocol: the user enters an email address, and the browser takes care of validating that email address, by taking the user to the email provider (Google, Yahoo, or whoever), and validating it with them.

Let's say it's Google: Google asks the user to confirm their username and password, and maybe even does some two-factor auth wizardry, and is then prepared to confirm to your browser that you are who you say you are. Google then passes a certificate back to the browser, which is cryptographically signed to prove it's from Google, and which contains the user's email address.

At this point the browser can trust that you do own that email address, and it can incidentally reuse that certificate for any other websites that use Persona.

Now it combines the certificate with the domain name of the website you want to log into in to a blob called an "assertion", and sends them on to our site for validation.

This is the point between the `navigator.id.request` and the `navigator.id.watch` callback for `onlogin`—we send the assertion via POST to the login URL on our site, which I've put at *accounts/login*.

On the server, we now have the job of verifying the assertion: is it really proof that the user owns that email address? Our server can check, because Google has signed part of the assertion with its public key. We can either write code to do the crypto for this step ourselves, or we can use a public service from Mozilla to do it for us.

Yes, letting Mozilla do it for us totally defeats the whole privacy point, but it's the *principle*. We could do it ourselves if we wanted to. It's left as an exercise for the reader! There are more details on the Mozilla site, including all the clever public key crypto that keeps Google from knowing what site you want to log in to, but also stops replay attacks and so on. Smart.

The Server Side: Custom Authentication

Next we prep an app for our accounts stuff:

```
$ python3 manage.py startapp accounts
```

Here's the view that handles the POST to *accounts/login*:

accounts/views.py.

```
import sys
from django.contrib.auth import authenticate
from django.contrib.auth import login as auth_login
from django.shortcuts import redirect

def login(request):
    print('login view', file=sys.stderr)
    # user = PersonaAuthenticationBackend().authenticate(request.POST['assertion'])
    user = authenticate(assertion=request.POST['assertion'])
    if user is not None:
        auth_login(request, user)
    return redirect('/')
```

You can see that it's clearly "spike" code from things like the commented-out line, evidence of an early experiment that failed.

Here's the `authenticate` function, which is implemented as a custom Django "authentication backend". (We could have done it inline in the view, but using a backend is the Django recommended way. It would let us reuse the authentication system in the admin site, for example.)

accounts/authentication.py.

```
import requests
import sys
from accounts.models import ListUser

class PersonaAuthenticationBackend(object):

    def authenticate(self, assertion):
        # Send the assertion to Mozilla's verifier service.
        data = {'assertion': assertion, 'audience': 'localhost'}
        print('sending to mozilla', data, file=sys.stderr)
        resp = requests.post('https://verifier.login.persona.org/verify', data=data)
        print('got', resp.content, file=sys.stderr)

        # Did the verifier respond?
        if resp.ok:
            # Parse the response
            verification_data = resp.json()

            # Check if the assertion was valid
            if verification_data['status'] == 'okay':
                email = verification_data['email']
                try:
                    return self.get_user(email)
                except ListUser.DoesNotExist:
                    return ListUser.objects.create(email=email)


    def get_user(self, email):
        return ListUser.objects.get(email=email)
```

This code is copy-pasted directly from the Mozilla site, as you can see from the explanatory comments.

You'll need to `pip install requests` into your virtualenv. If you've never used it before, Requests is a great alternative to the Python standard library tools for HTTP requests.

To finish off the job of customising authentication in Django, we just need a custom user model:

accounts/models.py.

```
from django.contrib.auth.models import AbstractBaseUser, PermissionsMixin
from django.db import models

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
```

That's what I call a minimal user model! One field, none of this firstname/lastname/username nonsense, and, pointedly, no password! Somebody else's problem! But, again, you can see that this code isn't ready for production, from the commented-out lines to the hardcoded harry email address.

At this point I'd recommend a little browse through the Django auth documentation.

Aside from that, you need a model manager for the user:

accounts/models.py (ch15l006).

```
from django.contrib.auth.models import AbstractBaseUser, BaseUserManager, PermissionsMixin

class ListUserManager(BaseUserManager):

    def create_user(self, email):
        ListUser.objects.create(email=email)

    def create_superuser(self, email, password):
        self.create_user(email)
```

A logout view:

accounts/views.py (ch15l007).

```
from django.contrib.auth import login as auth_login, logout as auth_logout
[...]

def logout(request):
    auth_logout(request)
    return redirect('/')
```

Some URLs for our two views:

superlists/urls.py (ch15l008).

```
urlpatterns = patterns('',
    url(r'^$', 'lists.views.home_page', name='home'),
    url(r'^lists/', include('lists.urls')),
    url(r'^accounts/', include('accounts.urls')),
    # url(r'^admin/', include(admin.site.urls)),
)
```

and

accounts/urls.py.

```
from django.conf.urls import patterns, url

urlpatterns = patterns('',
    url(r'^login$', 'accounts.views.login', name='login'),
    url(r'^logout$', 'accounts.views.logout', name='logout'),
)
```

Almost there. We switch on the auth backend and our new accounts app in *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',
)

AUTH_USER_MODEL = 'accounts.ListUser'
AUTHENTICATION_BACKENDS = (
    'accounts.authentication.PersonaAuthenticationBackend',
)

MIDDLEWARE_CLASSES = (
[...]
```

And a quick `makemigrations` to make the new user model real:

```
$ python3 manage.py makemigrations
Migrations for 'accounts':
  0001_initial.py:
    - Create model ListUser
```

And a `migrate` to build the database:

```
$ python3 manage.py migrate
[...]
Running migrations:
  Applying accounts.0001_initial... OK
```

And we should be all done! Why not spin up a dev server with `runserver` and see how it all looks (Figure 15-1)?

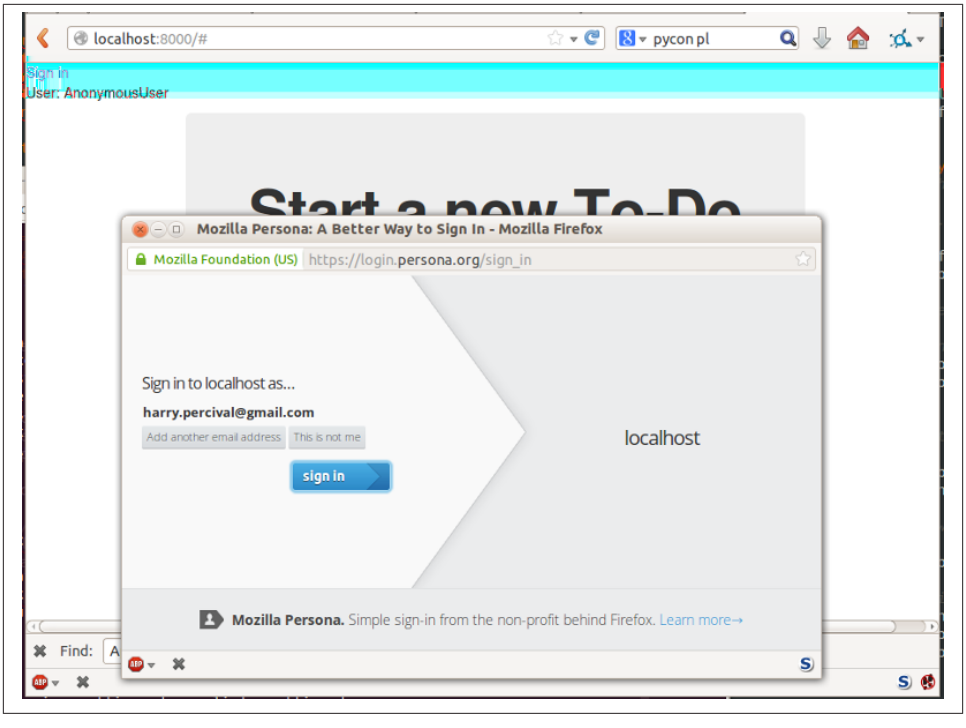


Figure 15-1. It works! It works! Mwahahahaha.

That's pretty much it! Along the way, I had to fight pretty hard, including debugging Ajax requests by hand in the Firefox console (see Figure 15-2), catching infinite page-refresh loops, stumbling over several missing attributes on my custom user model (because I didn't read the docs properly), and even one point switching to the dev version of Django to overcome a bug, which thankfully turned out to be irrelevant.

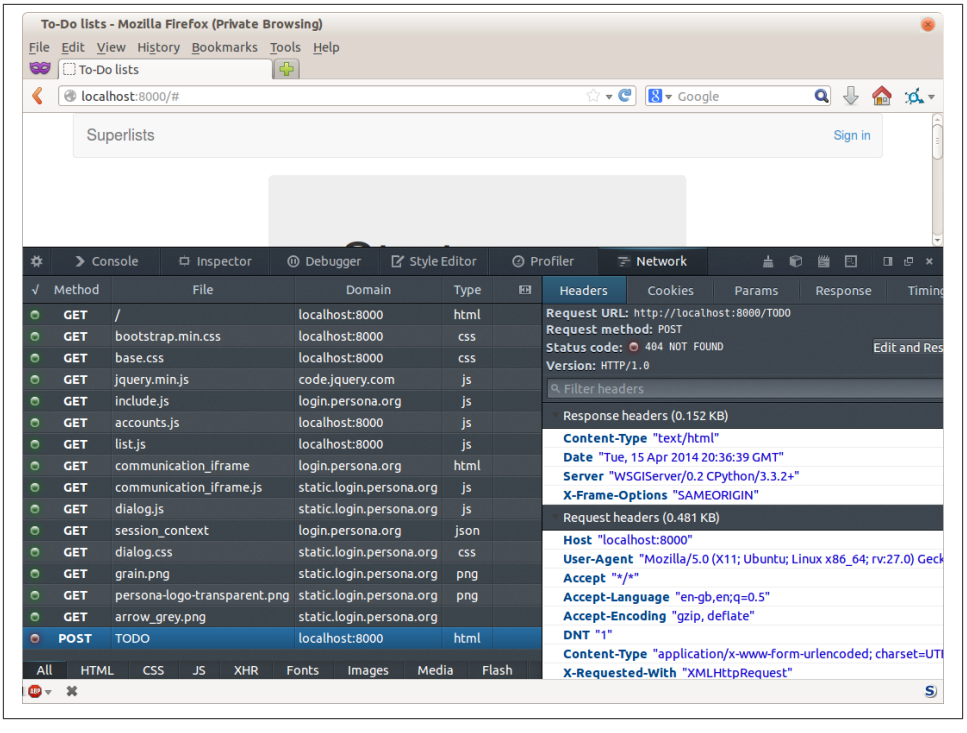


Figure 15-2. Debugging Ajax requests in the Firefox network console

If it's not working when you try it manually, and you see "audience mismatch" errors in the console, make sure you're visiting the site via *http://localhost:8000*, and not *127.0.0.1*.

Aside: Logging to stderr

While spiking, it's pretty critical to be able to see exceptions that are being generated by your code. Annoyingly, Django doesn't send all exceptions to the terminal by default, but you can make it do so with a variable called `LOGGING` in *settings.py*:

superlists/settings.py (ch15l011).

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
```

```
            },
        },
        'loggers': {
            'django': {
                'handlers': ['console'],
            },
        },
        'root': {'level': 'INFO'},
    }
```

Django uses the rather "enterprisey" logging module from the Python standard library, which, although very fully featured, does suffer from a fairly steep learning curve. It's covered a little more in Chapter 17, and in the Django docs.

But we now have a working solution! Let's commit it on our spike branch:

```
$ git status
$ git add accounts
$ git commit -am"spiked in custom auth backend with persona"
```

Time to de-spike!

De-spiking

De-spiking means rewriting your prototype code using TDD. We now have enough information to "do it properly". So what's the first step? An FT of course!

We'll stay on the spike branch for now, to see our FT pass against our spiked code. Then we'll go back to master, and commit just the FT.

Here's the basic outline:

functional_tests/test_login.py.

```
from .base import FunctionalTest

class LoginTest(FunctionalTest):

    def test_login_with_persona(self):
        # Edith goes to the awesome superlists site
        # and notices a "Sign in" link for the first time.
        self.browser.get(self.server_url)
        self.browser.find_element_by_id('login').click()

        # A Persona login box appears
        self.switch_to_new_window('Mozilla Persona')  #❶

        # Edith logs in with her email address
        ## Use mockmyid.com for test email
        self.browser.find_element_by_id(
            'authentication_email'  #❷
        ).send_keys('edith@mockmyid.com') #❸
```

```
        self.browser.find_element_by_tag_name('button').click()

        # The Persona window closes
        self.switch_to_new_window('To-Do')

        # She can see that she is logged in
        self.wait_for_element_with_id('logout')  #❹
        navbar = self.browser.find_element_by_css_selector('.navbar')
        self.assertIn('edith@mockmyid.com', navbar.text)
```

❶ ❹ The FT needs a couple of helper functions, both of which do something that's very common in Selenium testing: they wait for something to happen. Listings for them follow.

❷ I found the ID of the Persona login box by opening the site manually, and using the Firefox debug toolbar (Ctrl+Shift+I). See Figure 15-3.

❸ Rather than using a "real" email address and having to click through their authentication screens, we use a "fake" provider. MockMyID is one; you can also check out Persona Test User.

Evaluate Third-Party Systems' Test Infrastructure

Testing should be part of how you evaluate third-party systems. When you integrate with an external service, you're going to have to think through how you're going to work with it in your functional tests.

Often you can just use the same service in your tests and in "real life". But sometimes you're going to want to run against a "test" version of the third-party service. In the case of this integration with Persona, we could have used a "real" email address; when I first wrote this chapter, I actually had an FT that clicked through to Yahoo.com, and logged in with a throwaway account I'd created. The problem is that it made the FT totally reliant on particular details of Yahoo's email login screens, which can change at any time.

Instead, MockMyID and PersonaTestUser are both linked to from the Persona documentation, and they work very smoothly, letting us test just the important parts of the integration.

Perhaps more critically, think about payment systems. If you start integrating payments, they're going to be one of the most important parts of your site, and you're going to want to make sure they're tested thoroughly … but you don't want to be putting actual transactions on real credit cards through, every time you run an FT! So most providers will provide a "test" version of their payments API. These vary in quality (naming no names), so make sure you investigate them thoroughly.

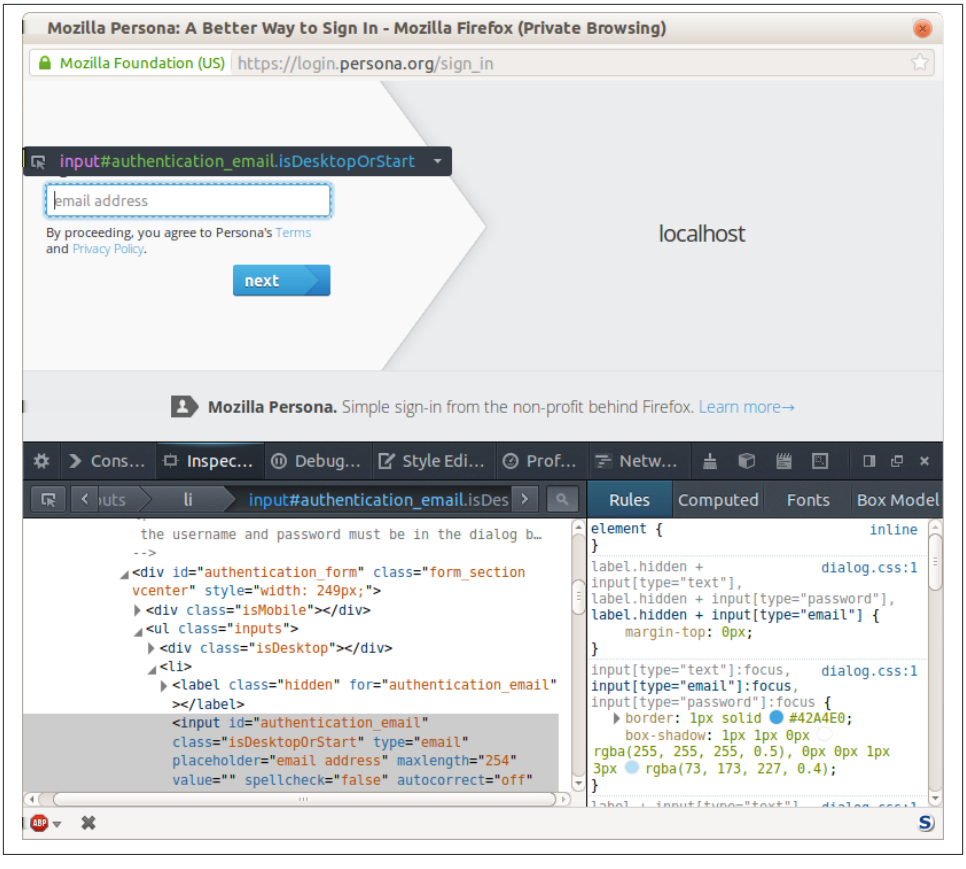


Figure 15-3. Using the Debug toolbar to find locators

A Common Selenium Technique: Explicit Waits

Here's the first of the two "wait" helper functions:

functional_tests/test_login.py (ch15l014).

```
import time
[...]

    def switch_to_new_window(self, text_in_title):
        retries = 60
        while retries > 0:
            for handle in self.browser.window_handles:
                self.browser.switch_to_window(handle)
                if text_in_title in self.browser.title:
                    return
            retries -= 1
            time.sleep(0.5)
        self.fail('could not find window')
```

In this one we've "rolled our own" wait—we iterate through all the current browser windows, looking for one with a particular title. If we can't find it, we do a short wait, and try again, decrementing a retry counter.

This is such a common pattern in Selenium tests that the team created an API for waiting—it doesn't quite handle all use cases though, so that's why we had to roll our own the first time around. When doing something simpler like waiting for an element with a given ID to appear on the page, we can use the `WebDriverWait` class:

functional_tests/test_login.py (ch15l015).

```
from selenium.webdriver.support.ui import WebDriverWait
[...]

    def wait_for_element_with_id(self, element_id):
        WebDriverWait(self.browser, timeout=30).until(
            lambda b: b.find_element_by_id(element_id)
        )
```

This is what Selenium calls an "explicit wait". If you remember, we already defined an "implicit wait" in `FunctionalTest.setUp`. We set that to just three seconds, which is fine in most cases, but when we're waiting for an external service like Persona, we sometimes need to bump that default timeout.

There are more examples in the Selenium docs, but I actually found reading the source code more instructive—there are good docstrings!

`implicitly_wait` is unreliable, especially once JavaScript is involved. Prefer the "wait-for" pattern in your FT whenever you need to check for asynchronous interactions on your pages. We'll see this again in Chapter 20.

And if we run the FT, it works!

```
$ python3 manage.py test functional_tests.test_login
Creating test database for alias 'default'...
Not Found: /favicon.ico
login view
sending to mozilla {'assertion': [...]
[...]

got b'{"audience":"localhost","expires":[...]
[...]

.
 ----------------------------------------------------------------------
Ran 1 test in 32.222s

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

You can even see some of the debug output I left in my spiked view implementations. Now it's time to revert all of our temporary changes, and reintroduce them one by one in a test-driven way.

Reverting Our Spiked Code

```
$ git checkout master # switch back to master branch
$ rm -rf accounts # remove any trace of spiked code
$ git add functional_tests/test_login.py
$ git commit -m "FT for login with Persona"
```

Now we rerun the FT and let it drive our development:

```
$ python3 manage.py test functional_tests.test_login
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"id","selector":"login"}' ; Stacktrace:
[...]
```

The first thing it wants us to do is add a login link. Incidentally, I prefer prefixing HTML IDs with `id_`; it's a convention to make it easy to tell the difference between classes and IDs in HTML and CSS. So let's tweak the FT first:

functional_tests/test_login.py (ch15l017).

```
        self.browser.find_element_by_id('id_login').click()
        [...]
        self.wait_for_element_with_id('id_logout')
```

Next a "do-nothing" login link. Bootstrap has some built-in classes for navigation bars, so we'll use them:

lists/templates/base.html.

```
    <div class="container">

        <nav class="navbar navbar-default" role="navigation">
            <a class="navbar-brand" href="/">Superlists</a>
            <a class="btn navbar-btn navbar-right" id="id_login" href="#">Sign in</a>
        </nav>

        <div class="row">
        [...]
```

After 30 seconds, that gives:

```
    AssertionError: could not find window
```

License to move on! Next thing: more JavaScript.

JavaScript Unit Tests Involving External Components: Our First Mocks!

To get our FT further, we're going to need to get the Persona window to pop up. For that, we'll need to de-spike our client-side JavaScript code that uses the Persona libraries. We'll test-drive that using JavaScript unit tests and mocking.

Housekeeping: A Site-Wide Static Files Folder

A bit of housekeeping first: create a site-wide static files directory inside *superlists/superlists*, and move all the Bootsrap CSS, QUnit code, and *base.css* into it, so it looks like this:

```
$ tree superlists -L 3 -I __pycache__
superlists
├── __init__.py
├── settings.py
├── static
│   ├── base.css
│   ├── bootstrap
│   │   ├── css
│   │   ├── fonts
│   │   └── js
│   └── tests
│       ├── qunit.css
│       └── qunit.js
├── urls.py
└── wsgi.py

6 directories, 7 files
```

Always do a commit before and after a bit of housekeeping like this.

That means adjusting our existing JavaScript unit tests:

lists/static/tests/tests.html (ch15l020).

```
<link rel="stylesheet" href="../../../superlists/static/tests/qunit.css">

[...]

<script src="http://code.jquery.com/jquery.min.js"></script>
<script src="../../../superlists/static/tests/qunit.js"></script>
<script src="../list.js"></script>
```

And we check they still work, by opening them up in a browser:

```
2 assertions of 2 passed, 0 failed.
```

Here's how we tell our settings file about the new static folder:

superlists/settings.py.

```
[...]
STATIC_ROOT = os.path.join(BASE_DIR, '../static')
STATICFILES_DIRS = (
    os.path.join(BASE_DIR, 'superlists', 'static'),
)
```

I recommend reintroducing the `LOGGING` setting from earlier at this point. There's no need for an explicit test for it; our current test suite will let us know in the unlikely event that it breaks anything. As we'll find out in Chapter 17, it'll be useful for debugging later.

And we can quickly run the layout + styling FT to check the CSS all still works:

```
$ python3 manage.py test functional_tests.test_layout_and_styling
[...]
OK
```

Next, create an app called `accounts` to hold all the code related to login. That will include our Persona JavaScript stuff:

```
$ python3 manage.py startapp accounts
$ mkdir -p accounts/static/tests
```

That's the housekeeping done. Now's a good time for a commit. Then, let's take another look at our spiked-in javascript:

```
var loginLink = document.getElementById('login');
if (loginLink) {
  loginLink.onclick = function() { navigator.id.request(); };
}
```

Mocking: Who, Why, What?

We want our login link's on-click to be bound to a function provided by the Persona library, `navigator.id.request`.

Now we don't want to call the *actual* third-party function in our unit tests, because we don't want our unit tests popping up Persona windows all over the shop. So instead, we are going to do what's called "mocking it out": creating a "fake" or "mock" implementation of the third-party API for our tests to run against.

What we're going to do is replace the real `navigator` object with a *fake* one that we've built ourselves, one that will be able to tell us what happens to it.

I had hoped that our first Mock example was going to be in Python, but it looks like it's going to be JavaScript instead. Needs must. You may find you it's worth rereading the rest of the chapter a couple of times after you get to the end of it, to let it all sink in.

Namespacing

In the context of *base.html*, `navigator` is just an object in the global scope, as provided by the *include.js* `<script>` tag that we get from Mozilla. Testing global variables is a pain though, so we can turn it into a local variable by passing it into an "initialize"[1] function. The code we'll end up with in *base.html* will look like this:

lists/templates/base.html.

```
<script src="/static/accounts/accounts.js"></script>
<script>
    $(document).ready(function() {

        Superlists.Accounts.initialize(navigator)

    });
</script>
```

I've specified that our `initialize` function will be *namespaced* inside some nested objects, `Superlists.Accounts`. JavaScript suffers from a programming model that's tied into a global scope, and this sort of namespacing/naming convention helps to keep things under control. Lots of JavaScript libraries might want to call a function `initialize`, but very few will call it `Superlists.Accounts.initialize`![2]

This call to `initialize` is simple enough that I'm happy it doesn't need any unit tests of its own.

A Simple Mock to Unit Tests Our initialize Function

The `initialize` function itself we *will* test. Copy the lists tests across to get the boilerplate HTML, and then adjust the following:

accounts/static/tests/tests.html.

```
<div id="qunit-fixture">
    <a id="id_login">Sign in</a>
</div>
```

1. UK-English speakers may bristle at that incorrect spelling of the word "initialise". I know, it grates with me too. But it's an increasingly accepted convention to use American spelling in code. It makes it easier to search, for example, and just to work together more generally, if we all agree on how words are spelt. We have to accept that we're in the minority here, and this is one battle we've probably lost.
2. The new shiny in the JavaScript world for avoiding namespacing problems is called *require.js*. It was one thing too many to squeeze into this book, but you should check it out.

```
        <script src="http://code.jquery.com/jquery.min.js"></script>
        <script src="../../../superlists/static/tests/qunit.js"></script>
        <script src="../accounts.js"></script>
        <script>
    /*global $, test, equal, sinon, Superlists */

    test("initialize binds sign in button to navigator.id.request", function () {
        var requestWasCalled = false; //❶
        var mockRequestFunction = function () { requestWasCalled = true; }; //❷
        var mockNavigator = { //❸
            id: {
                request: mockRequestFunction
            }
        };

        Superlists.Accounts.initialize(mockNavigator); //❹

        $('#id_login').trigger('click'); //❺

        equal(requestWasCalled, true); //❻
    });

        </script>
```

One of the best ways to understand this test, or indeed any test, is to work backwards. The first thing we see is the assertion:

❻ We are asserting that a variable called `requestWasCalled` is `true`. We're checking that, one way or another, the `request` function, as in `navigator.id.request`, was called.

❺ Called when? When a click event happens to the `id_login` element.

❹ Before we trigger that click event, we call our `Superlists.Accounts.initialize` function, just like we will on the real page. The only difference is, instead of passing it the real global `navigator` object from Persona, we pass in a fake one called `mockNavigator`.[3]

❸ That's defined as a generic JavaScript object, with an attribute called `id` which in turn has an attribute called `request`, which we're assigning to a variable called `mockRequestFunction`.

❷ `mockRequestFunction` we define as a very simple function, which if called will simply set the value of the `requestWasCalled` variable to `true`.

❶ And finally (firstly?) we make sure that `requestWasCalled` starts out as `false`.

3. I've called this object a "mock", but it's probably more correctly called a "spy". We don't have to concern ourselves with the differences in this book, but for more on the general class of tools called "Test Doubles", including the difference between stubs, mocks, fakes, and spies, see Mocks, Fakes and Stubs by Emily Bache.

The upshot of all this is: the only way this test will pass is if our `initialize` function binds the `click` event on `id_login` to the method `.id.request` of the object we pass it. If we get the tests passing when we use the mock object, we are reassured that our `initialize` function will also do the right thing when we give it a real object on our real page.

Does that make sense? Let's play around with the test and see if we can get the hang of it.

When testing events on DOM elements, you need an actual element to trigger events against, and to register listeners on. If you forget, it's a particularly fiendish test bug, because `.trigger` will just silently no-op, and you'll be left scratching your head about why it's not working. So don't forget to add the `<div id="id_login">` inside the `qunit-fixture` div!

Our first error is this:

```
1. Died on test #1
@file:///workspace/superlists/accounts/static/tests/tests.html:35:
Superlists is not defined
```

That's the equivalent of an `ImportError` in Python. Let's start work on *accounts/static/accounts.js*:

accounts/static/accounts.js.

```
window.Superlists = null;
```

Just as in Python we might do `Superlists = None`, here we do `window.Superlists = null`. Using `window.` makes sure we get the global object:

```
1. Died on test #1
@file:///workspace/superlists/accounts/static/tests/tests.html:35:
Superlists is null
```

OK, next baby step or two:

accounts/static/accounts.js.

```
window.Superlists = {
    Accounts: {}
};
```

gives:[4]

4. In the real world, when setting up a namespace like this, you'd want to follow a sort of "add-or-create" pattern, so that, if there's already a `window.Superlists` in the scope, we extend it rather than replacing it. `window.Superlists = window.Superlists || {}` is one formulation, and jQuery's `$.extend` is another possibilty. But, there's already a lot of content in this chapter, and I thought this was probably one too many things to talk about!

```
Superlists.Accounts.initialize is not a function
```

So let's make it a function:

accounts/static/accounts.js.

```
window.Superlists = {
    Accounts: {
        initialize: function () {}
    }
};
```

And now we get a real test failure instead of just errors:

```
1. initialize binds sign in button to navigator.id.request (1, 0, 1)

    1. failed
        Expected: true
        Result: false
```

Next—let's separate defining our `initialize` function from the part where we export it into the `Superlists` namespace. We'll also do a `console.log`, which is the JavaScript equivalent of a debug-print, to take a look at what the initialize function is being called with:

accounts/static/accounts.js (ch15l028).

```
var initialize = function (navigator) {
    console.log(navigator);
};

window.Superlists = {
    Accounts: {
        initialize: initialize
    }
};
```

In Firefox and I believe Chrome also, you can use the shortcut Ctrl-Shift-I to bring up the JavaScript console, and see the [object Object] that was logged (see Figure 15-4). If you click on it, you can see it has the properties we defined in our test: an `id`, and inside that, a function called `request`.

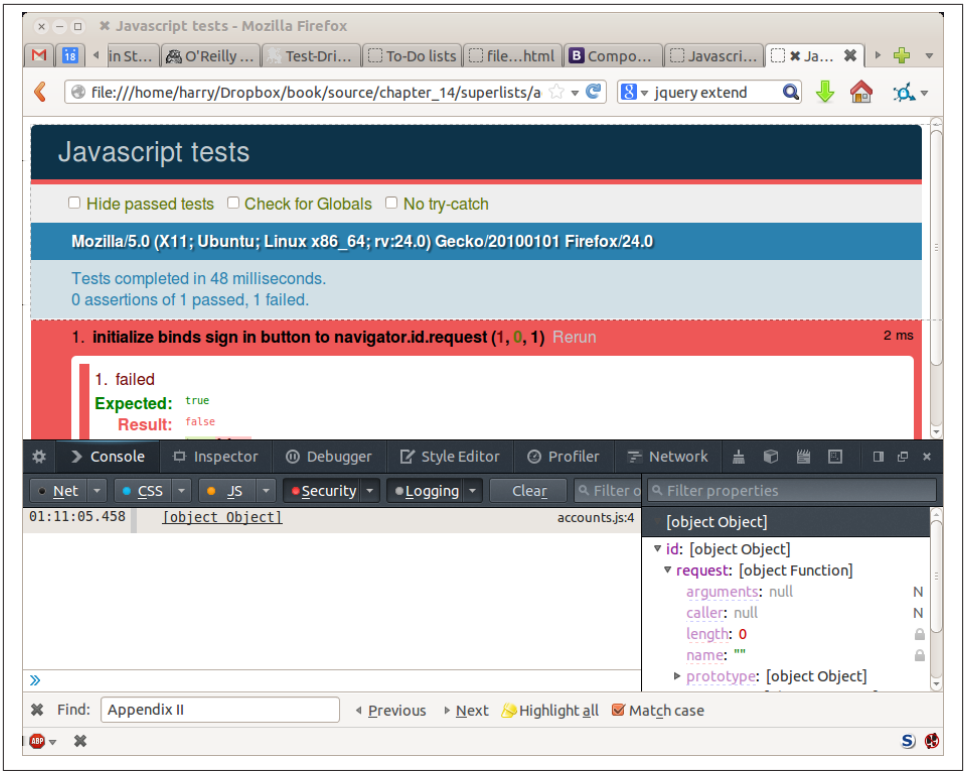


Figure 15-4. Debugging in the JavaScript console

So let's now just pile in and get the test to pass:

accounts/static/accounts.js (ch15l029).

```
var initialize = function (navigator) {
    navigator.id.request();
};
```

That gets the tests to pass, but it's not quite the implementation we want. We're calling `navigator.id.request` always, instead of only on click. We'll need to adjust our tests.

```
1 assertions of 1 passed, 0 failed.
1. initialize binds sign in button to navigator.id.request (0, 1, 1)
```

Before we do, let's just have a play around to see if we really understand what's going on. What happens if we do this?

accounts/static/accounts.js (ch15l029-1).

```
var initialize = function (navigator) {
    navigator.id.request();
    navigator.id.doSomethingElse();
};
```

We get:

```
1. Died on test #1
@file:///workspace/superlists/accounts/static/tests/tests.html:35:
navigator.id.doSomethingElse is not a function
```

You see, the mock navigator object that we pass in is entirely under our control. It has only the attributes and methods we give it. You can play around with it now if you like:

accounts/static/tests/tests.html.

```
var mockNavigator = {
    id: {
        request: mockRequestFunction,
        doSomethingElse: function () { console.log("called me!");}
    }
};
```

That will give you a pass, and if you open up the debug window, you'll see:

```
[01:22:27.456] "called me!"
```

Does that help to see what's going on? Let's revert those last two changes, and tweak our unit test so that it checks the `request` function is only called *after* we fire off the click event. We also add some error messages to help see which of the two `equal` assertions is failing:

accounts/static/tests/tests.html (ch15l032).

```
var mockNavigator = {
    id: {
        request: mockRequestFunction
    }
};
Superlists.Accounts.initialize(mockNavigator);
equal(requestWasCalled, false, 'check request not called before click');
$('#id_login').trigger('click');
equal(requestWasCalled, true, 'check request called after click');
```

Assertion messages (the third argument to `equal`), in QUnit, are actually "success" messages. Rather than only being displayed if the test fails, they are also displayed when the test passes. That's why they have the positive phrasing.

Now we get a neater failure:

```
1 assertions of 2 passed, 1 failed.
1. initialize binds sign in button to navigator.id.request (1, 1, 2)
    1. check request not called before click
        Expected: false
        Result: true
```

So let's make it so that the call to `navigator.id.request` only happens if our `id_log in` is clicked:

accounts/static/accounts.js (ch15l033).

```
/*global $ */

var initialize = function (navigator) {
    $('#id_login').on('click', function () {
        navigator.id.request();
    });
};
[...]
```

That passes. A good start! Let's try pulling it into our template:

lists/templates/base.html.

```
<script src="http://code.jquery.com/jquery.min.js"></script>
<script src="https://login.persona.org/include.js"></script>
<script src="/static/accounts.js"></script>
<script src="/static/list.js"></script>
<script>
    /*global $, Superlists, navigator */
    $(document).ready(function () {
        Superlists.Accounts.initialize(navigator);
    });
</script>
</body>
```

We also need to add the `accounts` app to *settings.py*, otherwise it won't be serving the static file at *accounts/static/accounts.js*:

superlists/settings.py.

```
+++ b/superlists/settings.py
@@ -37,4 +37,5 @@ INSTALLED_APPS = (
     'lists',
+    'accounts',
 )
```

A quick check on the FT … doesn't get any further unfortunately. To see why, we can open up the site manually, and check the JavaScript debug console:

```
[01:36:54.572] Error: navigator.id.watch must be called before
navigator.id.request @ https://login.persona.org/include.js:8
```

More Advanced Mocking

We now need to call Mozilla's `navigator.id.watch` correctly. Taking another look at our spike, it should be something like this:

```
var currentUser = '{{ user.email }}' || null;
var csrf_token = '{{ csrf_token }}';
console.log(currentUser);

navigator.id.watch({
  loggedInUser: currentUser, //❶
  onlogin: function(assertion) {
    $.post('/accounts/login', {assertion: assertion, csrfmiddlewaretoken: csrf_token}) //❷
    .done(function() { window.location.reload(); })
    .fail(function() { navigator.id.logout();});
```

```
    },
    onlogout: function() {
      $.post('/accounts/logout')
      .always(function() { window.location.reload(); });
    }
  });
```

Decoding that, the `watch` function needs to know a couple of things from the global scope:

❶ The current user's email, to be passed in as the `loggedInUser` parameter to `watch`.

❷ The current CSRF token, to pass in the Ajax POST request to the login view.[5]

We've also got two hardcoded URLs in there, which would be better to get from Django, something like this:

```
var urls = {
    login: "{% url 'login' %}",
    logout: "{% url 'logout' %}",
};
```

So that would be a third parameter to pass in from the global scope. We've already got an `initialize` function, so let's imagine using it like this:

```
Superlists.Accounts.initialize(navigator, user, token, urls);
```

Using a sinon.js mock to check we call the API correctly

"Rolling your own" mocks is possible as we've seen, and JavaScript actually makes it relatively easy, but using a mocking library can save us a lot of heavy lifting. The most popular one in the JavaScript world is called *sinon.js*. Let's download it (from *http://sinonjs.org*) and put it in our site-wide static tests folder:

```
$ tree superlists/static/tests/
superlists/static/tests/
├── qunit.css
├── qunit.js
└── sinon.js
```

Next we include it in our accounts tests:

accounts/static/tests/tests.html.

```
    <script src="http://code.jquery.com/jquery.min.js"></script>
    <script src="../../../superlists/static/tests/qunit.js"></script>
    <script src="../../../superlists/static/tests/sinon.js"></script>
    <script src="../accounts.js"></script>
```

5. Incidentally, notice we use `{{ csrf_token }}` which gives you the raw string token, rather than `{% csrf_token%}` which would give us a full HTML tag, `<input type="hidden" name="etc etc`.

And now we can write a test that uses Sinon's mock object:[6]

accounts/static/tests/tests.html (ch15l038).

```
test("initialize calls navigator.id.watch", function () {
    var user = 'current user';
    var token = 'csrf token';
    var urls = {login: 'login url', logout: 'logout url'};
    var mockNavigator = {
        id: {
            watch: sinon.mock() //❶
        }
    };

    Superlists.Accounts.initialize(mockNavigator, user, token, urls);

    equal(
        mockNavigator.id.watch.calledOnce, //❷
        true,
        'check watch function called'
    );
});
```

❶ We create a mock navigator object as before, but now instead of hand-crafting a function to mock out the function we're interested in, we use a `sinon.mock()` object.

❷ This object then records what happens to it inside special properties like `calledOnce`, which we can make assertions against.

There's more info in the Sinon docs—the front page actually has quite a good overview.

Here's our expected test failure:

```
2 assertions of 3 passed, 1 failed.

1. initialize binds sign in button to navigator.id.request (0, 2, 2)
2. initialize calls navigator.id.watch (1, 0, 1)
    1. check watch function called
        Expected: true
        Result: false
```

We add in the call to `watch`…

accounts/static/accounts.js.

```
var initialize = function (navigator) {
    $('#id_login').on('click', function () {
        navigator.id.request();
    });
```

6. Sinon also has more specialised objects for "spies" and "stubs". Mocks can do everything that spies and stubs can do though, so I figured, one less piece of terminology would keep things simple.

```
        navigator.id.watch();
    };
```

But that breaks the other test!

```
1 assertions of 2 passed, 1 failed.

1. initialize binds sign in button to navigator.id.request (1, 0, 1)
    1. Died on test #1
@file:///workspace/superlists/accounts/static/tests/tests.html:36:
missing argument 1 when calling function navigator.id.watch

2. initialize calls navigator.id.watch (0, 1, 1)
```

That was a puzzler—that "missing argument 1 when calling function navigator.id.watch" took me a while to figure out. Turns out that, in Firefox, `.watch` is a function on every object. We'll need to mock it out in the previous test too:

accounts/static/tests/tests.html.

```
test("initialize binds sign in button to navigator.id.request", function () {
    var requestWasCalled = false;
    var mockRequestFunction = function () { requestWasCalled = true; };
    var mockNavigator = {
        id: {
            request: mockRequestFunction,
            watch: function () {}
        }
    };
    [...]
```

And we're back to passing tests:

```
3 assertions of 3 passed, 0 failed.

1. initialize binds sign in button to navigator.id.request (0, 2, 2)
2. initialize calls navigator.id.watch (0, 1, 1)
```

Checking Call Arguments

We're not calling the `watch` function correctly yet—it needs to know the current user, and we have to set up a couple of callbacks for login and logout. Let's start with the user:

accounts/static/tests/tests.html (ch15l042).

```
test("watch sees current user", function () {
    var user = 'current user';
    var token = 'csrf token';
    var urls = {login: 'login url', logout: 'logout url'};
    var mockNavigator = {
        id: {
            watch: sinon.mock()
        }
    };

    Superlists.Accounts.initialize(mockNavigator, user, token, urls);
```

```
        var watchCallArgs = mockNavigator.id.watch.firstCall.args[0];
        equal(watchCallArgs.loggedInUser, user, 'check user');
    });
```

We have a very similar setup (which is a code smell incidentally—on the next test, we're going to want to do some de-duplication of test code). Then we use the `.firstCall.args[0]` property on the mock to check on the parameter we passed to the `watch` function (`args` being a list of positional arguments). That gives us:

```
3. watch sees current user (1, 0, 1)
    1. Died on test #1
@file:///workspace/superlists/accounts/static/tests/tests.html:72:
watchCallArgs is undefined
```

because we're not currently passing any arguments to `watch`. Step by step, we can do:

accounts/static/accounts.js (ch15l043).

```
        navigator.id.watch({});
```

and get a clearer error message:

```
3. watch sees current user (1, 0, 1)
    1. check user
        Expected: "current user"
        Result: undefined
```

and fix it thusly:

accounts/static/accounts.js (ch15l044).

```
var initialize = function (navigator, user, token, urls) {
    [...]

    navigator.id.watch({
        loggedInUser: user
    });
```

Good.

```
4 assertions of 4 passed, 0 failed.
```

QUnit setup and teardown, Testing Ajax

Next we need to check the `onlogin` callback, which is called when Persona has some user authentication information, and we need to send it up to our server for validation. That involves an Ajax call (`$.post`), and they're normally quite hard to test, but sinon.js has a helper called fake XMLHttpRequest.

This patches out the native JavaScript `XMLHttpRequest` class, so it's good practice to make sure we restore it afterwards. This gives us a good excuse to learn about QUnit's `setup` and `teardown` methods—they are used in a function called `module`, which acts a bit like a `unittest.TestCase` class, and groups all the tests that follow it together.

Aside on Ajax

If you've never used Ajax before, here is a very brief overview. You may find it useful to read up on it elsewhere before proceeding though.

Ajax stands for "Asynchronous JavaScript and XML", although the XML part is a bit of a misnomer these days, since everyone usually sends text or JSON rather than XML. It's a way of letting your client-side JavaScript code send and receive information via the HTTP protocol (GET and POST requests), but do so "asynchronously", ie, without blocking and without needing to reload the page.

Here we're going to use Ajax requests to send a POST request to our login view, sending it the assertion information from the Persona UI. We'll use the jQuery Ajax convenience functions.

Let's add this "module" after the first test, before the test for `"initialize calls navigator.id.watch"`:

accounts/static/tests/tests.html (ch15l045).

```
var user, token, urls, mockNavigator, requests, xhr; //❶
module("navigator.id.watch tests", {
    setup: function () {
        user = 'current user'; //❷
        token = 'csrf token';
        urls = { login: 'login url', logout: 'logout url' };
        mockNavigator = {
            id: {
                watch: sinon.mock()
            }
        };
        xhr = sinon.useFakeXMLHttpRequest(); //❸
        requests = []; //❹
        xhr.onCreate = function (request) { requests.push(request); }; //❺
    },
    teardown: function () {
        mockNavigator.id.watch.reset(); //❻
        xhr.restore(); //❼
    }
});

test("initialize calls navigator.id.watch", function () {
    [...]
```

❶ We pull out the variables `user`, `token`, `urls`, etc. up to a higher scope, so that they'll be available to all of the tests in the file.

❷ We initialise said variables inside the `setup` function, which, just like a `unittest setUp` function, will run before each test. That includes our `mockNavigator`.

❸ We also invoke Sinon's `useFakeXMLHttpRequest`, which patches out the browser's Ajax capabilities.

❹ ❺ There's one more bit of boilerplate: we tell Sinon to take any Ajax requests and put them into the `requests` array, so that we can inspect them in our tests.

❻ Finally we have the cleanup—we "reset" the mock for the `watch` function in between each test (otherwise calls from one test would show up in others).

❼ And we put the JavaScript `XMLHttpRequest` back to the way we found it.

That lets us rewrite our two tests to be much shorter:

accounts/static/tests/tests.html (ch15l046).

```
test("initialize calls navigator.id.watch", function () {
    Superlists.Accounts.initialize(mockNavigator, user, token, urls);
    equal(mockNavigator.id.watch.calledOnce, true, 'check watch function called');
});


test("watch sees current user", function () {
    Superlists.Accounts.initialize(mockNavigator, user, token, urls);
    var watchCallArgs = mockNavigator.id.watch.firstCall.args[0];
    equal(watchCallArgs.loggedInUser, user, 'check user');
});
```

And they still pass, but their name is neatly prefixed with our module name:

```
4 assertions of 4 passed, 0 failed.

1. initialize binds sign in button to navigator.id.request (0, 2, 2)
2. navigator.id.watch tests: initialize calls navigator.id.watch (0, 1, 1)
3. navigator.id.watch tests: watch sees current user (0, 1, 1)
```

And here's how we test the `onlogin` callback:

accounts/static/tests/tests.html (ch15l047).

```
test("onlogin does ajax post to login url", function () {
    Superlists.Accounts.initialize(mockNavigator, user, token, urls);
    var onloginCallback = mockNavigator.id.watch.firstCall.args[0].onlogin; //❶
    onloginCallback(); //❷
    equal(requests.length, 1, 'check ajax request'); //❸
    equal(requests[0].method, 'POST');
    equal(requests[0].url, urls.login, 'check url');
});

test("onlogin sends assertion with csrf token", function () {
    Superlists.Accounts.initialize(mockNavigator, user, token, urls);
    var onloginCallback = mockNavigator.id.watch.firstCall.args[0].onlogin;
    var assertion = 'browser-id assertion';
    onloginCallback(assertion);
    equal(
        requests[0].requestBody,
        $.param({ assertion: assertion, csrfmiddlewaretoken: token }), //❹
```

```
            'check POST data'
        );
    });
```

❶ The mock we set on the mock navigator's watch function lets us extract the callback function we set as "onlogin."

❷ We can then actually call that function in order to test it.

❸ Sinon's `fakeXMLHttpRequest` server will catch any Ajax requests we make, and put them into the `requests` array. We can then check on things like whether it was a POST and what URL it was sent to.

❹ The actual POST parameters are held in `.requestBody`, but they are URL-encoded (using the &key=val syntax). jQuery's `$.param` function does URL-encoding, so we use that to do our comparison.

And the two tests fail as expected:

```
4. navigator.id.watch tests: onlogin does ajax post to login url (1, 0, 1)
    1. Died on test #1
@file:///workspace/superlists/accounts/static/tests/tests.html:78:
onloginCallback is not a function

5. navigator.id.watch tests: onlogin sends assertion with csrf token (1, 0, 1)
    1. Died on test #1
@file:///workspace/superlists/accounts/static/tests/tests.html:90:
onloginCallback is not a function
```

Another unit-test/code cycle. Here's the failure messages I went through:

```
    1. check ajax request
    Expected: 1

...

    3. check url
    Expected: "login url"

...

    7 assertions of 8 passed, 1 failed.
    1. check POST data
    Expected:
    "assertion=browser-id+assertion&csrfmiddlewaretoken=csrf+token"
    Result: null

...

    1. check POST data
    Expected:
    "assertion=browser-id+assertion&csrfmiddlewaretoken=csrf+token"
    Result: "assertion=browser-id+assertion"

...
```

```
8 assertions of 8 passed, 0 failed.
```

And I ended up with this code:

accounts/static/accounts.js.

```
    navigator.id.watch({
        loggedInUser: user,
        onlogin: function (assertion) {
            $.post(
                urls.login,
                { assertion: assertion, csrfmiddlewaretoken: token }
            );
        }
    });
```

Logout

At the time of writing, the "onlogout" part of the watch API's status was uncertain. It works, but it's not necessary for our purposes. We'll just make it a do-nothing function, as a placeholder. Here's a minimal test for that:

accounts/static/tests/tests.html (ch15l053).

```
test("onlogout is just a placeholder", function () {
    Superlists.Accounts.initialize(mockNavigator, user, token, urls);
    var onlogoutCallback = mockNavigator.id.watch.firstCall.args[0].onlogout;
    equal(typeof onlogoutCallback, "function", "onlogout should be a function");
});
```

And we get quite a simple logout function:

accounts/static/accounts.js (ch15l054).

```
    },
    onlogout: function () {}
});
```

More Nested Callbacks! Testing Asynchronous Code

This is what JavaScript's all about folks! Thankfully, sinon.js really does help. We still need to test that our login `post` methods *also* set some callbacks for things to do *after* the POST request comes back:

```
    .done(function() { window.location.reload(); })
    .fail(function() { navigator.id.logout();});
```

I'm going to skip testing the `window.location.reload`, because it's a bit unnecessarily complicated,[7] and I think we can allow that this will be tested by our Selenium test. We will do a test for the on-fail callback though, just to demonstrate that it is possible:

7. You can't mock out `window.location.reload`, so instead you have to define an (untested) function called `Superlists.Accounts.refreshPage`, and then put a mock on *that* to check that it gets set as the Ajax `.done` callback.

accounts/static/tests/tests.html (ch15l055).

```
test("onlogin post failure should do navigator.id.logout ", function () {
    mockNavigator.id.logout = sinon.mock(); //❶
    Superlists.Accounts.initialize(mockNavigator, user, token, urls);
    var onloginCallback = mockNavigator.id.watch.firstCall.args[0].onlogin;
    var server = sinon.fakeServer.create(); //❷
    server.respondWith([403, {}, "permission denied"]); //❸

    onloginCallback();
    equal(mockNavigator.id.logout.called, false, 'should not logout yet');

    server.respond(); //❹
    equal(mockNavigator.id.logout.called, true, 'should call logout');
});
```

❶ We put a mock on the `navigator.id.logout` function which we're interested in.

❷ We use Sinon's `fakeServer`, which is an abstraction on top of the `fakeXMLHttpRequest` to simulate Ajax server responses.

❸ We set up our fake server to respond with a 403 "permission denied" response, to simulate what will happen for unauthorized users.

❹ We then explicitly tell the fake server to send that response. Only then should we see the logout call.

That gets us to this—a slight change to our spiked code:

accounts/static/accounts.js (ch15l056).

```
    onlogin: function (assertion) {
        $.post(
            urls.login,
            { assertion: assertion, csrfmiddlewaretoken: token }
        ).fail(function () { navigator.id.logout(); });
    },
    onlogout: function () {}
```

Finally we add our `window.location.reload`, just to check it doesn't break any unit tests:

accounts/static/accounts.js (ch15l057).

```
    navigator.id.watch({
        loggedInUser: user,
        onlogin: function (assertion) {
            $.post(
                urls.login,
                { assertion: assertion, csrfmiddlewaretoken: token }
            )
                .done(function () { window.location.reload(); })
                .fail(function () { navigator.id.logout(); });
        },
        onlogout: function () {}
    });
```

Everything's still OK:

```
11 assertions of 11 passed, 0 failed.
```

If those chained `.done` and `.fail` calls are bugging you—they bug me a little—you can rewrite that as, eg:

```
var deferred = $.post(
    urls.login,
    { assertion: assertion, csrfmiddlewaretoken: token }
);
deferred.done(function () { window.location.reload(); })
deferred.fail(function () { navigator.id.logout(); });
```

But async code is always a bit mind-bending. I find it just about readable as it is: "do a post to urls.login with the assertion and csrf token, when it's done, do a window reload, or if it fails, do a navigator.id.logout". You can read up on JavaScript deferreds, aka "promises", here.

We're approaching the moment of truth: will our FTs get any further? First, we adjust our `initialize` call:

lists/templates/base.html.

```
<script>
    /*global $, Superlists, navigator */
    $(document).ready(function () {
        var user = "{{ user.email }}" || null;
        var token = "{{ csrf_token }}";
        var urls = {
            login: "TODO",
            logout: "TODO",
        };
        Superlists.Accounts.initialize(navigator, user, token, urls);
    });
</script>
```

And we run the FT…

```
$ python3 manage.py test functional_tests.test_login
Creating test database for alias 'default'...
Not Found: /favicon.ico
Not Found: /TODO
E
======================================================================
ERROR: test_login_with_persona (functional_tests.test_login.LoginTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/workspace/superlists/functional_tests/test_login.py", line 47, in
test_login_with_persona
    self.wait_for_element_with_id('id_logout')
  File "/workspace/superlists/functional_tests/test_login.py", line 23, in
wait_for_element_with_id
    lambda b: b.find_element_by_id(element_id)
[...]
```

```
selenium.common.exceptions.TimeoutException: Message: ''

 ----------------------------------------------------------------------
Ran 1 test in 28.779s

FAILED (errors=1)
Destroying test database for alias 'default'...
```

Hooray! I mean, I know it failed, but we saw it popping up the Persona dialog and getting through it and everything! Next chapter: the server side.

On Spiking and Mocking with JavaScript

Spiking
: Exploratory coding to find out about a new API, or to explore the feasibility of a new solution. Spiking can be done without tests. It's a good idea to do your spike on a new branch, and go back to master when de-spiking.

Mocking
: We use mocking in unit tests when we have an external dependency that we don't want to actually use in our tests. A mock is used to simulate the third-party API. Whilst it is possible to "roll your own" mocks in JavaScript, a mocking framework like Sinon provides a lot of helpful shortcuts which will make it easier to write (and more importantly, read) your tests.

Unit testing Ajax
: Sinon is a great help here. Manually mocking Ajax methods is a real pain.

CHAPTER 16

Server-Side Authentication and Mocking in Python

Let's crack on with the server side of our new auth system. In this chapter we'll do some more mocking, this time with Python. We'll also find out about how to customise Django's authentication system.

A Look at Our Spiked Login View

At the end of the last chapter, we had a working client side that was trying to send authentication assertions to our server's login view. Let's start by building that view, and then move inwards to build the backend authentication functions.

Here's the spiked version of our login view:

```
def persona_login(request):
    print('login view', file=sys.stderr)
    #user = PersonaAuthenticationBackend().authenticate(request.POST['assertion'])
    user = authenticate(assertion=request.POST['assertion'])  #❶
    if user is not None:
        login(request, user)  #❷
    return redirect('/')
```

❶ `authenticate` is our customised authentication function, which we'll de-spike later. Its job is to take the assertion from the client side and validate it.

❷ `login` is Django's built-in login function. It stores a session object on the server, tied to the user's cookies, so that we can recognise them as being an authenticated user on future requests.

Our `authenticate` function is going to make calls out, over the Internet, to Mozilla's servers. We don't want that to happen in our unit test, so we'll want to mock out `authenticate`.

Mocking in Python

The popular *mock* package was added to the standard library as part of Python 3.3.[1] It provides a magical object called a `Mock`, which is a bit like the Sinon mock objects we saw in the last chapter, only much cooler. Check this out:

```
>>> from unittest.mock import Mock
>>> m = Mock()
>>> m.any_attribute
<Mock name='mock.any_attribute' id='140716305179152'>
>>> m.foo
<Mock name='mock.foo' id='140716297764112'>
>>> 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()
1
```

A mock object would be a pretty neat thing to use to mock out the `authenticate` function, wouldn't it? Here's how you can do that.

Testing Our View by Mocking Out authenticate

(I trust you to set up a tests folder with a dunderinit. Don't forget to delete the default *tests.py*, as well.)

accounts/tests/test_views.py.

```
from django.test import TestCase
from unittest.mock import patch


class LoginViewTest(TestCase):

    @patch('accounts.views.authenticate')  #❶
    def test_calls_authenticate_with_assertion_from_post(
        self, mock_authenticate  #❷
    ):
        mock_authenticate.return_value = None  #❸
        self.client.post('/accounts/login', {'assertion': 'assert this'})
        mock_authenticate.assert_called_once_with(assertion='assert this')  #❹
```

1. If you're using Python 3.2, upgrade! Or if you're stuck with it, `pip3 install mock`, and use `from mock` instead of `from unittest.mock`.

❶ The decorator called `patch` is a bit like the Sinon `mock` function we saw in the last chapter. It lets you specify an object you want to "mock out". In this case we're mocking out the `authenticate` function, which we expect to be using in *accounts/views.py*.

❷ The decorator adds the mock object as an additional argument to the function it's applied to.

❸ We can then configure the mock so that it has certain behaviours. Having `authenticate` return `None` is the simplest, so we set the special `.return_value` attribute. Otherwise it would return another mock, and that would probably confuse our view.

❹ Mocks can make assertions! In this case, they can check whether they were called, and what with.

So what does that give us?

```
$ python3 manage.py test accounts
[...]
AttributeError: <module 'accounts.views' from
'/workspace/superlists/accounts/views.py'> does not have the attribute
'authenticate'
```

We tried to patch something that doesn't exist yet. We need to import `authenticate` into our *views.py*:[2]

accounts/views.py

```
from django.contrib.auth import authenticate
```

Now we get:

```
AssertionError: Expected 'authenticate' to be called once. Called 0 times.
```

That's our expected failure; to implement, we'll have to wire up a URL for our login view:

superlists/urls.py

```
urlpatterns = patterns('',
    url(r'^$', 'lists.views.home_page', name='home'),
    url(r'^lists/', include('lists.urls')),
    url(r'^accounts/', include('accounts.urls')),
    # url(r'^admin/', include(admin.site.urls)),
)
```

accounts/urls.py

```
from django.conf.urls import patterns, url

urlpatterns = patterns('',
```

2. Even though we're going to define our own `authenticate` function, we still import from `django.contrib.auth`. Django will dynamically replace it with our function once we've configured it in *settings.py*. This has the benefit that, if we later switch to a third-party library for our `authenticate` function, our views.py doesn't need to change.

```
    url(r'^login$', 'accounts.views.persona_login', name='persona_login'),
)
```

Will a minimal view do anything?

accounts/views.py.

```
from django.contrib.auth import authenticate

def persona_login():
    pass
```

Yep:

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

And so:

accounts/views.py (ch16l008).

```
def persona_login(request):
    pass
```

Then:

```
ValueError: The view accounts.views.persona_login didn't return an HttpResponse
object. It returned None instead.
```

accounts/views.py (ch16l009).

```
from django.contrib.auth import authenticate
from django.http import HttpResponse

def persona_login(request):
    return HttpResponse()
```

And we're back to:

```
AssertionError: Expected 'authenticate' to be called once. Called 0 times.
```

We try:

accounts/views.py.

```
def persona_login(request):
    authenticate()
    return HttpResponse()
```

And sure enough, we get:

```
AssertionError: Expected call: authenticate(assertion='assert this')
Actual call: authenticate()
```

And then we can fix that too:

accounts/views.py.

```
def persona_login(request):
    authenticate(assertion=request.POST['assertion'])
    return HttpResponse()
```

OK so far. One Python function mocked and tested.

Checking the View Actually Logs the User In

But our `authenticate` view also needs to actually log the user in by calling the Django `auth.login` function, if `authenticate` returns a user. Then it needs to return something other than an empty response—since this is an Ajax view, it doesn't need to return HTML, just an "OK" string will do:

accounts/tests/test_views.py (ch16l011).

```
from django.contrib.auth import get_user_model
from django.test import TestCase
from unittest.mock import patch
User = get_user_model()  #❶


class LoginViewTest(TestCase):
    @patch('accounts.views.authenticate')
    def test_calls_authenticate_with_assertion_from_post(
        [...]

    @patch('accounts.views.authenticate')
    def test_returns_OK_when_user_found(
        self, mock_authenticate
    ):
        user = User.objects.create(email='a@b.com')
        user.backend = ''  # required for auth_login to work
        mock_authenticate.return_value = user
        response = self.client.post('/accounts/login', {'assertion': 'a'})
        self.assertEqual(response.content.decode(), 'OK')
```

❶ I should explain this use of `get_user_model` from `django.contrib.auth`. Its job is to find the project's user model, and it works whether you're using the standard user model or a custom one (like we will be).

That test covers the desired response. Now test that the user actually gets logged in correctly. We can do that by inspecting the Django test client, to see if the session cookie has been set correctly.

Check out the Django docs on authentication at this point.

accounts/tests/test_views.py (ch16l012).

```
from django.contrib.auth import get_user_model, SESSION_KEY
[...]

    @patch('accounts.views.authenticate')
    def test_gets_logged_in_session_if_authenticate_returns_a_user(
        self, mock_authenticate
```

```
):
    user = User.objects.create(email='a@b.com')
    user.backend = ''  # required for auth_login to work
    mock_authenticate.return_value = user
    self.client.post('/accounts/login', {'assertion': 'a'})
    self.assertEqual(self.client.session[SESSION_KEY], user.pk)  #❶


@patch('accounts.views.authenticate')
def test_does_not_get_logged_in_if_authenticate_returns_None(
    self, mock_authenticate
):
    mock_authenticate.return_value = None
    self.client.post('/accounts/login', {'assertion': 'a'})
    self.assertNotIn(SESSION_KEY, self.client.session)  #❷
```

❶ The Django test client keeps track of the session for its user. For the case where the user gets authenticated successfully, we check that their user ID (the primary key, or pk) is associated with their session.

❷ In the case where the user should not be authenticated, the `SESSION_KEY` should not appear in their session.

Django Sessions: How a User's Cookies Tell the Server She Is Authenticated

Being an attempt to explain sessions, cookies, and authentication in Django.

Because HTTP is stateless, servers need a way of recognising different clients with *every single request*. IP addresses can be shared, so the usual solution is to give each client a unique session ID, which it will store in a cookie, and submit with every request. The server will store that ID somewhere (by default, in the database), and then it can recognise each request that comes in as being from a particular client.

If you log in to the site using the dev server, you can actually take a look at your session ID by hand if you like. It's stored under the key `sessionid` by default. See Figure 16-1.

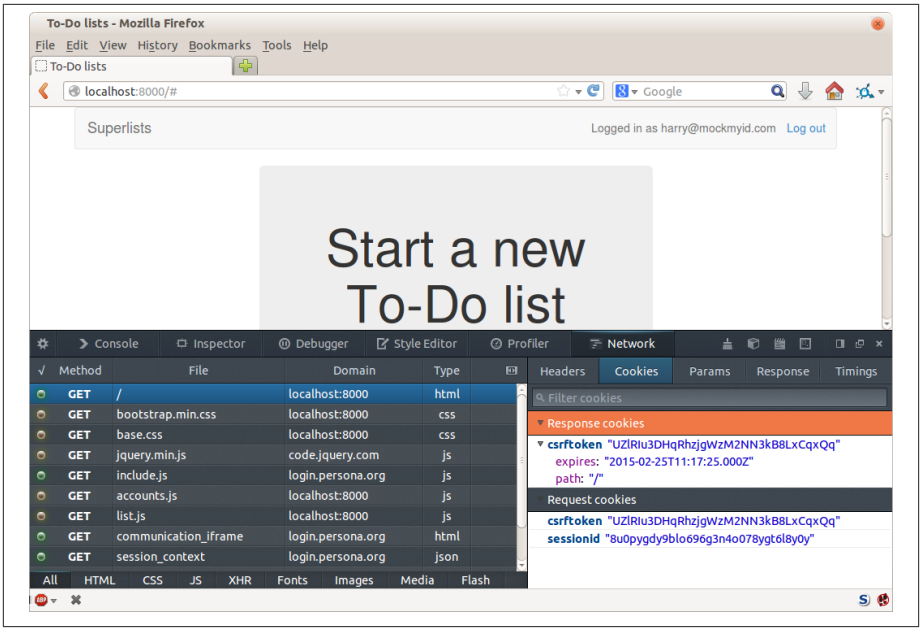


Figure 16-1. Examining the session cookie in the Debug toolbar

These session cookies are set for all visitors to a Django site, whether they're logged in or not.

When we want to recognise a client as being a logged-in and authenticated user, again, rather asking the client to send their username and password with every single request, the server can actually just mark that client's session as being an authenticated session, and associate it with a user ID in its database.

A session is a dictionary-like data structure, and the user ID is stored under the key given by `django.contrib.auth.SESSION_KEY`. You can check this out in a `manage.py` console if you like:

```
$ python3 manage.py shell
[...]
In [1]: from django.contrib.sessions.models import Session

# substitute your session id from your browser cookie here
In [2]: session = Session.objects.get(
    session_key="8u0pygdy9blo696g3n4o078ygt6l8y0y"
)

In [3]: print(session.get_decoded())
{'_auth_user_id': 'harry@mockmyid.com', '_auth_user_backend':
'accounts.authentication.PersonaAuthenticationBackend'}
```

You can also store any other information you like on a user's session, as a way of temporarily keeping track of some state. This works for non-logged-in users too. Just use

`request.session` inside any view, and it works as a dict. There's more information in the Django docs on sessions.

That gives us two failures:

```
$ python3 manage.py test accounts
[...]
    self.assertEqual(self.client.session[SESSION_KEY], user.pk)
KeyError: '_auth_user_id'

[...]
AssertionError: '' != 'OK'
+ OK
```

The Django function that takes care of logging in a user, by marking their session, is available at `django.contrib.auth.login`. So we go through another couple of TDD cycles, until:

accounts/views.py.

```
from django.contrib.auth import authenticate, login
from django.http import HttpResponse

def persona_login(request):
    user = authenticate(assertion=request.POST['assertion'])
    if user:
        login(request, user)
    return HttpResponse('OK')
```

```
...

 OK
```

We now have a working login view.

Testing Login with Mocks

An alternative way of testing that the Django login function was called correctly would be to mock it out too:

accounts/tests/test_views.py.

```
from django.http import HttpRequest
from accounts.views import persona_login
[...]

    @patch('accounts.views.login')
    @patch('accounts.views.authenticate')
    def test_calls_auth_login_if_authenticate_returns_a_user(
        self, mock_authenticate, mock_login
    ):
        request = HttpRequest()
        request.POST['assertion'] = 'asserted'
```

```
            mock_user = mock_authenticate.return_value
            login(request)
            mock_login.assert_called_once_with(request, mock_user)
```

The upside of this version of the test is that it doesn't need to rely on the magic of the Django test client, and it doesn't need to know anything about how Django sessions work—all you need to know is the name of the function you're supposed to call.

Its downside is that it is very much testing implementation, rather than testing behaviour—it's tightly coupled to the particular name of the Django login function and its API.

De-spiking Our Custom Authentication Backend: Mocking Out an Internet Request

Our custom authentication backend is next. Here's how it looked in the spike:

```
class PersonaAuthenticationBackend(object):

    def authenticate(self, assertion):
        # Send the assertion to Mozilla's verifier service.
        data = {'assertion': assertion, 'audience': 'localhost'}
        print('sending to mozilla', data, file=sys.stderr)
        resp = requests.post('https://verifier.login.persona.org/verify', data=data)
        print('got', resp.content, file=sys.stderr)

        # Did the verifier respond?
        if resp.ok:
            # Parse the response
            verification_data = resp.json()

            # Check if the assertion was valid
            if verification_data['status'] == 'okay':
                email = verification_data['email']
                try:
                    return self.get_user(email)
                except ListUser.DoesNotExist:
                    return ListUser.objects.create(email=email)


    def get_user(self, email):
        return ListUser.objects.get(email=email)
```

Decoding this:

- We take an assertion and send it off to Mozilla using `requests.post`.
- We check its response code (`resp.ok`), and then check for a `status=okay` in the response JSON.
- We then extract an email address, and either find an existing user with that address, or create a new one.

1 if = 1 More Test

A rule of thumb for these sorts of tests: any `if` means an extra test, and any `try/except` means an extra test, so this should be about four tests. Let's start with one:

accounts/tests/test_authentication.py.

```
from unittest.mock import patch
from django.test import TestCase

from accounts.authentication import (
    PERSONA_VERIFY_URL, DOMAIN, PersonaAuthenticationBackend
)

class AuthenticateTest(TestCase):

    @patch('accounts.authentication.requests.post')
    def test_sends_assertion_to_mozilla_with_domain(self, mock_post):
        backend = PersonaAuthenticationBackend()
        backend.authenticate('an assertion')
        mock_post.assert_called_once_with(
            PERSONA_VERIFY_URL,
            data={'assertion': 'an assertion', 'audience': DOMAIN}
        )
```

In *authenticate.py* we'll just have a few placeholders:

accounts/authentication.py.

```
import requests

PERSONA_VERIFY_URL = 'https://verifier.login.persona.org/verify'
DOMAIN = 'localhost'


class PersonaAuthenticationBackend(object):

    def authenticate(self, assertion):
        pass
```

At this point we'll need to:

```
(virtualenv)$ pip install requests
```

Don't forget to add `requests` to *requirements.txt* too, or the next deploy won't work…

Then let's see how the tests get on!

```
$ python3 manage.py test accounts
[...]
AssertionError: Expected 'post' to be called once. Called 0 times.
```

And we can get that to passing in three steps (make sure the Goat sees you doing each one individually!):

accounts/authentication.py.

```
    def authenticate(self, assertion):
        requests.post(
            PERSONA_VERIFY_URL,
            data={'assertion': assertion, 'audience': DOMAIN}
        )
```

Grand:

```
$ python3 manage.py test accounts
[...]

Ran 5 tests in 0.023s

OK
```

Next let's check that `authenticate` should return `None` if it sees an error from the request:

accounts/tests/test_authentication.py (ch16l020).

```
    @patch('accounts.authentication.requests.post')
    def test_returns_none_if_response_errors(self, mock_post):
        mock_post.return_value.ok = False
        backend = PersonaAuthenticationBackend()

        user = backend.authenticate('an assertion')
        self.assertIsNone(user)
```

And that passes straight away—we currently return `None` in all cases!

Patching at the Class Level

Next we want to check that the response JSON has `status=okay`. Adding this test would involve a bit of duplication—let's apply the "three strikes" rule:

accounts/tests/test_authentication.py (ch16l021).

```
@patch('accounts.authentication.requests.post')  #❶
class AuthenticateTest(TestCase):

    def setUp(self):
        self.backend = PersonaAuthenticationBackend()  #❷


    def test_sends_assertion_to_mozilla_with_domain(self, mock_post):
        self.backend.authenticate('an assertion')
        mock_post.assert_called_once_with(
            PERSONA_VERIFY_URL,
            data={'assertion': 'an assertion', 'audience': DOMAIN}
        )


    def test_returns_none_if_response_errors(self, mock_post):
```

```
        mock_post.return_value.ok = False  #❸
        user = self.backend.authenticate('an assertion')
        self.assertIsNone(user)


    def test_returns_none_if_status_not_okay(self, mock_post):
        mock_post.return_value.json.return_value = {'status': 'not okay!'}  #❹
        user = self.backend.authenticate('an assertion')
        self.assertIsNone(user)
```

❶ You can apply a `patch` at the class level as well, and that has the effect that every test method in the class will have the patch applied, and the mock injected.

❷ We can now use the `setUp` function to prepare any useful variables which we're going to use in all of our tests.

❸ ❹ Now each test is only adjusting the setup variables *it* needs, rather than setting up a load of duplicated boilerplate—it's more readable.

And that's all very well, but everything still passes!

```
OK
```

Time to test for the positive case where `authenticate` should return a user object. We expect this to fail:

accounts/tests/test_authentication.py (ch16l022-1).

```
from django.contrib.auth import get_user_model
User = get_user_model()
[...]

    def test_finds_existing_user_with_email(self, mock_post):
        mock_post.return_value.json.return_value = {'status': 'okay', 'email': 'a@b.com'}
        actual_user = User.objects.create(email='a@b.com')
        found_user = self.backend.authenticate('an assertion')
        self.assertEqual(found_user, actual_user)
```

Indeed, a fail:

```
AssertionError: None != <User: >
```

Let's code. We'll start with a "cheating" implementation, where we just get the first user we find in the database:

accounts/authentication.py (ch16l023).

```
import requests
from django.contrib.auth import get_user_model
User = get_user_model()
[...]

    def authenticate(self, assertion):
        requests.post(
            PERSONA_VERIFY_URL,
            data={'assertion': assertion, 'audience': DOMAIN}
        )
        return User.objects.first()
```

That gets our new test passing, but still, none of the other tests are failing:

```
$ python3 manage.py test accounts
[...]

Ran 8 tests in 0.030s

OK
```

They're passing because `objects.first()` returns `None` if there are no users in the database. Let's make our other cases more realistic, by making sure there's always at least one user in the database for all our tests:

accounts/tests/test_authentication.py (ch16l022-2).

```
    def setUp(self):
        self.backend = PersonaAuthenticationBackend()
        user = User(email='other@user.com')
        user.username = 'otheruser'  #❶
        user.save()
```

❶ By default, Django's users have a `username` attribute, which has to be unique, so this value is just a placeholder to allow us to create multiple users. Later on, we'll get rid of usernames in favour of using emails as the primary key.

That gives us three failures:

```
FAIL: test_finds_existing_user_with_email
AssertionError: <User: otheruser> != <User: >
[...]
FAIL: test_returns_none_if_response_errors
AssertionError: <User: otheruser> is not None
[...]
FAIL: test_returns_none_if_status_not_okay
AssertionError: <User: otheruser> is not None
```

Let's start building our guards for cases where authentication should fail—if the response errors, or if the status is not `okay`. Suppose we start with this:

accounts/authentication.py (ch16l024-1).

```
def authenticate(self, assertion):
    response = requests.post(
        PERSONA_VERIFY_URL,
        data={'assertion': assertion, 'audience': DOMAIN}
    )
    if response.json()['status'] == 'okay':
        return User.objects.first()
```

That actually fixes two of the tests, slightly surprisingly:

```
AssertionError: <User: otheruser> != <User: >

FAILED (failures=1)
```

Let's get the final test passing by retrieving the right user, and then we'll have a look at that surprise pass:

accounts/authentication.py (ch16l024-2).

```
        if response.json()['status'] == 'okay':
            return User.objects.get(email=response.json()['email'])
```

```
...

    OK
```

Beware of Mocks in Boolean Comparisons

So how come our `test_returns_none_if_response_errors` isn't failing?

Because we've mocked out `requests.post`, the `response` is a Mock object, which as you remember, returns all attributes and properties as more Mocks.[3] So, when we do:

accounts/authentication.py.

```
        if response.json()['status'] == 'okay':
```

`response` is actually a mock, `response.json()` is a mock, and `response.json()['status']` is a mock too! We end up comparing a mock with the string "okay", which evaluates to False, and so we return None by default. Let's make our test more explicit, by saying that the response JSON will be an empty dict:

accounts/tests/test_authentication.py (ch16l025).

```
        def test_returns_none_if_response_errors(self, mock_post):
            mock_post.return_value.ok = False
            mock_post.return_value.json.return_value = {}
            user = self.backend.authenticate('an assertion')
            self.assertIsNone(user)
```

That gives:

```
        if response.json()['status'] == 'okay':
    KeyError: 'status'
```

And we can fix it like this:

accounts/authentication.py (ch16l026).

```
        if response.ok and response.json()['status'] == 'okay':
            return User.objects.get(email=response.json()['email'])
```

```
...

    OK
```

Great! Our `authenticate` function is now working the way we want it to.

3. Actually, this is only happening because we're using the `patch` decorator, which returns a `MagicMock`, an even mockier version of `mock` that can also behave like a dictionary. More info in the docs.

Creating a User if Necessary

Next we should check that, if our `authenticate` function has a valid assertion from Persona, but we don't have a user record for that person in our database, we should create one. Here's the test for that:

accounts/tests/test_authentication.py (ch16l027).

```
def test_creates_new_user_if_necessary_for_valid_assertion(self, mock_post):
    mock_post.return_value.json.return_value = {'status': 'okay', 'email': 'a@b.com'}
    found_user = self.backend.authenticate('an assertion')
    new_user = User.objects.get(email='a@b.com')
    self.assertEqual(found_user, new_user)
```

That fails in our application code when we try find an existing user with that email:

```
    return User.objects.get(email=response.json()['email'])
django.contrib.auth.models.DoesNotExist: User matching query does not exist.
```

So we add a `try/except`, returning an "empty" user at first:

accounts/authentication.py (ch16l028).

```
    if response.ok and response.json()['status'] == 'okay':
        try:
            return User.objects.get(email=response.json()['email'])
        except User.DoesNotExist:
            return User.objects.create()
```

And that fails, but this time it fails when the *test* tries to find the new user by email:

```
    new_user = User.objects.get(email='a@b.com')
django.contrib.auth.models.DoesNotExist: User matching query does not exist.
```

And so we fix it by assigning the correct email addresss:

accounts/authentication.py (ch16l029).

```
    if response.ok and response.json()['status'] == 'okay':
        email = response.json()['email']
        try:
            return User.objects.get(email=email)
        except User.DoesNotExist:
            return User.objects.create(email=email)
```

That gets us to passing tests:

```
$ python3 manage.py test accounts
[...]
Ran 9 tests in 0.019s
OK
```

The get_user Method

The next thing we have to build is a `get_user` method for our authentication backend. This method's job is to retrieve a user based on their email address, or to return `None` if it can't find one. (That last wasn't well documented at the time of writing, but that is the interface we have to comply with. See the source for details.)

Here's a couple of tests for those two requirements:

accounts/tests/test_authentication.py (ch16l030).

```
class GetUserTest(TestCase):

    def test_gets_user_by_email(self):
        backend = PersonaAuthenticationBackend()
        other_user = User(email='other@user.com')
        other_user.username = 'otheruser'
        other_user.save()
        desired_user = User.objects.create(email='a@b.com')
        found_user = backend.get_user('a@b.com')
        self.assertEqual(found_user, desired_user)


    def test_returns_none_if_no_user_with_that_email(self):
        backend = PersonaAuthenticationBackend()
        self.assertIsNone(
            backend.get_user('a@b.com')
        )
```

Here's our first failure:

```
AttributeError: 'PersonaAuthenticationBackend' object has no attribute
'get_user'
```

Let's create a placeholder one then:

accounts/authentication.py (ch16l031).

```
class PersonaAuthenticationBackend(object):

    def authenticate(self, assertion):
        [...]

    def get_user(self):
        pass
```

Now we get:

```
TypeError: get_user() takes 1 positional argument but 2 were given
```

So:

accounts/authentication.py (ch16l032).

```
    def get_user(self, email):
        pass
```

Next:

```
        self.assertEqual(found_user, desired_user)
AssertionError: None != <User: >
```

And (step by step, just to see if our test fails the way we think it will):

accounts/authentication.py (ch16l033).

```
    def get_user(self, email):
        return User.objects.first()
```

That gets us past the first assertion, and onto

```
        self.assertEqual(found_user, desired_user)
    AssertionError: <User: otheruser> != <User: >
```

And so we call `get` with the email as an argument:

accounts/authentication.py (ch16l034).

```
    def get_user(self, email):
        return User.objects.get(email=email)
```

That gets us to passing tests:

Now our test for the None case fails:

```
ERROR: test_returns_none_if_no_user_with_that_email
[...]
django.contrib.auth.models.DoesNotExist: User matching query does not exist.
```

Which prompts us to finish the method like this:

accounts/authentication.py (ch16l035).

```
    def get_user(self, email):
        try:
            return User.objects.get(email=email)
        except User.DoesNotExist:
            return None #❶
```

❶ You could just use `pass` here, and the function would return `None` by default. However, because we specifically need the function to return `None`, explicit is better than implicit here.

That gets us to passing tests:

```
OK
```

And we have a working authentication backend!

```
$ python3 manage.py test accounts
[...]
Ran 11 tests in 0.020s
OK
```

Now we can define our custom user model.

A Minimal Custom User Model

Django's built-in user model makes all sorts of assumptions about what information you want to track about users, from explicitly recording first name and last name, to forcing you to use a username. I'm a great believer in not storing information about users unless you absolutely must, so a user model that records an email address and nothing else sounds good to me!

accounts/tests/test_models.py.

```
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
```

That gives us an expected failure:

```
django.core.exceptions.ValidationError: {'username': ['This field cannot be
blank.'], 'password': ['This field cannot be blank.']}
```

Password? Username? Bah! How about this?

accounts/models.py.

```
from django.db import models

class User(models.Model):
    email = models.EmailField()
```

And we wire it up inside *settings.py* using a variable called `AUTH_USER_MODEL`. While we're at it, we'll add our new authentication backend too:

superlists/settings.py (ch16l039).

```
AUTH_USER_MODEL = 'accounts.User'
AUTHENTICATION_BACKENDS = (
    'accounts.authentication.PersonaAuthenticationBackend',
)
```

Now Django tells us off because it wants a couple of bits of metadata on any custom user model:

```
AttributeError: type object 'User' has no attribute 'REQUIRED_FIELDS'
```

Sigh. Come on, Django, it's only got one field, you should be able to figure out the answers to these questions for yourself. Here you go:

accounts/models.py.

```
class User(models.Model):
    email = models.EmailField()
    REQUIRED_FIELDS = ()
```

Next silly question?[4]

```
AttributeError: type object 'User' has no attribute 'USERNAME_FIELD'
```

So:

4. You might ask, if I think Django is so silly, why don't I submit a pull request to fix it? Should be quite a simple fix. Well, I promise I will, as soon as I've finished writing the book. For now, snarky comments will have to suffice.

accounts/models.py.

```
class User(models.Model):
    email = models.EmailField()
    REQUIRED_FIELDS = ()
    USERNAME_FIELD = 'email'
```

The next error is a database error:

```
django.db.utils.OperationalError: no such table: accounts_user
```

That prompts us, as usual, to do a migration:

```
$ python3 manage.py makemigrations
System check identified some issues:

WARNINGS:
accounts.User: (auth.W004) 'User.email' is named as the 'USERNAME_FIELD', but
it is not unique.
        HINT: Ensure that your authentication backend(s) can handle non-unique
usernames.
Migrations for 'accounts':
  0001_initial.py:
    - Create model User
```

Let's hold that thought, and see if we can get the tests passing again.

A Slight Disappointment

Meanwhile, we have a couple of weird unexpected failures:

```
$ python3 manage.py test accounts
[...]
ERROR: test_gets_logged_in_session_if_authenticate_returns_a_user
[...]
ERROR: test_returns_OK_when_user_found
[...]
    user.save(update_fields=['last_login'])
[...]
ValueError: The following fields do not exist in this model or are m2m fields:
last_login
```

It looks like Django is going to insist on us having a `last_login` field on our user model too. Oh well. My pristine, single-field user model is despoiled. I still love it though.

accounts/models.py.

```
from django.db import models
from django.utils import timezone

class User(models.Model):
    email = models.EmailField()
    last_login = models.DateTimeField(default=timezone.now)
    REQUIRED_FIELDS = ()
    USERNAME_FIELD = 'email'
```

We get another database error, so let's clear down the migration and re-create it:

```
$ rm accounts/migrations/0001_initial.py
$ python3 manage.py makemigrations
System check identified some issues:
[...]
Migrations for 'accounts':
  0001_initial.py:
    - Create model User
```

That gets the tests passing, although they are still giving us some warnings:

```
$ python3 manage.py test accounts
[...]
System check identified some issues:

WARNINGS:
accounts.User: (auth.W004) 'User.email' is named as the 'USERNAME_FIELD', but
it is not unique.
[...]

Ran 12 tests in 0.041s

OK
```

Tests as Documentation

Let's go all the way and make the email field into the primary key, and thus implicitly remove the auto-generated `id` column.

Although that warning is probably enough of a justification to go ahead and make the change, it would be better to have a specific test:

accounts/tests/test_models.py (ch16l043).

```
    def test_email_is_primary_key(self):
        user = User()
        self.assertFalse(hasattr(user, 'id'))
```

It'll help us remember if we ever come back and look at the code again in future.

```
        self.assertFalse(hasattr(user, 'id'))
AssertionError: True is not false
```

Your tests can be are a form of documentation for your code—they express what your requirements are of a particular class or function. Sometimes, if you forget why you've done something a particular way, going back and looking at the tests will give you the answer. That's why it's important to give your tests explicit, verbose method names.

And here's the implementation (feel free to check what happens with `unique=True` first):

accounts/models.py (ch16l044).

```
    email = models.EmailField(primary_key=True)
```

That works:

```
$ python3 manage.py test accounts
[...]
Ran 13 tests in 0.021s
OK
```

One final cleanup of migrations to make sure we've got everything there:

```
$ rm accounts/migrations/0001_initial.py
$ python3 manage.py makemigrations
Migrations for 'accounts':
  0001_initial.py:
    - Create model User
```

No warnings now!

Users Are Authenticated

Our user model needs one last property before it's complete: standard Django users have an API which includes several methods, most of which we won't need, but there is one that will come in useful: `.is_authenticated()`:

accounts/tests/test_models.py (ch16l045).

```
    def test_is_authenticated(self):
        user = User()
        self.assertTrue(user.is_authenticated())
```

Which gives:

```
AttributeError: 'User' object has no attribute 'is_authenticated'
```

And so, the ultra-simple:

accounts/models.py.

```
class User(models.Model):
    email = models.EmailField(primary_key=True)
    last_login = models.DateTimeField(default=timezone.now)
    REQUIRED_FIELDS = ()
    USERNAME_FIELD = 'email'

    def is_authenticated(self):
        return True
```

And that works:

```
$ python3 manage.py test accounts
[...]
Ran 14 tests in 0.021s
OK
```

The Moment of Truth: Will the FT Pass?

I think we're just about ready to try our functional test! Let's just wire up our base template. Firstly, it needs to show a different message for logged-in and non-logged-in users:

lists/templates/base.html.

```
<nav class="navbar navbar-default" role="navigation">
    <a class="navbar-brand" href="/">Superlists</a>
    {% if user.email %}
        <a class="btn navbar-btn navbar-right" id="id_logout" href="#">Log out</a>
        <span class="navbar-text navbar-right">Logged in as {{ user.email }}</span>
    {% else %}
        <a class="btn navbar-btn navbar-right" id="id_login" href="#">Sign in</a>
    {% endif %}
</nav>
```

Lovely. Then we wire up our various context variables for the call to `initialize`:

lists/templates/base.html.

```
<script>
    /*global $, Superlists, navigator */
    $(document).ready(function () {
        var user = "{{ user.email }}" || null;
        var token = "{{ csrf_token }}";
        var urls = {
            login: "{% url 'persona_login' %}",
            logout: "TODO",
        };
        Superlists.Accounts.initialize(navigator, user, token, urls);
    });
</script>
```

So how does our FT get along?

```
$ python3 manage.py test functional_tests.test_login
Creating test database for alias 'default'...
[...]
Ran 1 test in 26.382s

OK
```

Woohoo!

I've been waiting to do a commit up until this moment, just to make sure everything works. At this point, you could make a series of separate commits—one for the login view, one for the auth backend, one for the user model, one for wiring up the template. Or you could decide that, since they're all interrelated, and none will work without the others, you may as well just have one big commit:

```
$ git status
$ git add .
$ git diff --staged
$ git commit -am "Custom Persona auth backend + custom user model"
```

Finishing Off Our FT, Testing Logout

We'll extend our FT to check that the logged-in status persists, ie it's not just something we set in JavaScript on the client side, but the server knows about it too and will maintain the logged-in state if she refreshes the page. We'll also test that she can log out.

I started off writing code a bit like this:

functional_tests/test_login.py.

```
        # Refreshing the page, she sees it's a real session login,
        # not just a one-off for that page
        self.browser.refresh()
        self.wait_for_element_with_id('id_logout')
        navbar = self.browser.find_element_by_css_selector('.navbar')
        self.assertIn('edith@mockmyid.com', navbar.text)
```

And, after four repetitions of very similar code, a couple of helper functions suggested themselves:

functional_tests/test_login.py (ch16l050).

```
    def wait_to_be_logged_in(self):
        self.wait_for_element_with_id('id_logout')
        navbar = self.browser.find_element_by_css_selector('.navbar')
        self.assertIn('edith@mockmyid.com', navbar.text)

    def wait_to_be_logged_out(self):
        self.wait_for_element_with_id('id_login')
        navbar = self.browser.find_element_by_css_selector('.navbar')
        self.assertNotIn('edith@mockmyid.com', navbar.text)
```

And I extended the FT like this:

functional_tests/test_login.py (ch16l049).

```
        [...]
        # The Persona window closes
        self.switch_to_new_window('To-Do')

        # She can see that she is logged in
        self.wait_to_be_logged_in()

        # Refreshing the page, she sees it's a real session login,
        # not just a one-off for that page
        self.browser.refresh()
        self.wait_to_be_logged_in()

        # Terrified of this new feature, she reflexively clicks "logout"
        self.browser.find_element_by_id('id_logout').click()
        self.wait_to_be_logged_out()

        # The "logged out" status also persists after a refresh
        self.browser.refresh()
        self.wait_to_be_logged_out()
```

I also found that improving the failure message in the `wait_for_element_with_id` function helped to see what was going on:

functional_tests/test_login.py.

```
    def wait_for_element_with_id(self, element_id):
        WebDriverWait(self.browser, timeout=30).until(
            lambda b: b.find_element_by_id(element_id),
            'Could not find element with id {}. Page text was {}'.format(
                element_id, self.browser.find_element_by_tag_name('body').text
            )
        )
```

With that, we can see that the test is failing because the logout button doesn't work:

```
$ python3 manage.py test functional_tests.test_login
  File "/workspace/superlists/functional_tests/test_login.py", line 36, in
wait_to_be_logged_out
[...]
selenium.common.exceptions.TimeoutException: Message: 'Could not find element
with id id_login. Page text was Superlists\nLog out\nLogged in as
edith@mockmyid.com\nStart a new To-Do list'
```

Implementing a logout button is actually very simple: we can use Django's built-in logout view, which clears down the user's session and redirects them to a page of our choice:

accounts/urls.py.

```
urlpatterns = patterns('',
    url(r'^login$', 'accounts.views.persona_login', name='persona_login'),
    url(r'^logout$', 'django.contrib.auth.views.logout', {'next_page': '/'}, name='logout'),
)
```

And in *base.html*, we just make the logout into a normal URL link:

lists/templates/base.html.

```
<a class="btn navbar-btn navbar-right" id="id_logout" href="{% url 'logout' %}">Log out</a>
```

And that gets us a fully passing FT—indeed, a fully passing test suite:

```
$ python3 manage.py test functional_tests.test_login
[...]
OK
$ python3 manage.py test
[...]
Ran 54 tests in 78.124s

OK
```

In the next chapter, we'll start trying to put our login system to good use. In the meantime, do a commit, and enjoy this recap:

On Mocking in Python

The Mock library
: Michael Foord (who used to work for the company that spawned PythonAnywhere, just before I joined) wrote the excellent "Mock" library that's now been integrated into the standard library of Python 3. It contains most everything you might need for mocking in Python.

The patch decorator
: `unittest.mock` provides a function called `patch`, which can be used to "mock out" any object from the module you're testing. It's commonly used as a decorator on a test method, or even at the class level, where it's applied to all the test methods of that class.

Mocks are truthy and can mask errors
: Be aware that mocking things out can cause counterintuitive behaviour in `if` statements. Mocks are truthy, and they can also mask errors, because they have all attributes and methods.

Too many mocks are a code smell
: Overly mocky tests end up very tightly coupled to their implementation. Sometimes this is unavoidable. But, in general, try to find ways of organising your code so that you don't need too many mocks.

CHAPTER 17

Test Fixtures, Logging, and Server-Side Debugging

Now that we have a functional authentication system, we want to use it to identify users, and be able to show them all the lists they have created.

To do that, we're going to have to write FTs that have a logged-in user. Rather than making each test go through the (time-consuming) Persona dialog, we want to be able to skip that part.

This is about separation of concerns. Functional tests aren't like unit tests, in that they don't usually have a single assertion. But, conceptually, they should be testing a single thing. There's no need for every single FT to test the login/logout mechanisms. If we can figure out a way to "cheat" and skip that part, we'll spend less time waiting for duplicated test paths.

Don't overdo de-duplication in FTs. One of the benefits of an FT is that it can catch strange and unpredictable interactions between different parts of your application.

Skipping the Login Process by Pre-creating a Session

It's quite common for a user to return to a site and still have a cookie, which means they are "pre-authenticated", so this isn't an unrealistic cheat at all. Here's how you can set it up:

functional_tests/test_my_lists.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 .base import FunctionalTest


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()
        ## to set a cookie we need to first visit the domain.
        ## 404 pages load the quickest!
        self.browser.get(self.server_url + "/404_no_such_url/")
        self.browser.add_cookie(dict(
            name=settings.SESSION_COOKIE_NAME,
            value=session.session_key, #❷
            path='/',
        ))
```

❶ We create a session object in the database. The session key is the primary key of the user object (which is actually their email address).

❷ We then add a cookie to the browser that matches the session on the server—on our next visit to the site, the server should recognise us as a logged-in user.

Note that, as it is, this will only work because we're using `LiveServerTestCase`, so the `User` and `Session` objects we create will end up in the same database as the test server. Later we'll need to modify it so that it works against the database on the staging server too.

JSON Test Fixtures Considered Harmful

When we pre-populate the database with test data, as we've done here with the `User` object and its associated `Session` object, what we're doing is setting up a "test fixture".

Django comes with built-in support for saving database objects as JSON (using the `manage.py dumpdata`), and automatically loading them in your test runs using the `fixtures` class attribute on `TestCase`.

More and more people are starting to say: don't use JSON fixtures. They're a nightmare to maintain when your model changes. Instead, if you can, load data directly using the Django ORM, or look into a tool like `factory_boy`.

Checking It Works

To check it works, it would be good to use the `wait_to_be_logged_in` function we defined in our last test. To access it from a different test, we'll need to pull it up into `FunctionalTest`, as well as a couple of other methods. We'll also tweak them slightly so that they can take an arbitrary email address as a parameter:

functional_tests/base.py (ch17l002-2).

```
from selenium.webdriver.support.ui import WebDriverWait
[...]

class FunctionalTest(StaticLiveServerCase):
    [...]

    def wait_for_element_with_id(self, element_id):
        [...]


    def wait_to_be_logged_in(self, email):
        self.wait_for_element_with_id('id_logout')
        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_element_with_id('id_login')
        navbar = self.browser.find_element_by_css_selector('.navbar')
        self.assertNotIn(email, navbar.text)
```

That means a small tweak in *test_login.py*:

functional_tests/test_login.py (ch17l003).

```
TEST_EMAIL = 'edith@mockmyid.com'
[...]

    def test_login_with_persona(self):
        [...]

        self.browser.find_element_by_id(
            'authentication_email'
        ).send_keys(TEST_EMAIL)
        self.browser.find_element_by_tag_name('button').click()

        [...]

        # She can see that she is logged in
        self.wait_to_be_logged_in(email=TEST_EMAIL)
        [...]
        self.wait_to_be_logged_in(email=TEST_EMAIL)
        [...]
        self.wait_to_be_logged_out(email=TEST_EMAIL)
        [...]
        self.wait_to_be_logged_out(email=TEST_EMAIL)
```

Just to check we haven't broken anything, we rerun the login test:

```
$ python3 manage.py test functional_tests.test_login
[...]
OK
```

And now we can write a placeholder for the "My Lists" test, to see if our pre-authenticated session creator really does work:

functional_tests/test_my_lists.py (ch17l004).

```
    def test_logged_in_users_lists_are_saved_as_my_lists(self):
        email = 'edith@example.com'

        self.browser.get(self.server_url)
        self.wait_to_be_logged_out(email)

        # Edith is a logged-in user
        self.create_pre_authenticated_session(email)

        self.browser.get(self.server_url)
        self.wait_to_be_logged_in(email)
```

That gets us:

```
$ python3 manage.py test functional_tests.test_my_lists
[...]
OK
```

That's a good place for a commit:

```
$ git add functional_tests
$ git commit -m"placeholder test_my_lists and move login checkers into base"
```

The Proof Is in the Pudding: Using Staging to Catch Final Bugs

That's all very well for running the FTs locally, but how would it work against the staging server? Let's try and deploy our site. Along the way we'll catch an unexpected bug (that's what staging is for!), and then we'll have to figure out a way of managing the database on the test server.

```
$ cd deploy_tools
$ fab deploy --host=superlists-staging.ottg.eu
[...]
```

And restart Gunicorn…

```
elspeth@server: sudo restart gunicorn-superlists-staging.ottg.eu
```

Here's what happens when we run the functional tests:

```
$ python3 manage.py test functional_tests \
--liveserver=superlists-staging.ottg.eu

======================================================================
ERROR: test_login_with_persona (functional_tests.test_login.LoginTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/worskpace/functional_tests/test_login.py", line 50, in
test_login_with_persona
[...]
    self.wait_for_element_with_id('id_logout')
[...]
selenium.common.exceptions.TimeoutException: Message: 'Could not find element
with id id_logout. Page text was Superlists\nSign in\nStart a new To-Do list'

======================================================================
ERROR: test_logged_in_users_lists_are_saved_as_my_lists
(functional_tests.test_my_lists.MyListsTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/worskpace/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)
[...]
selenium.common.exceptions.TimeoutException: Message: 'Could not find element
with id id_logout. Page text was Superlists\nSign in\nStart a new To-Do list'
```

We can't log in—either with the real Persona or with our pre-authenticated session. There's some kind of bug.

I had considered just going back and fixing this in the previous chapter, and pretending it never happened, but I think leaving it illustrates the point of running tests against a staging environment. It would have been pretty embarrassing if we'd deployed this bug straight to our live site.

Aside from that, we'll get to practice a bit of server-side debugging.

Setting Up Logging

In order to track this bug down, we have to set up Gunicorn to do some logging. Adjust the Gunicorn config on the server, using `vi` or `nano`:

server: /etc/init/gunicorn-superlists-staging.ottg.eu.conf.

```
[...]
exec ../virtualenv/bin/gunicorn \
    --bind unix:/tmp/superlists-staging.ottg.eu.socket \
    --access-logfile ../access.log \
    --error-logfile ../error.log \
    superlists.wsgi:application
```

That will put an access log and error log into the *~/sites/$SITENAME* folder. Then we add some debug calls in our `authenticate` function (again, we can do this directly on the server):

accounts/authentication.py.

```
    def authenticate(self, assertion):
        logging.warning('entering authenticate function')
        response = requests.post(
            PERSONA_VERIFY_URL,
            data={'assertion': assertion, 'audience': settings.DOMAIN}
        )
        logging.warning('got response from persona')
        logging.warning(response.content.decode())
        [...]
```

Using the "root" logger like this (`logging.warning`) isn't generally a good idea. We'll set up a more robust logging configuration at the end of the chapter.

You should also make sure your *settings.py* still contains the `LOGGING` settings which will actually send stuff to the 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'},
}
```

We restart Gunicorn again, and then either rerun the FT, or just try to log in manually. While that happens, we can watch the logs on the server with a:

```
elspeth@server: $ tail -f error.log  # assumes we are in ~/sites/$SITENAME folder
[...]
WARNING:root:{"status":"failure","reason":"audience mismatch: domain mismatch"}
```

You may even find the page gets stuck in a "redirect loop", as Persona tries to resubmit the assertion again and again.

It turns out it's because I overlooked an important part of the Persona system, which is that authentications are only valid for particular domains. We've left the domain hard-coded as "localhost" in *accounts/authentication.py*:

accounts/authentication.py.

```
PERSONA_VERIFY_URL = 'https://verifier.login.persona.org/verify'
DOMAIN = 'localhost'
User = get_user_model()
```

We can try and hack in a fix on the server:

accounts/authentication.py.

```
DOMAIN = 'superlists-staging.ottg.eu'
```

And check whether it works by doing a manual login. It does.

Fixing the Persona Bug

Here's how we go about baking in a fix, switching back to coding on our local PC. We start by moving the definition for the `DOMAIN` variable into *settings.py*, where we can later use the deploy script to override it:

superlists/settings.py (ch17l011).

```
# This setting is changed by the deploy script
DOMAIN = "localhost"

ALLOWED_HOSTS = [DOMAIN]
```

We feed that change back through the tests:

accounts/tests/test_authentication.py.

```
@@ -1,12 +1,14 @@
 from unittest.mock import patch
+from django.conf import settings
 from django.contrib.auth import get_user_model
 from django.test import TestCase
 User = get_user_model()

 from accounts.authentication import (
-    PERSONA_VERIFY_URL, DOMAIN, PersonaAuthenticationBackend
+    PERSONA_VERIFY_URL, PersonaAuthenticationBackend
 )

+
 @patch('accounts.authentication.requests.post')
 class AuthenticateTest(TestCase):

@@ -21,7 +23,7 @@ class AuthenticateTest(TestCase):
         self.backend.authenticate('an assertion')
         mock_post.assert_called_once_with(
             PERSONA_VERIFY_URL,
-            data={'assertion': 'an assertion', 'audience': DOMAIN}
+            data={'assertion': 'an assertion', 'audience': settings.DOMAIN}
         )
```

And then we change the implementation:

accounts/authentication.py.

```
@@ -1,8 +1,8 @@
 import requests
+from django.conf import settings
 from django.contrib.auth import get_user_model
 User = get_user_model()

 PERSONA_VERIFY_URL = 'https://verifier.login.persona.org/verify'
-DOMAIN = 'localhost'


@@ -11,7 +11,7 @@ class PersonaAuthenticationBackend(object):
     def authenticate(self, assertion):
         response = requests.post(
             PERSONA_VERIFY_URL,
-            data={'assertion': assertion, 'audience': DOMAIN}
+            data={'assertion': assertion, 'audience': settings.DOMAIN}
         )
         if response.ok and response.json()['status'] == 'okay':
             email = response.json()['email']
```

Rerunning the tests just to be sure:

```
$ python3 manage.py test accounts
[...]
Ran 14 tests in 0.053s
OK
```

Next we update our fabfile to make it adjust the domain in *settings.py*, removing the cumbersome two-line `sed` on `ALLOWED_HOSTS`:

deploy_tools/fabfile.py.

```
def _update_settings(source_folder, site_name):
    settings_path = source_folder + '/superlists/settings.py'
    sed(settings_path, "DEBUG = True", "DEBUG = False")
    sed(settings_path, 'DOMAIN = "localhost"', 'DOMAIN = "%s"' % (site_name,))
    secret_key_file = source_folder + '/superlists/secret_key.py'
    if not exists(secret_key_file):
        [...]
```

We redeploy, and spot the `sed` in the output:

```
$ fab deploy --host=superlists-staging.ottg.eu
[...]
[superlists-staging.ottg.eu] run: sed -i.bak -r -e s/DOMAIN =
"localhost"/DOMAIN = "superlists-staging.ottg.eu"/g "$(echo
/home/harry/sites/superlists-staging.ottg.eu/source/superlists/settings.py)"
[...]
```

Managing the Test Database on Staging

Now we can rerun our FTs, and get to the next failure: our attempt to create pre-authenticated sessions doesn't work, so the "My Lists" test fails:

```
$ python3 manage.py test functional_tests \
--liveserver=superlists-staging.ottg.eu

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\nSign in\nStart a new To-Do list'

Ran 7 tests in 72.742s

FAILED (errors=1)
```

It's because our test utility function `create_pre_authenticated_session` only acts on the local database. Let's find out how our tests can manage the database on the server.

A Django Management Command to Create Sessions

To do things on the server, we'll need to build a self-contained script that can be run from the command line on the server, most probably via Fabric.

When trying to build standalone scripts that work with the Django environment, can talk to the database and so on, there are some fiddly issues you need to get right, like setting the `DJANGO_SETTINGS_MODULE` environment variable correctly, and getting the `sys.path` right. Instead of messing about with all that, Django lets you create your own "management commands" (commands you can run with `python manage.py`), which will do all that path mangling for you. They live in a folder called *management/commands* inside your apps:

```
$ mkdir -p functional_tests/management/commands
$ touch functional_tests/management/__init__.py
$ touch functional_tests/management/commands/__init__.py
```

The boilerplate in a management command is a class that inherits from `django.core.management.BaseCommand`, and that defines a method called `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 handle(self, email, *_, **__):
            session_key = create_pre_authenticated_session(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
```

We've taken the code for `create_pre_authenticated_session` code from *test_my_lists.py*. `handle` will pick up an email address as the first command-line argument, and then return the session key that we'll want to add to our browser cookies, and the management command prints it out at the command line. Try it out:

```
$ python3 manage.py create_session a@b.com
Unknown command: 'create_session'
```

One more step: we need to add `functional_tests` to our *settings.py* for it to recognise it as a real app that might have management commands as well as tests:

superlists/settings.py.

```
+++ b/superlists/settings.py
@@ -42,6 +42,7 @@ INSTALLED_APPS = (
     'lists',
     'south',
     'accounts',
+    'functional_tests',
 )
```

Now it works:

```
$ python3 manage.py create_session a@b.com
qnslckvp2aga7tm6xuivyb0ob1akzzwl
```

Getting the FT to Run the Management Command on the Server

Next we need to adjust `test_my_lists` so that it runs the local function when we're on the local server, and make it run the management command on the staging server if we're on that:

functional_tests/test_my_lists.py (ch17l016).

```
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.against_staging:
            session_key = create_session_on_server(self.server_host, email)
        else:
            session_key = create_pre_authenticated_session(email)
        ## to set a cookie we need to first visit the domain.
        ## 404 pages load the quickest!
        self.browser.get(self.server_url + "/404_no_such_url/")
        self.browser.add_cookie(dict(
            name=settings.SESSION_COOKIE_NAME,
            value=session_key,
            path='/',
        ))

    [...]
```

Let's see how we know whether or not we're working against the staging server. `self.against_staging` gets populated in *base.py*:

functional_tests/base.py (ch17l017).

```
from .server_tools import reset_database  #❶

class FunctionalTest(StaticLiveServerCase):

    @classmethod
    def setUpClass(cls):
        for arg in sys.argv:
            if 'liveserver' in arg:
                cls.server_host = arg.split('=')[1] #❷
                cls.server_url = 'http://' + cls.server_host
                cls.against_staging = True #❸
                return
        super().setUpClass()
        cls.against_staging = False
        cls.server_url = cls.live_server_url

    @classmethod
    def tearDownClass(cls):
        if not cls.against_staging:
            super().tearDownClass()

    def setUp(self):
        if self.against_staging:
            reset_database(self.server_host) #❹
        self.browser = webdriver.Firefox()
        self.browser.implicitly_wait(3)
```

❷ ❸ Instead of just storing `cls.server_url`, we also store the `server_host` and `against_staging` attributes if we detect the `liveserver` command-line argument.

❶ ❹ We also need a way of resetting the server database in between each test. I'll explain the logic of the session-creation code, which should also explain how this works.

An Additional Hop via subprocess

Because our tests are Python 3, we can't directly call our Fabric functions, which are Python 2. Instead, we have to do an extra hop and call the `fab` command as a new process, like we do from the command line when we do server deploys. Here's how that looks, in a module called *server_tools*:

functional_tests/server_tools.py.

```
from os import path
import subprocess
THIS_FOLDER = path.abspath(path.dirname(__file__))

def create_session_on_server(host, email):
    return subprocess.check_output(
        [
            'fab',
            'create_session_on_server:email={}'.format(email), #❶❷
            '--host={}'.format(host),
            '--hide=everything,status', #❸
        ],
        cwd=THIS_FOLDER
    ).decode().strip() #❹


def reset_database(host):
    subprocess.check_call(
        ['fab', 'reset_database', '--host={}'.format(host)],
        cwd=THIS_FOLDER
    )
```

Here we use the `subprocess` module to call some Fabric functions using the `fab` command.

❶ As you can see, the command-line syntax for arguments to `fab` functions is quite simple, a colon and then a variable=argument syntax.

❷ Incidentally, this is also the first time I've shown the "new-style" string formatting syntax. As you can see it uses curly brackets `{}` instead of `%s`. I slightly prefer it to the old style, but you're bound to come across both if you spend any time with Python. Take a look at some of the examples in the Python docs to learn more.

❸ ❹ Because of all the hopping around via Fabric and subprocesses, we're forced to be quite careful about extracting the session key as a string from the output of the command as it gets run on the server.

You may need to adapt the call to `subprocess` if you are using a custom username or password: make it match the `fab` arguments you use when you run the automated deployment script.

By the time you read this book, Fabric may well have been fully ported to Python 3. If so, investigate using the Fabric context managers to call Fabric functions directly inline with your code.

Finally, let's look at the fabfile that defines those two commands we want to run server side, to reset the database or set up the session:

functional_tests/fabfile.py.

```
from fabric.api import env, run


def _get_base_folder(host):
    return '~/sites/' + host

def _get_manage_dot_py(host):
    return '{path}/virtualenv/bin/python {path}/source/manage.py'.format(
        path=_get_base_folder(host)
    )


def reset_database():
    run('{manage_py} flush --noinput'.format(
        manage_py=_get_manage_dot_py(env.host)
    ))


def create_session_on_server(email):
    session_key = run('{manage_py} create_session {email}'.format(
        manage_py=_get_manage_dot_py(env.host),
        email=email,
    ))
    print(session_key)
```

Does that make a reasonable amount of sense? We've got a function that can create a session in the database. If we detect we're running locally, we call it directly. If we're against the server, there's a couple of hops: we use `subprocess` to get to Fabric via `fab`, which lets us run a management command that calls that same function, on the server.

How about an ASCII-art illustration?

Locally:

```
MyListsTest
.create_pre_authenticated_session --> .management.commands.create_session
                                      .create_pre_authenticated_session
```

Against staging:

```
MyListsTest
.create_pre_authenticated_session     .management.commands.create_session
                                      .create_pre_authenticated_session
     |
    \|/                                          /|\
                                                  |
server_tools
.create_session_on_server               run manage.py create_session

     |                                            /|\
    \|/                                           |

subprocess.check_output  -->   fab  -->  fabfile.create_session_on_server
```

Anyway, let's see if it works. First, locally, to check we didn't break anything:

```
$ python3 manage.py test functional_tests.test_my_lists
[...]
OK
```

Next, against the server. We push our code up first:

```
$ git push  # you'll need to commit changes first.
$ cd deploy_tools
$ fab deploy --host=superlists-staging.ottg.eu
```

And now we run the test—notice we can include `elspeth@` in the specification of the `liveserver` argument now:

```
$ python3 manage.py test functional_tests.test_my_lists \
--liveserver=elspeth@superlists-staging.ottg.eu
Creating test database for alias 'default'...
[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
```

Looking good! We can rerun all the tests to make sure…

```
$ python3 manage.py test functional_tests \
--liveserver=elspeth@superlists-staging.ottg.eu
Creating test database for alias 'default'...
[superlists-staging.ottg.eu] Executing task 'reset_database'
[...]
Ran 7 tests in 89.494s
```

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

Hooray!

I've shown one way of managing the test database, but you could experiment with others—for example, if you were using MySQL or Postgres, you could open up an SSH tunnel to the server, and use port forwarding to talk to the database directly. You could then amend `settings.DATABASES` during FTs to talk to the tunnelled port.

Warning: Be Careful Not to Run Test Code Against the Live Server

We're into dangerous territory, now that we have code that can directly affect a database on the server. You want to be very, very careful that you don't accidentally blow away your production database by running FTs against the wrong host.

You might consider putting some safeguards in place at this point. For example, you could put staging and production on different servers, and make it so they use different keypairs for authentication, with different passphrases.

This is similar dangerous territory to running tests against clones of production data, if you remember my little story about accidentally sending thousands of duplicate invoices to clients. LFMF.

Baking In Our Logging Code

Before we finish, let's "bake in" our logging code. It would be useful to keep our new logging code in there, under source control, so that we can debug any future login problems. They may indicate someone is up to no good, after all.

Let's start by saving the Gunicorn config to our template file in *deploy_tools*:

deploy_tools/gunicorn-upstart.template.conf.

```
[...]
chdir /home/elspeth/sites/SITENAME/source

exec ../virtualenv/bin/gunicorn \
    --bind unix:/tmp/SITENAME.socket \
    --access-logfile ../access.log \
    --error-logfile ../error.log \
    superlists.wsgi:application
```

Using Hierarchical Logging Config

When we hacked in the `logging.warning` earlier, we were using the root logger. That's not normally a good idea, since any third-party module can mess with the root logger. The normal pattern is to use a logger named after the file you're in, by using:

```
logger = logging.getLogger(__name__)
```

Logging configuration is hierarchical, so you can define "parent" loggers for top-level modules, and all the Python modules inside them will inherit that config.

Here's how we add a logger for both our apps into *settings.py*:

superlists/settings.py.

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
        },
    },
    'loggers': {
         'django': {
             'handlers': ['console'],
         },
         'accounts': {
             'handlers': ['console'],
         },
         'lists': {
             'handlers': ['console'],
         },
     },
     'root': {'level': 'INFO'},
}
```

Now *accounts.models*, *accounts.views*, *accounts.authentication*, and all the others will inherit the `logging.StreamHandler` from the parent *accounts* logger.

Unfortunately, because of Django's project structure, there's no way of defining a top-level logger for your whole project (aside from using the root logger), so you have to define one logger per app.

Here's how to write a test for logging behaviour:

accounts/tests/test_authentication.py (ch17l023).

```
import logging
[...]

@patch('accounts.authentication.requests.post')
class AuthenticateTest(TestCase):
    [...]
```

```
    def test_logs_non_okay_responses_from_persona(self, mock_post):
        response_json = {
            'status': 'not okay', 'reason': 'eg, audience mismatch'
        }
        mock_post.return_value.ok = True
        mock_post.return_value.json.return_value = response_json  #❶

        logger = logging.getLogger('accounts.authentication')  #❷
        with patch.object(logger, 'warning') as mock_log_warning:  #❸
            self.backend.authenticate('an assertion')

        mock_log_warning.assert_called_once_with(
            'Persona says no. Json was: {}'.format(response_json)  #❹
        )
```

❶ We set up our test with some data that should cause some logging.

❷ We retrieve the actual logger for the module we're testing.

❸ We use `patch.object` to temporarily mock out its warning function, by using `with` to make it a *context manager* around the function we're testing.

❹ And then it's available for us to make assertions against.

That gives us:

```
AssertionError: Expected 'warning' to be called once. Called 0 times.
```

Let's just try it out, to make sure we really are testing what we think we are:

accounts/authentication.py (ch17l024).

```
import logging
logger = logging.getLogger(__name__)
[...]

        if response.ok and response.json()['status'] == 'okay':
            [...]
        else:
            logger.warning('foo')
```

We get the expected failure:

```
AssertionError: Expected call: warning("Persona says no. Json was: {'status':
'not okay', 'reason': 'eg, audience mismatch'}")
Actual call: warning('foo')
```

And so we settle in with our real implementation:

accounts/authentication.py (ch17l025).

```
    else:
        logger.warning(
            'Persona says no. Json was: {}'.format(response.json())
        )
```

```
$ python3 manage.py test accounts
[...]
```

```
Ran 15 tests in 0.033s

OK
```

You can easily imagine how you could test more combinations at this point, if you wanted different error messages for `response.ok != True`, and so on.

Wrap-Up

We now have test fixtures that work both locally and on the server, and we've got some more robust logging configuration.

But before we can deploy our actual live site, we'd better actually give the users what they wanted—the next chapter describes how to give them the ability to save their lists on a "My Lists" page.

Fixtures and Logging

De-duplicate your FTs, with caution
: Every single FT doesn't need to test every single part of your application. In our case, we wanted to avoid going through the full login process for every FT that needs an authenticated user, so we used a test fixture to "cheat" and skip that part. You might find other things you want to skip in your FTs. A word of caution however: functional tests are there to catch unpredictable interactions between different parts of your application, so be wary of pushing de-duplication to the extreme.

Test fixtures
: Test fixtures refers to test data that needs to be set up as a precondition before a test is run—often this means populating the database with some information, but as we've seen (with browser cookies), it can involve other types of preconditions.

Avoid JSON fixtures
: Django makes it easy to save and restore data from the database in JSON format (and others) using the `dumpdata` and `loaddata` management commands. Most people recommend against using these for test fixtures, as they are painful to manage when your database schema changes. Use the ORM, or a tool like factory_boy.

Fixtures also have to work remotely
: `LiveServerTestCase` makes it easy to interact with the test database using the Django ORM for tests running locally. Interacting with the database on the staging server is not so straightforward—one solution is Django management commands, as I've shown, but you should explore what works for you, and be careful!

Use loggers named after the module you're in
: The root logger is a single global object, available to any library that's loaded in your Python process, so you're never quite in control of it. Instead, follow the `logging.getLogger(__name__)` pattern to get one that's unique to your module, but that inherits from a top-level configuration you control.

Test important log messages
: As we saw, log messages can be critical to debugging issues in production. If a log message is important enough to keep in your codebase, it's probably important enough to test. We follow the rule of thumb that anything above `logging.INFO` definitely needs a test. Using `patch.object` on the logger for the module you're testing is one convenient way of unit testing it.

CHAPTER 18

Finishing "My Lists": Outside-In TDD

In this chapter I'd like to talk about a technique called "Outside-In" TDD. It's pretty much what we've been doing all along. Our "double-loop" TDD process, in which we write the functional test first and then the unit tests, is already a manifestation of outside-in—we design the system from the outside, and build up our code in layers. Now I'll make it explicit, and talk about some of the common issues involved.

The Alternative: "Inside Out"

The alternative to "Outside In" is to work "Inside Out", which is the way most people intuitively work before they encounter TDD. After coming up with a design, the natural inclination is sometimes to implement it starting with the innermost, lowest-level components first.

For example, when faced with our current problem, providing users with a "My Lists" page of saved lists, the temptation is to start by adding an "owner" attribute to the List model object, reasoning that an attribute like this is "obviously" going to be required. Once that's in place, we would modify the more peripheral layers of code, such as views and templates, taking advantage of the new attribute, and then finally add URL routing to point to the new view.

It feels comfortable because it means you're never working on a bit of code that is dependent on something that hasn't yet been implemented. Each bit of work on the inside is a solid foundation on which to build the next layer out.

But working inside-out like this also has some weaknesses.

Why Prefer "Outside-In"?

The most obvious problem with inside-out is that it requires us to stray from a TDD workflow. Our functional test's first failure might be due to missing URL routing, but

we decide to ignore that and go off adding attributes to our database model objects instead.

We might have ideas in our head about the new desired behaviour of our inner layers like database models, and often these ideas will be pretty good, but they are actually just speculation about what's really required, because we haven't yet built the outer layers that will use them.

One problem that can result is to build inner components that are more general or more capable than we actually need, which is a waste of time, and an added source of complexity for your project. Another common problem is that you create inner components with an API which is convenient for their own internal design, but which later turns out to be inappropriate for the calls your outer layers would like to make … worse still, you might end up with inner components which, you later realise, don't actually solve the problem that your outer layers need solved.

In contrast, working outside-in in allows you to use each layer to imagine the most convenient API you could want from the layer beneath it. Let's see it in action.

The FT for "My Lists"

As we work through the following functional test, we start with the most outward-facing (presentation layer), through to the view functions (or "controllers"), and lastly the innermost layers, which in this case will be model code.

We know our `create_pre_authenticated_session` code works now, so we can just write our FT to look for a "My Lists" page:

functional_tests/test_my_lists.py

```
def test_logged_in_users_lists_are_saved_as_my_lists(self):
    # Edith is a logged-in user
    self.create_pre_authenticated_session('edith@example.com')

    # She goes to the home page and starts a list
    self.browser.get(self.server_url)
    self.get_item_input_box().send_keys('Reticulate splines\n')
    self.get_item_input_box().send_keys('Immanentize eschaton\n')
    first_list_url = self.browser.current_url

    # She notices a "My lists" link, for the first time.
    self.browser.find_element_by_link_text('My lists').click()

    # She sees that her list is in there, named according to its
    # first list item
    self.browser.find_element_by_link_text('Reticulate splines').click()
    self.assertEqual(self.browser.current_url, first_list_url)

    # She decides to start another list, just to see
    self.browser.get(self.server_url)
```

```
        self.get_item_input_box().send_keys('Click cows\n')
        second_list_url = self.browser.current_url

        # Under "my lists", her new list appears
        self.browser.find_element_by_link_text('My lists').click()
        self.browser.find_element_by_link_text('Click cows').click()
        self.assertEqual(self.browser.current_url, second_list_url)

        # She logs out.  The "My lists" option disappears
        self.browser.find_element_by_id('id_logout').click()
        self.assertEqual(
            self.browser.find_elements_by_link_text('My lists'),
            []
        )
```

If you run it, the first error should look like this:

```
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"link text","selector":"My lists"}' ; Stacktrace:
```

The Outside Layer: Presentation and Templates

The test is currently failing saying that it can't find a link saying "My Lists". We can address that at the presentation layer, in *base.html*, in our navigation bar. Here's the minimal code change:

lists/templates/base.html (ch18l002-1).

```
{% if user.email %}
    <ul class="nav navbar-nav">
        <li><a href="#">My lists</a></li>
    </ul>
    <a class="btn navbar-btn navbar-right" id="id_logout" [...]
```

Of course, that link doesn't actually go anywhere, but it does get us along to the next failure:

```
    self.browser.find_element_by_link_text('Reticulate splines').click()
[...]
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"link text","selector":"Reticulate splines"}' ; Stacktrace:
```

Which is telling us we're going to have to build a page that lists all of a user's lists by title. Let's start with the basics—a URL and a placeholder template for it.

Again, we can go outside-in, starting at the presentation layer with just the URL and nothing else:

lists/templates/base.html (ch18l002-2).

```
<ul class="nav navbar-nav">
    <li><a href="{% url 'my_lists' user.email %}">My lists</a></li>
</ul>
```

Moving Down One Layer to View Functions (the Controller)

That will cause a template error, so we'll start to move down from the presentation layer and URLs down to the controller layer, Django's view functions.

As always, we start with a test:

lists/tests/test_views.py (ch18l003).

```
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')
```

That gives:

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

And we fix it, still at the presentation level, in *urls.py*:

lists/urls.py.

```
urlpatterns = patterns('',
    url(r'^(\d+)/$', 'lists.views.view_list', name='view_list'),
    url(r'^new$', 'lists.views.new_list', name='new_list'),
    url(r'^users/(.+)/$', 'lists.views.my_lists', name='my_lists'),
)
```

That gives us a test failure, which informs us of what we should do as we move down to the next level:

```
django.core.exceptions.ViewDoesNotExist: Could not import lists.views.my_lists.
View does not exist in module lists.views.
```

We move in from the presentation layer to the views layer, and create a minimal placeholder:

lists/views.py (ch18l005).

```
def my_lists(request, email):
    return render(request, 'my_lists.html')
```

And, a minimal template:

lists/templates/my_lists.html.

```
{% extends 'base.html' %}

{% block header_text %}My Lists{% endblock %}
```

That gets our unit tests passing, but our FT is still at the same point, saying that the "My Lists" page doesn't yet show any lists. It wants them to be clickable links named after the first item:

```
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"link text","selector":"Reticulate splines"}' ; Stacktrace:
```

Another Pass, Outside-In

At each stage, we still let the FT drive what development we do.

Starting again at the outside layer, in the template, we begin to write the template code we'd like to use to get the "My Lists" page to work the way we want it to. As we do so, we start to specify the API we want from the code at the layers below.

A Quick Restructure of the Template Inheritance Hierarchy

Currently there's no place in our base template for us to put any new content. Also, the "My Lists" page doesn't need the new item form, so we'll put that into a block too, making it optional:

lists/templates/base.html (ch18l007-1).

```
        <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>
```

lists/templates/base.html (ch18l007-2).

```
        <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="http://code.jquery.com/jquery.min.js"></script>
[...]
```

Designing Our API Using the Template

Meanwhile, in *my_lists.html* we override the `list_form` and say it should be empty…

lists/templates/my_lists.html.

```
{% extends 'base.html' %}

{% block header_text %}My Lists{% endblock %}

{% block list_form %}{% endblock %}
```

And then we can just work inside the `extra_content` block:

lists/templates/my_lists.html.

```
[...]

{% block list_form %}{% endblock %}

{% block extra_content %}
    <h2>{{ owner.email }}'s lists</h2>  ❶
    <ul>
        {% for list in owner.list_set.all %}  ❷
            <li><a href="{{ list.get_absolute_url }}">{{ list.name }}</a></li>  ❸
        {% endfor %}
    </ul>
{% endblock %}
```

We've made several design decisions in this template which are going to filter their way down through the code:

❶ We want a variable called `owner` to represent the user in our template.

❷ We want to be able to iterate through the lists created by the user using `owner.list_set.all` (I happen to know we get this for free from the Django ORM).

❸ We want to use `list.name` to print out the "name" of the list, which is currently specified as the text of its first element.

Outside-In TDD is sometimes called "programming by wishful thinking", and you can see why. We start writing code at the higher levels based on what we wish we had at the lower levels, even though it doesn't exist yet!

We can rerun our FTs, to check we didn't break anything, and to see whether we've got any further:

```
$ python3 manage.py test functional_tests
[...]
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
```

```
element: {"method":"link text","selector":"Reticulate splines"}' ; Stacktrace:

 ----------------------------------------------------------------------
Ran 7 tests in 77.613s

FAILED (errors=1)
```

Well, no further, but at least we didn't break anything. Time for a commit:

```
$ git add lists
$ git diff --staged
$ git commit -m "url, placeholder view, and first-cut templates for my_lists"
```

Moving Down to the Next Layer: What the View Passes to the Template

lists/tests/test_views.py (ch18l011).

```
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)
```

Gives:

```
KeyError: 'owner'
```

So:

lists/views.py (ch18l012).

```
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})
```

That gets our new test passing, but we'll also see an error from the previous test. We just need to add a user for it as well:

lists/tests/test_views.py (ch18l013).

```
    def test_my_lists_url_renders_my_lists_template(self):
        User.objects.create(email='a@b.com')
        [...]
```

And we get to an OK:

```
OK
```

The Next "Requirement" from the Views Layer: New Lists Should Record Owner

Before we move down to the model layer, there's another part of the code at the views layer that will need to use our model: we need some way for newly created lists to be assigned to an owner, if the current user is logged in to the site.

Here's a first crack at writing the test:

lists/tests/test_views.py (ch18l014).

```
from django.http import HttpRequest
[...]
from lists.views import new_list
[...]

class NewListTest(TestCase):
    [...]

    def test_list_owner_is_saved_if_user_is_authenticated(self):
        request = HttpRequest()
        request.user = User.objects.create(email='a@b.com')
        request.POST['text'] = 'new list item'
        new_list(request)
        list_ = List.objects.first()
        self.assertEqual(list_.owner, request.user)
```

This test uses the raw view function, and manually constructs an `HttpRequest` because it's slightly easier to write the test that way. Although the Django test client does have a helper function called `login`, it doesn't work well with external authentication services. The alternative would be to manually create a session object (like we do in the functional tests), or to use mocks, and I think both of those would end up uglier than this version. If you're curious, you could have a go at writing it differently.

The test fails as follows:

```
AttributeError: 'List' object has no attribute 'owner'
```

To fix this, we can try writing code like this:

lists/views.py (ch18l015).

```
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})
```

But it won't actually work, because we don't know how to save a list owner yet:

```
        self.assertEqual(list_.owner, request.user)
    AttributeError: 'List' object has no attribute 'owner'
```

A Decision Point: Whether to Proceed to the Next Layer with a Failing Test

In order to get this test passing, as it's written now, we have to move down to the model layer. However, it means doing more work with a failing test, which is not ideal.

The alternative is to rewrite the test to make it more *isolated* from the level below, using mocks.

On the one hand, it's a lot more effort to use mocks, and it can lead to tests that are harder to read. On the other hand, imagine if our app was more complex, and there were several more layers between the outside and the inside. Imagine leaving three or four or five layers of tests, all failing while we wait to get to the bottom layer to implement our critical feature. While tests are failing, we're not sure that layer really works, on its own terms, or not. We have to wait until we get to the bottom layer.

This is a decision point you're likely to run into in your own projects. Let's investigate both approaches. We'll start by taking the shortcut, and leaving the test failing. In the next chapter, we'll come back to this exact point, and investigate how things would have gone if we'd used more isolation.

Let's do a commit, and then *tag* the commit as a way of remembering our position for the next chapter:

```
$ git commit -am"new_list view tries to assign owner but cant"
$ git tag revisit_this_point_with_isolated_tests
```

Moving Down to the Model Layer

Our outside-in design has driven out two requirements for the model layer: we want to be able to assign an owner to a list using the attribute `.owner`, and we want to be able to access the list's owner with the API `owner.list_set.all`.

Let's write a test for that:

lists/tests/test_models.py (ch18l018).

```
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())
```

And that gives us a new unit test failure:

```
        list_ = List.objects.create(owner=user)
        [...]
    TypeError: 'owner' is an invalid keyword argument for this function
```

The naive implementation would be this:

```
    from django.conf import settings
    [...]

    class List(models.Model):
        owner = models.ForeignKey(settings.AUTH_USER_MODEL)
```

But we want to make sure the list owner is optional. Explicit is better than implicit, and tests are documentation, so let's have a test for that too:

lists/tests/test_models.py (ch18l020).

```
    def test_list_owner_is_optional(self):
        List.objects.create()  # should not raise
```

The correct implementation is this:

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])
```

Now running the tests gives the usual database error:

```
        return Database.Cursor.execute(self, query, params)
    django.db.utils.OperationalError: table lists_list has no column named owner_id
```

Because we need to do make some migrations:

```
    $ python3 manage.py makemigrations
    Migrations for 'lists':
      0006_list_owner.py:
        - Add field owner to list
```

We're almost there, a couple more failures:

```
    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_saving_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.
```

We're moving back up to the views layer now, just doing a little tidying up. Notice that these are in the old test for the `new_list` view, when we haven't got a logged-in user. We should only save the list owner when the user is actually logged in. The `.is_authenticated()` function we defined in Chapter 16 comes in useful now (when they're not logged in, Django represents users using a class called `AnonymousUser`, whose `.is_authenticated()` always returns `False`):

lists/views.py (ch18l023).

```
    if form.is_valid():
        list_ = List()
        if request.user.is_authenticated():
            list_.owner = request.user
        list_.save()
        form.save(for_list=list_)
        [...]
```

And that gets us passing!

```
$ python3 manage.py test lists
Creating test database for alias 'default'...
.....................................
 ----------------------------------------------------------------------
Ran 39 tests in 0.237s

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

This is a good time for a commit:

```
$ git add lists
$ git commit -m"lists can have owners, which are saved on creation."
```

Final Step: Feeding Through the .name API from the Template

The last thing our outside-in design wanted came from the templates, which wanted to be able to access a list "name" based on the text of its first item:

lists/tests/test_models.py (ch18l024).

```
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 (ch18l025).

```
    @property
    def name(self):
        return self.item_set.first().text
```

The @property Decorator in Python

If you haven't seen it before, the `@property` decorator transforms a method on a class to make it appear to the outside world like an attribute.

This is a powerful feature of the language, because it makes it easy to implement "duck typing", to change the implementation of a property without changing the interface of the class. In other words, if we decide to change `.name` into being a "real" attribute on the model, which is stored as text in the database, then we will be able to do so entirely transparently—as far as the rest of our code is concerned, they will still be able to just access `.name` and get the list name, without needing to know about the implementation.

Of course, in the Django template language, `.name` would still call the method even if it didn't have `@property`, but that's a particularity of Django, and doesn't apply to Python in general…

And that, believe it or not, actually gets us a passing test, and a working "My Lists" page (Figure 18-1)!

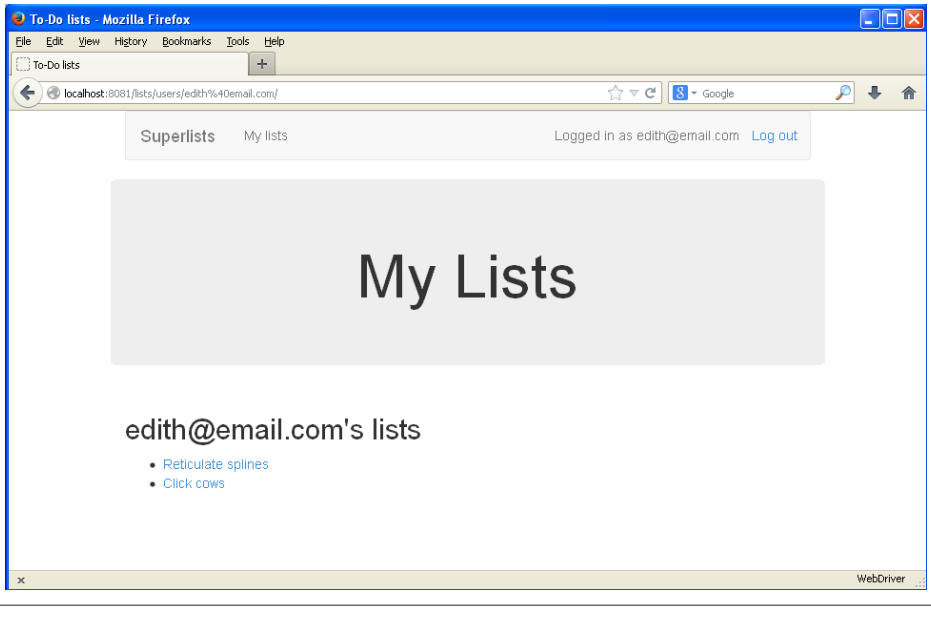


Figure 18-1. The "My Lists" page, in all its glory (and proof I did test on Windows)

```
$ python3 manage.py test functional_tests
[...]
Ran 7 tests in 93.819s

OK
```

But we know we cheated to get there. The Testing Goat is eyeing us suspiciously. We left a test failing at one layer while we implemented its dependencies at the lower layer. Let's see how things would play out if we were to use better test isolation…

Outside-In TDD

Outside-In TDD
: A methodology for building code, driven by tests, which proceeds by starting from the "outside" layers (presentation, GUI), and moving "inwards" step by step, via view/controller layers, down towards the model layer. The idea is to drive the design of your code from the use to which it is going to be put, rather than trying to anticipate requirements from the ground up.

Programming by wishful thinking
: The outside-in process is sometimes called "programming by wishful thinking". Actually, any kind of TDD involves some wishful thinking. We're always writing tests for things that don't exist yet.

The pitfalls of outside-in
: Outside-In isn't a silver bullet. It encourages us to focus on things that are immediately visible to the user, but it won't automatically remind us to write other critical tests that are less user-visible, things like security for example. You'll need to remember them yourself.

CHAPTER 19

Test Isolation, and "Listening to Your Tests"

In the last chapter, we made the decision to leave a unit test failing in the views layer while we proceeded to write more tests and more code at the models layer to get it to pass.

We got away with it because our app was simple, but I should stress that, in a more complex application, this would be a dangerous decision. Proceeding to work on lower levels while you're not sure that the higher levels are *really* finished or not is a risky strategy.

I'm grateful to Gary Bernhardt, who took a look at an early draft of the previous chapter, and encouraged me to get into a longer discussion of test isolation.

Ensuring isolation between layers does involve more effort (and more of the dreaded mocks!), but it can also help to drive out improved design, as we'll see in this chapter.

Revisiting Our Decision Point: The Views Layer Depends on Unwritten Models Code

Let's revisit the point we were at half-way through the last chapter, when we couldn't get the `new_list` view to work because lists didn't have the `.owner` attribute yet.

We'll actually go back in time and check out the old codebase, so that we can see how things would have worked if we'd used more isolated tests.

```
$ git checkout -b more-isolation  # a branch for this experiment
$ git reset --hard revisit_this_point_with_isolated_tests
```

Here's what our failing tests looks like:

lists/tests/test_views.py.

```
class NewListTest(TestCase):
    [...]

    def test_list_owner_is_saved_if_user_is_authenticated(self):
        request = HttpRequest()
        request.user = User.objects.create(email='a@b.com')
        request.POST['text'] = 'new list item'
        new_list(request)
        list_ = List.objects.first()
        self.assertEqual(list_.owner, request.user)
```

And here's what our attempted solution looked like:

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})
```

And at this point, the view test is failing because we don't have the model layer yet:

```
    self.assertEqual(list_.owner, request.user)
AttributeError: 'List' object has no attribute 'owner'
```

You won't see this error unless you actually check out the old code and revert *lists/models.py*. You should definitely do this, part of the objective of this chapter is to see whether we really can write tests for a models layer that doesn't exist yet.

A First Attempt at Using Mocks for Isolation

Lists don't have owners yet, but we can let the views layer tests pretend they do by using a bit of mocking:

lists/tests/test_views.py (ch19l003).

```
from unittest.mock import Mock, patch

from django.http import HttpRequest
from django.test import TestCase
[...]

    @patch('lists.views.List')  #❶
    def test_list_owner_is_saved_if_user_is_authenticated(self, mockList):
        mock_list = List.objects.create()  #❷
        mock_list.save = Mock()
```

```
        mockList.return_value = mock_list
        request = HttpRequest()
        request.user = User.objects.create()  #❸
        request.POST['text'] = 'new list item'

        new_list(request)

        self.assertEqual(mock_list.owner, request.user)  #❹
```

❶ We mock out the `List` function to be able to get access to any lists that might be created by the view.

❷ Then we create a real List object for the view to use. It has to be a real List object, otherwise the Item that the view is trying to save will fail with a foreign key error (this is an indication that the test is only partially isolated).

❸ We set a real user on the request object.

❹ And now we can make assertions about whether the list has had the `.owner` attribute set on it.

If we try to run this test now, it should pass.

```
$ python3 manage.py test lists
[...]
Ran 37 tests in 0.145s
OK
```

If you don't see a pass, make sure that your views code in *views.py* is exactly as I've shown it, using `List()`, not `List.objects.create`.

Using mocks does tie you to specific ways of using an API. This is one of the many trade-offs involved in the use of mock objects.

Using Mock side_effects to Check the Sequence of Events

The trouble with this test is that it can still let us get away with writing the wrong code by mistake. Imagine if we accidentally call save before we we assign the owner:

lists/views.py.

```
    if form.is_valid():
        list_ = List()
        list_.save()
        list_.owner = request.user
        form.save(for_list=list_)
        return redirect(list_)
```

The test, as it's written now, still passes:

```
OK
```

So we actually need to check, not just that the owner is assigned, but that it's assigned *before* we call save on our list object.

Here's how we can test the sequence of events using mocks—you can mock out a function, and use it as a spy to check on the state of the world at the moment it's called:

lists/tests/test_views.py (ch19l005).

```
@patch('lists.views.List')
def test_list_owner_is_saved_if_user_is_authenticated(self, mockList):
    mock_list = List.objects.create()
    mock_list.save = Mock()
    mockList.return_value = mock_list
    request = HttpRequest()
    request.user = Mock()
    request.user.is_authenticated.return_value = True
    request.POST['text'] = 'new list item'

    def check_owner_assigned():  #❶
        self.assertEqual(mock_list.owner, request.user)  #❷
    mock_list.save.side_effect = check_owner_assigned  #❸

    new_list(request)

    mock_list.save.assert_called_once_with()  #❹
```

❶ ❷ We define a function that makes the assertion about the thing we want to happen first: checking the list's owner has been set.

❸ We assign that check function as a `side_effect` to the thing we want to check happened second. When the view calls our mocked save function, it will go through this assertion. We make sure to set this up before we actually call the function we're testing.

❹ Finally, we make sure that the function with the `side_effect` was actually triggered, ie we did `.save()`. Otherwise our assertion may actually never have been run.

Two common mistakes when using mock side-effects are: assigning the side effect too late, i.e. *after* you call the function under test, and forgetting to check that the side-effect function was actually called. And by common, I mean, "I made them both several times while writing this chapter".

At this point, if you've still got the "broken" code from above, where we assign the owner but call save in the wrong order, you should now see a fail:

```
ERROR: test_list_owner_is_saved_if_user_is_authenticated
(lists.tests.test_views.NewListTest)
[...]
  File "/workspace/superlists/lists/views.py", line 17, in new_list
```

```
        list_.save()
    [...]
      File "/workspace/superlists/lists/tests/test_views.py", line 84, in
    check_owner_assigned
        self.assertEqual(mock_list.owner, request.user)
    AttributeError: 'List' object has no attribute 'owner'
```

Notice how the failure happens when we try and save, and then go inside our `side_effect` function.

We can get it passing again like this:

lists/views.py.

```
        if form.is_valid():
            list_ = List()
            list_.owner = request.user
            list_.save()
            form.save(for_list=list_)
            return redirect(list_)
```

...

```
    OK
```

But, boy, that's getting to be an ugly test!

Listen to Your Tests: Ugly Tests Signal a Need to Refactor

Whenever you find yourself having to write a test like this, and you're finding it hard work, it's likely that your tests are trying to tell you something. Nine lines of setup (three lines for the mock user, four more lines for the request object, and three for our side-effect function) is way too many.

What this test is trying to tell us is that our view is doing too much work, dealing with creating a form, creating a new list object *and* deciding whether or not to save an owner for the list.

We've already seen that we can make our views simpler and easier to understand by pushing some of the work down to a form class. Why does the view need to create the list object? Perhaps our `ItemForm.save` could do that? And why does the view need to make decisions about whether or not to save the `request.user`? Again, the form could do that.

While we're giving this form more responsibilities, it feels like it should probably get a new name too. We could call `NewListForm` instead, since that's a better representation of what it does… something like this?

lists/views.py.

```
    # don't enter this code yet, we're only imagining it.

    def new_list(request):
```

```
    form = NewListForm(data=request.POST)
    if form.is_valid():
        list_ = form.save(owner=request.user)  # creates both List and Item
        return redirect(list_)
    else:
        return render(request, 'home.html', {"form": form})
```

That would be neater! Let's see how we'd get to that state by using fully isolated tests.

Rewriting Our Tests for the View to Be Fully Isolated

Our first attempt at a test suite is for this view was highly *integrated*. It needed the database layer and the forms layer to be fully functional in order for it to pass. We've started trying to make it more isolated, let's now go all the way.

Keep the Old Integrated Test Suite Around as a Sanity Check

Let's rename our old `NewListTest` class to `NewListViewIntegratedTest`, and throw away our attempt at a mocky test for saving the owner, puttting back the integrated version, with a skip on it for now:

lists/tests/test_views.py (ch19l008).

```
import unittest
[...]

class NewListViewIntegratedTest(TestCase):

    def test_saving_a_POST_request(self):
        [...]

    @unittest.skip
    def test_list_owner_is_saved_if_user_is_authenticated(self):
        request = HttpRequest()
        request.user = User.objects.create(email='a@b.com')
        request.POST['text'] = 'new list item'
        new_list(request)
        list_ = List.objects.first()
        self.assertEqual(list_.owner, request.user)
```

Have you heard the term "integration test" and are wondering what the difference is with an "integrated test"? Go and take a peek at the definitions box in Chapter 22.

```
$ python3 manage.py test lists
[...]
Ran 37 tests in 0.139s
OK
```

A New Test Suite with Full Isolation

Let's start with a blank slate, and see if we can use isolated tests to drive a replacement of our `new_list` view. We'll call it `new_list2`, build it alongside the old view, and when we're ready, we can swap it in and see if the old integrated tests all still pass.

lists/views.py (ch19l009).

```
def new_list(request):
    [...]

def new_list2(request):
    pass
```

Thinking in Terms of Collaborators

In order to rewrite our tests to be fully isolated, we need to throw out our old way of thinking about the tests in terms of the "real" effects of the view on things like the database, and instead think of it in terms of the objects it collaborates with, and how it interacts with them.

In the new world, the view's main collaborator will be a form object, so we mock that out in order to be able to fully control it, and in order to be able to define, by wishful thinking, the way we want our form to work.

lists/tests/test_views.py (ch19l010).

```
from lists.views import new_list, new_list2
[...]

@patch('lists.views.NewListForm')  #❶
class NewListViewUnitTest(unittest.TestCase):  #❷

    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
```

❷ The Django `TestCase` class makes it too easy to write integrated tests. As a way of making sure we're writing "pure", isolated unit tests, we'll only use `unittest.TestCase`

❶ We mock out the NewListForm class (which doesn't even exist yet). It's going to be used in all the tests, so we mock it out at the class level.

❸ We set up a basic POST request in `setUp`, building up the request by hand rather than using the (overly integrated) Django Test Client.

❹ And we check the first thing about our new view: it initialises its collaborator, the `NewListForm`, with the correct constructor—the data from the request.

That will start with a failure, saying we don't have a `NewListForm` in our view yet.

```
AttributeError: <module 'lists.views' from
'/workspace/superlists/lists/views.py'> does not have the attribute
'NewListForm'
```

Let's create a placeholder for it:

lists/views.py (ch19l011).

```
from lists.forms import ExistingListItemForm, ItemForm, NewListForm
[...]
```

and:

lists/forms.py (ch19l012).

```
class ItemForm(forms.models.ModelForm):
    [...]

class NewListForm(object):
    pass

class ExistingListItemForm(ItemForm):
    [...]
```

Next we get a real failure:

```
AssertionError: Expected 'NewListForm' to be called once. Called 0 times.
```

And we implement like this:

lists/views.py (ch19l012-2).

```
def new_list2(request):
    NewListForm(data=request.POST)
```

```
$ python3 manage.py test lists
[...]
Ran 38 tests in 0.143s
OK
```

Let's continue. If the form is valid, we want to call save on it:

lists/tests/test_views.py (ch19l013).

```
@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)
```

That takes us to this:

lists/views.py (ch19l014).

```
def new_list2(request):
    form = NewListForm(data=request.POST)
    form.save(owner=request.user)
```

In the case where the form is valid, we want the view to return a redirect, to send us to see the object that the form has just created. So we mock out another of the view's collaborators, the `redirect` function:

lists/tests/test_views.py (ch19l015).

```
    @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 We mock out the `redirect` function, this time at the method level.

❷ `patch` decorators are applied innermost first, so the new mock is injected to our method as before the `mockNewListForm`.

❸ We specify we're testing the case where the form is valid.

❹ We check that the response from the view is the result of the `redirect` function.

❺ And we check that the redirect function was called with the object that the form returns on save.

That takes us to here:

lists/views.py (ch19l016).

```
def new_list2(request):
    form = NewListForm(data=request.POST)
    list_ = form.save(owner=request.user)
    return redirect(list_)
```

```
$ python3 manage.py test lists
[...]
Ran 40 tests in 0.163s
OK
```

And now the failure case—if the form is invalid, we want to render the home page template:

lists/tests/test_views.py (ch19l017).

```
@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}
    )
```

That gives us:

```
AssertionError: <django.http.response.HttpResponseRedirect object at
0x7f8d3f338a50> != <MagicMock name='render()' id='140244627467408'>
```

When using assert methods on mocks, like `assert_called_once_with`, it's doubly important to make sure you run the test and see it fail. It's all too easy to make a typo in your assert function name and end up calling a mock method that does nothing (mine was to write `asssert_called_once_with` with three essses, try it!)

We make a deliberate mistake, just to make sure our tests are comprehensive:

lists/views.py (ch19l018).

```
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})
```

That passes but it shouldn't! One more test then:

lists/tests/test_views.py (ch19l019).

```
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)
```

Which fails:

```
        self.assertFalse(mock_form.save.called)
    AssertionError: True is not false
```

And we get to to our neat, small finished view:

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})
...
```

```
    $ python3 manage.py test lists
    [...]
    Ran 42 tests in 0.163s
    OK
```

Moving Down to the Forms Layer

So we've built up our view function based on a "wishful thinking" version of a form called `NewItemForm`, which doesn't even exist yet.

We'll need the form's save method to create a new list, and a new item based on the text from the form's validated POST data. If we were to just dive in and use the ORM, the code might look something a bit like this:

```
    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()
```

This implementation depends on two classes from the model layer, `Item` and `List`. So, what would a well isolated test look like?

```
    class NewListFormTest(unittest.TestCase):

        @patch('lists.forms.List')  #❶
        @patch('lists.forms.Item')  #❷
        def test_save_creates_new_list_and_item_from_post_data(
            self, mockItem, mockList  #❸
        ):
```

```
        mock_item = mockItem.return_value
        mock_list = mockList.return_value
        user = Mock()
        form = NewListForm(data={'text': 'new item text'})
        form.is_valid() #❹

        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  #❺

        form.save(owner=user)

        self.assertTrue(mock_item.save.called)  #❻
```

❶ ❷ ❸ We mock out the two collaborators for our form from the models layer below.

❹ We need to call `is_valid()` so that the form populates the `.cleaned_data` dictionary where it stores validated data.

❺ We use the `side_effect` method to make sure that, when we save the new item object, we're doing so with a saved List and with the correct item text.

❻ As always, we double-check that our side-effect function was actually called.

Yuck! What an ugly test!

Keep Listening to Your Tests: Removing ORM Code from Our Application

Again, these tests are trying to tell us something: the Django ORM is hard to mock out, and our form class needs to know too much about how it works. Programming by wishful thinking again, what would be a simpler API that our form could use? How about something like this:

```
    def save(self):
        List.create_new(first_item_text=self.cleaned_data['text'])
```

Our wishful thinking says: how about we had a helper method that would live on the List class[1] and it will encapsulate all the logic of saving a new list object and its associated first item.

So let's write a test for that instead:

1. It could easily just be a standalone function, but hanging it on the model class is a nice way to keep track of where it lives, and gives a bit more of a hint as to what it will do.

lists/tests/test_forms.py (ch19l021).

```
import unittest
from unittest.mock import patch, Mock
from django.test import TestCase

from lists.forms import (
    DUPLICATE_ITEM_ERROR, EMPTY_LIST_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=lambda: 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'
        )
```

And while we're at it we can test the case where the user is an authenticated user too:

lists/tests/test_forms.py (ch19l022).

```
    @patch('lists.forms.List.create_new')
    def test_save_creates_new_list_with_owner_if_user_authenticatd(
        self, mock_List_create_new
    ):
        user = Mock(is_authenticated=lambda: 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
        )
```

You can see this is a much more readable test. Let's start implementing our new form. We start with the import:

lists/forms.py (ch19l023).

```
from lists.models import Item, List
```

Now mock tells us to create a placeholder for our `create_new` method:

```
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
```

And after a few steps, we should end up with a form save method like this:

lists/forms.py (ch19l025).

```
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'])
```

And passing tests:

```
$ python3 manage.py test lists
Ran 44 tests in 0.192s
OK
```

Hiding ORM Code Behind Helper Methods

One of the techniques that emerged from our use of isolated tests was the "ORM helper method".

Django's ORM lets you get things done quickly with a reasonably readable syntax (it's certainly much nicer than raw SQL!). But some people like to try and minimise the amount of ORM code in the application—particularly removing it from the views and forms layers.

One reason is that it makes it much easier to test those layers. But another is that it forces us to build helper functions that express our domain logic more clearly. Compare:

```
list_ = List()
list_.save()
item = Item()
item.list = list_
item.text = self.cleaned_data['text']
item.save()
```

With:

```
List.create_new(first_item_text=self.cleaned_data['text'])
```

This also applies to read queries as well as write. Imagine something like this:

```
Book.objects.filter(in_print=True, pub_date__lte=datetime.today())
```

Versus a helper method, like:

```
Book.all_available_books()
```

When we build helper functions, we can give them names that express what we are doing in terms of the business domain, which can actually make our code more legible, as well as giving us the benefit of keeping all ORM calls at the model layer, and thus making our whole application more loosely coupled.

Finally, Moving Down to the Models Layer

At the models layer, we no longer need to write isolated tests—the whole point of the models layer is to integrate with the database, so it's appropriate to write integrated tests:

lists/tests/test_models.py (ch19l026).

```
class ListModelTest(TestCase):

    def test_get_absolute_url(self):
        list_ = List.objects.create()
        self.assertEqual(list_.get_absolute_url(), '/lists/%d/' % (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)
```

Which gives:

```
TypeError: create_new() got an unexpected keyword argument 'first_item_text'
```

And that will take us to a first cut implementation that looks like this:

lists/models.py (ch19l027).

```
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_)
```

Notice we've been able to get all the way down to the models layer, driving a nice design for the views and forms layers, and the List model still doesn't support having an owner!

Now let's test the case where the list should have an owner, and add:

lists/tests/test_models.py (ch19l028).

```
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)
```

And while we're at it, we can write the tests for the new owner attribute:

lists/tests/test_models.py (ch19l029).

```
class ListModelTest(TestCase):
    [...]

    def test_lists_can_have_owners(self):
        List(owner=User())  # should not raise


    def test_list_owner_is_optional(self):
        List().full_clean()  # should not raise
```

These two are almost exactly the same tests we used in the last chapter, but I've re-written them slightly so they don't actually save objects—just having them as in-memory objects is enough to for this test.

Use in-memory (unsaved) model objects in your tests whenever you can, it makes your tests faster.

That gives:

```
$ python3 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)
```

We implement, just like we did in the last chapter:

lists/models.py (ch19l030-1).

```
from django.conf import settings
[...]


class List(models.Model):
    owner = models.ForeignKey(settings.AUTH_USER_MODEL, blank=True, null=True)
    [...]
```

That will give us all sorts of integrity failures, until we do a migration:

```
django.db.utils.OperationalError: no such column: lists_list.owner_id

FAILED (errors=28)
```

Building the migration will get us down to three failures:

```
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.
```

Let's deal with the first one, which is for our `create_new` method:

lists/models.py (ch19l030-3).

```
    @staticmethod
    def create_new(first_item_text, owner=None):
        list_ = List.objects.create(owner=owner)
        Item.objects.create(text=first_item_text, list=list_)
```

Back to Views

Two of our old integrated tests for the views layer are failing. What's happening?

```
ValueError: Cannot assign "<SimpleLazyObject:
<django.contrib.auth.models.AnonymousUser object at 0x7fbad1cb6c10>>":
"List.owner" must be a "User" instance.
```

Ah, the old view isn't discerning enough about what it does with list owners yet:

lists/views.py.

```
    if form.is_valid():
        list_ = List()
        list_.owner = request.user
        list_.save()
```

This is the point at which we realise that our old code wasn't fit for purpose. Let's fix it to get all our tests passing:

lists/views.py (ch19l031).

```
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):
    [...]
```

One of the benefits of integrated tests is that they help you to catch less predictable interactions like this. We'd forgotten about to write a test for the case where the user is not authenticated, but because the integrated tests use the stack all the way down, errors from the model layer came up to let us know we'd forgotten something:

```
$ python3 manage.py test lists
[...]
Ran 48 tests in 0.175s
OK
```

The Moment of Truth (and the Risks of Mocking)

So let's try switching out our old view, and activating our new view. We can make the swap in *urls.py*:

lists/urls.py.

```
[...]
    url(r'^new$', 'lists.views.new_list2', name='new_list'),
```

We should also remove the `unittest.skip` from our integrated test class, and make it point at our new view (`new_list2`), to see if our new code for list owners really works:

lists/tests/test_views.py (ch19l033).

```
class NewListViewIntegratedTest(TestCase):

    def test_saving_a_POST_request(self):
        [...]

    def test_list_owner_is_saved_if_user_is_authenticated(self):
        request = HttpRequest()
        request.user = User.objects.create(email='a@b.com')
        request.POST['text'] = 'new list item'
        new_list2(request)
        list_ = List.objects.first()
        self.assertEqual(list_.owner, request.user)
```

So what happens when we run our tests? Oh no!

```
ERROR: test_list_owner_is_saved_if_user_is_authenticated
[...]
ERROR: test_saving_a_POST_request
[...]
ERROR: test_redirects_after_POST
```

```
(lists.tests.test_views.NewListViewIntegratedTest)
  File "/workspace/superlists/lists/views.py", line 30, in new_list2
    return redirect(list_)
[...]
TypeError: argument of type 'NoneType' is not iterable

FAILED (errors=3)
```

Here's an important lesson to learn about test isolation: it might help you to drive out good design for individual layers, but it won't automatically verify the integration *between* your layers.

What's happened here is that the view was expecting the form to return a list item:

lists/views.py.

```
    list_ = form.save(owner=request.user)
    return redirect(list_)
```

But we forgot to make it return anything:

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'])
```

Thinking of Interactions Between Layers as "Contracts"

Ultimately, even if we had been writing nothing but isolated unit tests, our functional tests would have picked up this particular slip-up. But ideally we'd want our feedback cycle to be quicker—functional tests may take a couple of minutes to run, or even a few hours once your app starts to grow. Is there any way to avoid this sort of problem before it happens?

Methodologically, the way to do it is to think about the interaction between your layers in terms of contracts. Whenever we mock out the behaviour of one layer, we have to make a mental note that there is now an implicit contract between the layers, and that a mock on one layer should probably translate into a test at the layer below.

Here's the part of the contract that we missed:

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 The mocked `form.save` function is returning an object, which we expect our view to be able to use.

Identifying Implicit Contracts

It's worth reviewing each of the tests in `NewListViewUnitTest` and seeing what each mock is saying about the implicit contract:

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  #❹
    [...]


@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)  #❺

def test_renders_home_template_with_form_if_form_invalid(
    [...]
```

❶ We need to be able to initialise our form by passing it a POST request as data.

❷ ❹ It should have an `is_valid()` function which returns True or False appropriately, based on the input data.

❸ The form should have a `.save` method which will accept a `request.user`, which may or may not be a logged-in user, and deal with it appropriately.

❺ The form's `.save` method should return a new list object, for our view to redirect the user to.

If we have a look through our form tests, we'll see that, actually, only item ❸ is tested explicitly. On items ❶ and ❷ we were lucky—they're default features of a Django `ModelForm`, and they are actually covered by our tests for the parent `ItemForm` class.

But contract clause number ❹ managed to slip through the net.

When doing outside-in TDD with isolated tests, you need to keep track of each test's implicit assumptions about the contract which the next layer should implement, and remember to test each of those in turn later. You could use our scratchpad for this, or create a placeholder test with a `self.fail`.

Fixing the Oversight

Let's add a new test that our form should return the new saved list:

lists/tests/test_forms.py (ch19l038-1).

```
@patch('lists.forms.List.create_new')
def test_save_returns_new_list_object(self, mock_List_create_new):
    user = Mock(is_authenticated=lambda: 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)
```

And, actually, this is a good example—we have an implicit contract with the `List.create_new`, we want it to return the new list object. Let's add a placeholder test for that.

lists/tests/test_models.py (ch19l038-2).

```
class ListModelTest(TestCase):
    [...]

    def test_create_returns_new_list_object(self):
        self.fail()
```

So, we have one test failures that's telling us to fix the form save:

```
AssertionError: None != <MagicMock name='create_new()' id='139802647565536'>
FAILED (failures=2, errors=3)
```

Like this:

lists/forms.py (ch19l039-1).

```
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'])
```

That's a start, now we should look at our placeholder test:

```
[...]
FAIL: test_create_returns_new_list_object
    self.fail()
AssertionError: None

FAILED (failures=1, errors=3)
```

We flesh it out:

lists/tests/test_models.py (ch19l039-2).

```
    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>
```

And we add our return value:

lists/models.py (ch19l039-3).

```
    @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_
```

And that gets us to a fully passing test suite:

```
$ python3 manage.py test lists
[...]
Ran 50 tests in 0.169s

OK
```

One More Test

That's our code for saving list owners test-driven all the way down and working. But our functional test isn't passing quite yet:

```
$ python3 manage.py test functional_tests.test_my_lists
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"link text","selector":"Reticulate splines"}' ; Stacktrace:
```

It's because we have one last feature to implement, the `.name` attribute on list objects. Again, we can grab the test and code from the last chapter:

lists/tests/test_models.py (ch19l040).

```
    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')
```

(Again, since this is a model-layer test, it's OK to use the ORM. You could conceivable write this test using mocks, but there wouldn't be much point).

lists/models.py (ch19l041).

```
    @property
    def name(self):
        return self.item_set.first().text
```

And that gets us to a passing FT!

```
$ python3 manage.py test functional_tests.test_my_lists

Ran 1 test in 21.428s

OK
```

Tidy Up: What to Keep from Our Integrated Test Suite

Now everything is working, we can remove some redundant tests, and decide whether we want to keep any of our old integrated tests.

Removing Redundant Code at the Forms Layer

We can get rid of the test for the old save method on the `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_LIST_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_)
-
```

And in our actual code, we can get rid of two redundant save methods in *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_LIST_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)
-
```

Removing the Old Implementation of the View

We can now completely remove the old `new_list` view, and rename `new_list2` to `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)

[.. several more]
```

lists/urls.py.

```
--- a/lists/urls.py
+++ b/lists/urls.py
@@ -2,6 +2,6 @@ from django.conf.urls import patterns, url

 urlpatterns = patterns('',
     url(r'^(\d+)/$', 'lists.views.view_list', name='view_list'),
-    url(r'^new$', 'lists.views.new_list2', name='new_list'),
```

```
+    url(r'^new$', 'lists.views.new_list', name='new_list'),
     url(r'^users/(.+)/$', 'lists.views.my_lists', name='my_lists'),
 )
```

lists/views.py (ch19l047).

```
def new_list(request):
    form = NewListForm(data=request.POST)
    if form.is_valid():
        list_ = form.save(owner=request.user)
        [...]
```

And a quick check that all the tests still pass:

```
OK
```

Removing Redundant Code at the Forms Layer

Finally, we have to decide what (if anything) to keep from our integrated test suite.

One option is to throw them all away, and decide that the FTs will pick up any integration problems. That's perfectly valid.

On the other hand, we saw how integrated tests can warn you when you've made small mistakes in integrated your layers. We could keep just a couple of tests around as "sanity-checks", to give us a quicker feedback cycle.

How about these three:

lists/tests/test_views.py (ch19l048).

```
class NewListViewIntegratedTest(TestCase):

    def test_saving_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_LIST_ERROR))


    def test_saves_list_owner_if_user_logged_in(self):
        request = HttpRequest()
        request.user = User.objects.create(email='a@b.com')
        request.POST['text'] = 'new list item'
        new_list(request)
        list_ = List.objects.first()
        self.assertEqual(list_.owner, request.user)
```

If you're going to keep any intermediate-level tests at all, I like these three because they feel like they're doing the most "integration" jobs: they test the full stack, from the request down to the actual database, and they cover the three most important use cases of our view.

Conclusions: When to Write Isolated Versus Integrated Tests

Django's testing tools make it very easy to quickly put together integrated tests. The test runner helpfully creates a fast, in-memory version of your database and resets it for you in between each tests. The `TestCase` class and the Test Client make it easy to test your views, from checking whether database objects are modified, confirming that your URL mappings work, and inspecting the rendering of the templates. This lets you get started with testing very easily and get good coverage across your whole stack.

On the other hand, these kinds of integrated tests won't necessarily deliver the full benefit that rigorous unit testing and outside-in TDD are meant to confer in terms of design.

If we look at the example in this chapter, compare the code we had before and after:

Before.

```
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})
```

After.

```
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})
```

If we hadn't bothered to go down the isolation route, would we have bothered to refactor the view function? I know I didn't in the first draft of this book. I'd like to think I would have "in real life", but it's hard to be sure. But writing isolated tests does make you very aware of where the complexities in your code lie.

Let Complexity Be Your Guide

I'd say the point at which isolated tests start to become worth it is to do with complexity. The example in this book is extremely simple, so it's not often been worth it so far. Even in the example in this chapter, I can convince myself I didn't really *need* to write those isolated tests.

But once an application gains a little more complexity—if it starts growing any more layers between views and models, if you find yourself writing helper methods, or your own classes, then you will probably gain from writing more isolated tests.

Should You Do Both?

We already have our suite of functional tests, which will serve the purpose of telling us if we ever make any mistakes in integrating the different parts of our code together. Writing isolated tests can help us to drive out better design for our code, and to verify correctness in finer detail. Would a middle layer of integration tests serve any additional purpose?

I think the answer is potentially yes, if they can provide a faster feedback cycle, and help you identify more clearly what integration problems you suffer from—their tracebacks may provide you with better debug information than you would get from a functional test, for example.

There may even be a case for building them as a separate test suite—you could have one suite of fast, isolated unit tests that don't even use `manage.py`, because they don't need any of the database cleanup and teardown that the Django test runner gives you, and then the intermediate layer that uses Django, and finally the functional tests layer that, say, talks to a staging server. It may be worth it if each layer delivers incremental benefits.

It's a judgement call. I hope that, by going through this chapter, I've given you a feel for what the trade-offs are.

Onwards!

We're happy with our new version, so let's bring them across to master:

```
$ git add .
$ git commit -m"add list owners via forms. more isolated tests"
$ git checkout master
$ git checkout -b master-noforms-noisolation-bak # optional backup
$ git checkout master
$ git reset --hard more-isolation  # reset master to our branch.
```

In the meantime—those FTs are taking an annoyingly long time to run. I wonder if there's something we can do about that?

On the Pros and Cons of Different Types of Test, and Decoupling ORM code

Functional tests

- Provide the best guarantee that your application really works correctly, from the point of view of the user.
- But: it's a slower feedback cycle,
- And they don't necessarily help you write clean code.

Integrated tests (reliant on, eg, the ORM or the Django Test Client)

- Are quick to write,
- Easy to understand,
- Will warn you of any integration issues,
- But may not always drive good design (that's up to you!).
- And are usually slower than isolated tests

Isolated ("mocky") tests

- These involve the most hard work.
- They can be harder to read and understand,
- But: these are the best ones for guiding you towards better design.
- And they run the fastest.

Decoupling our application from ORM code

When striving to write isolated tests, one of the consequences is that we find ourselves forced to remove ORM code from places like views and forms, by hiding it behind helper functions or methods. This can be beneficial in terms of decoupling your application from the ORM, but also just because it makes your code more readable. As with all things, it's a judgement call as to whether the additional effort is worth it in particular circumstances.

CHAPTER 20

Continuous Integration (CI)

As our site grows, it takes longer and longer to run all of our functional tests. If this continues, the danger is that we're going to stop bothering.

Rather than let that happen, we can automate the running of functional tests by setting up a "Continuous Integration" or CI server. That way, in day-to-day development, we can just run the FT that we're working on at that time, and rely on the CI server to run all the tests automatically, and let us know if we've broken anything accidentally. The unit tests should stay fast enough that we can keep running them every few seconds.

The CI server of choice these days is called Jenkins. It's a bit Java, a bit crashy, a bit ugly, but it's what everyone uses, and it has a great plugin ecosystem, so let's get it up and running.

Installing Jenkins

There are several hosted-CI services out there that essentially provide you with a Jenkins server, ready to go. I've come across Sauce Labs, Travis, Circle-CI, ShiningPanda, and there are probably lots more. But I'm going to assume we're installing everything on a server we control.

It's not a good idea to install Jenkins on the same server as our staging or production servers. Apart from anything else, we may want Jenkins to be able to reboot the staging server!

We'll install the latest version from the official Jenkins apt repo, because the Ubuntu default still has a few annoying bugs with locale/unicode support, and it also doesn't set itself up to listen on the public Internet by default:

```
# instructions taken from jenkins site
user@server:$ wget -q -O - http://pkg.jenkins-ci.org/debian/jenkins-ci.org.key |\
     sudo apt-key add -
user@server:$ echo deb http://pkg.jenkins-ci.org/debian binary/ | sudo tee \
    /etc/apt/sources.list.d/jenkins.list
user@server:$ sudo apt-get update
user@server:$ sudo apt-get install jenkins
```

While we're at we'll install a few other dependencies:

```
user@server:$ sudo apt-get install git firefox python3 python-virtualenv xvfb
```

At the time of writing, the shiningpanda plugin was incompatible with Python 3.4. It works fine with Python 3.3, so I recommend using a slightly older distro, so that the default Python 3 is slightly older. Ubuntu Saucy (13.10) is OK, for example, but Trusty isn't.

You should then be able to visit it at the URL for your server on port `8080`, as in Figure 20-1.

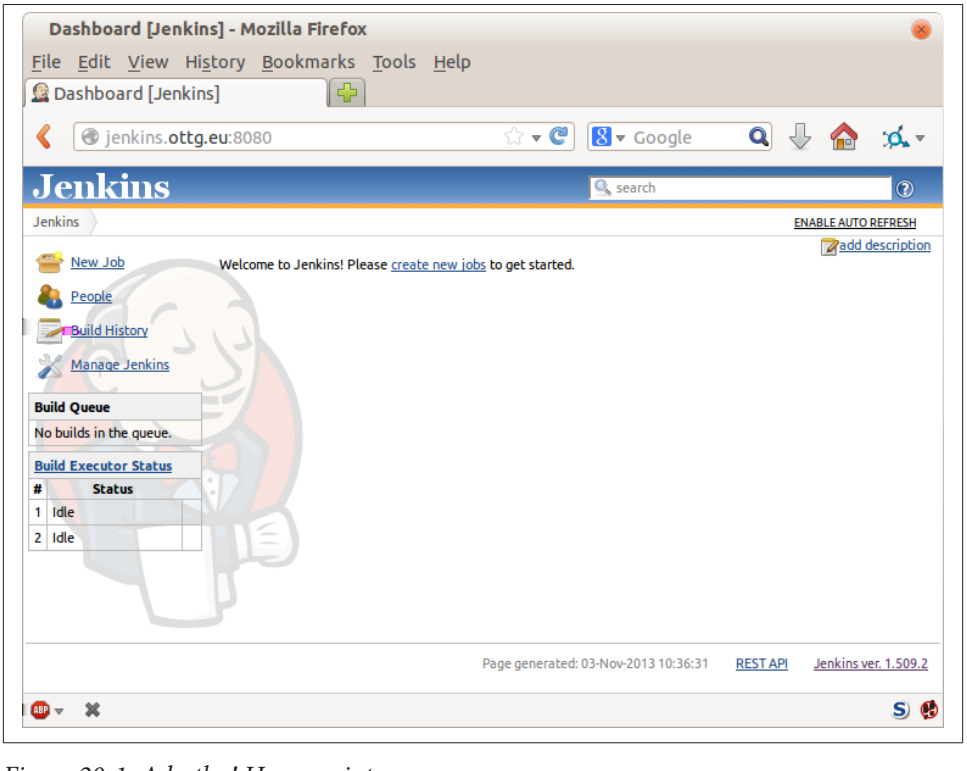


Figure 20-1. A butler! How quaint…

Configuring Jenkins Security

The first thing we'll do is set up some authentication, since our server is available on the public Internet:

- Manage Jenkins → Configure Global Security → Enable security.
- Choose "Jenkins' own user database", "Matrix-based security".
- Disable all permissions for Anonymous.
- And add a user for yourself; give it all the permissions (Figure 20-2).
- The next screen offers you the option to create an account that matches that username, and set a password.[1]

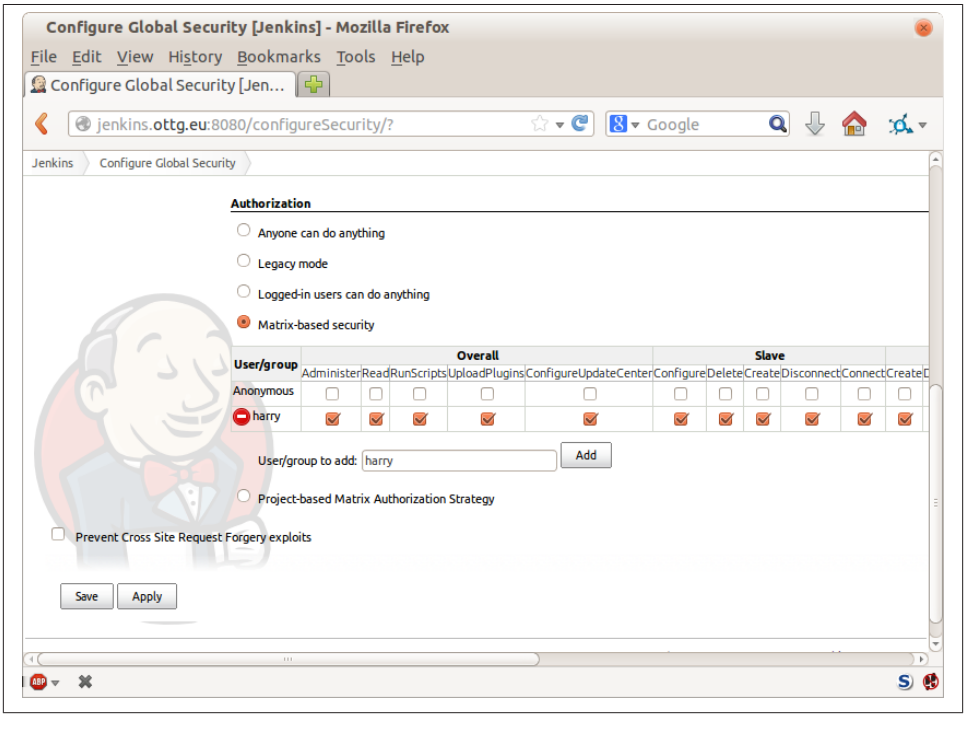


Figure 20-2. Locking it down…

1. If you miss that screen, you can still hit "signup", and as long as you use the same username you specified earlier, you'll have an account set up.

Adding Required Plugins

Next we install a few plugins, to help us work with Git, Python, and virtual displays; see Figure 20-3:

- Manage Jenkins → Manage Plugins → Available

We'll want the plugins for:

- Git
- ShiningPanda
- Xvfb

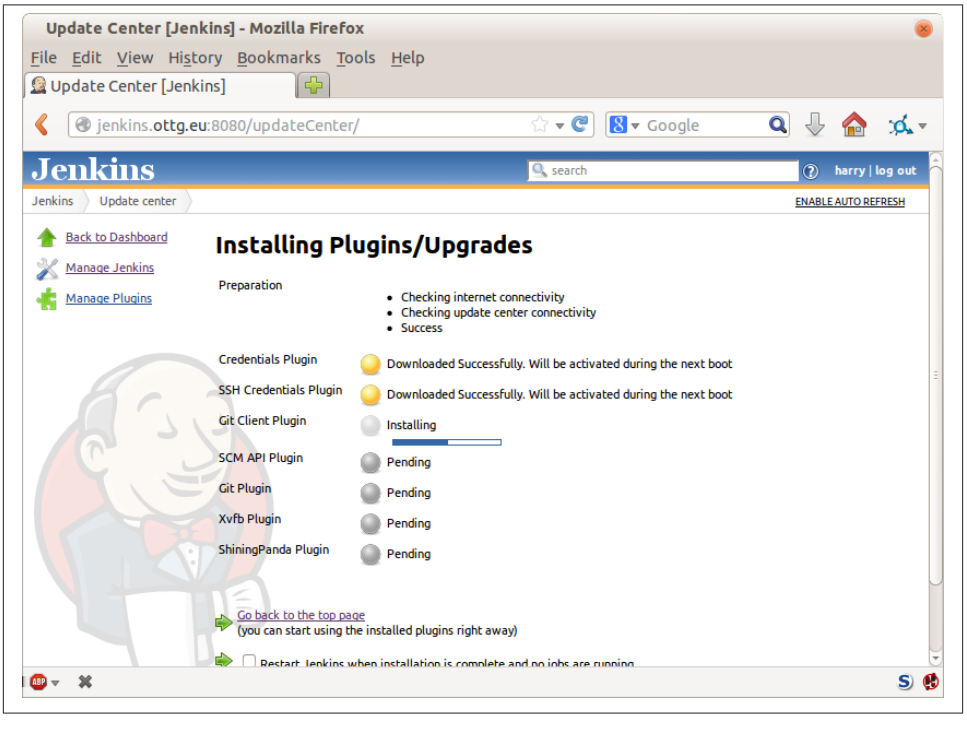


Figure 20-3. Installing plugins…

Restart afterwards using either the tick-box on that last screen, or from the command line with a `sudo service jenkins restart`.

Telling Jenkins where to find Python 3 and Xvfb

We need to tell the ShiningPanda plugin where Python 3 is installed (usually */usr/bin/python3*, but you can check with a `which python3`): * Manage Jenkins → Configure System.

- Python → Python installations → Add Python (Figure 20-4).
- Xvfb installation → Add Xvfb installation; enter **`/usr/bin`** as the installation directory.

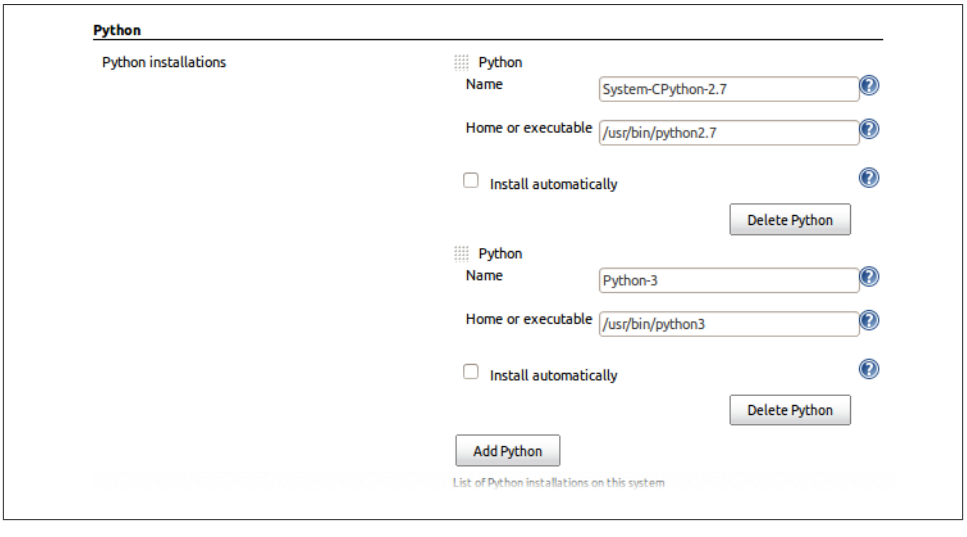


Figure 20-4. Where did I leave that Python?

Setting Up Our Project

Now we've got the basic Jenkins configured, let's set up our project:

- New Job → Build a free-style software project.
- Add the Git repo, as in Figure 20-5.

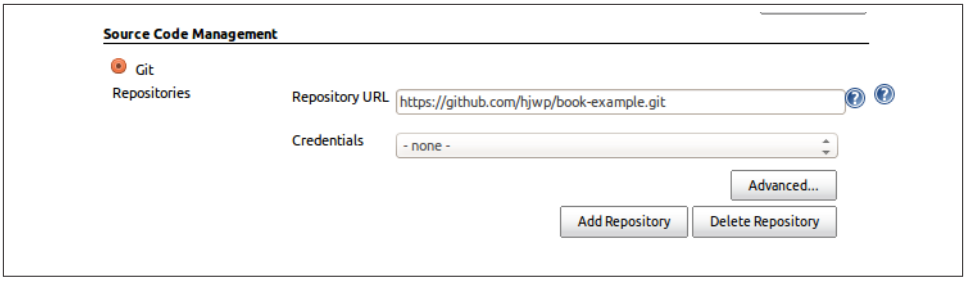


Figure 20-5. Get it from Git

- Set it to poll every hour (Figure 20-6) (check out the help text here—there are many other options for ways of triggering builds).

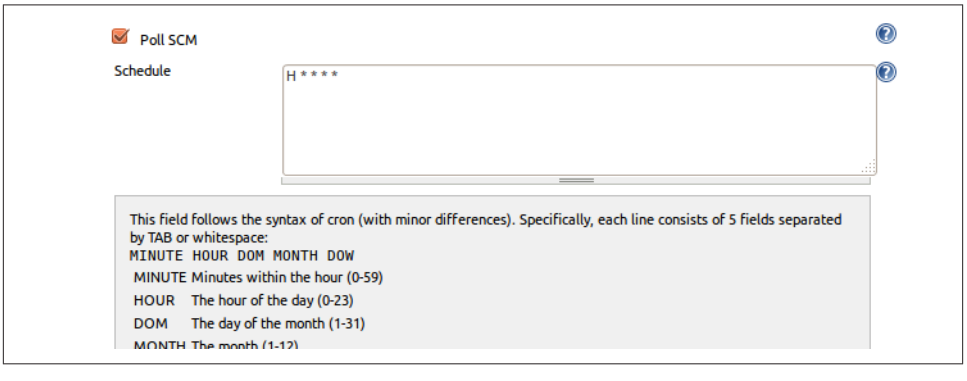


Figure 20-6. Poll Github for changes

- Run the tests inside a Python 3 virtualenv.
- Run the unit tests and functional tests separately. See Figure 20-7.

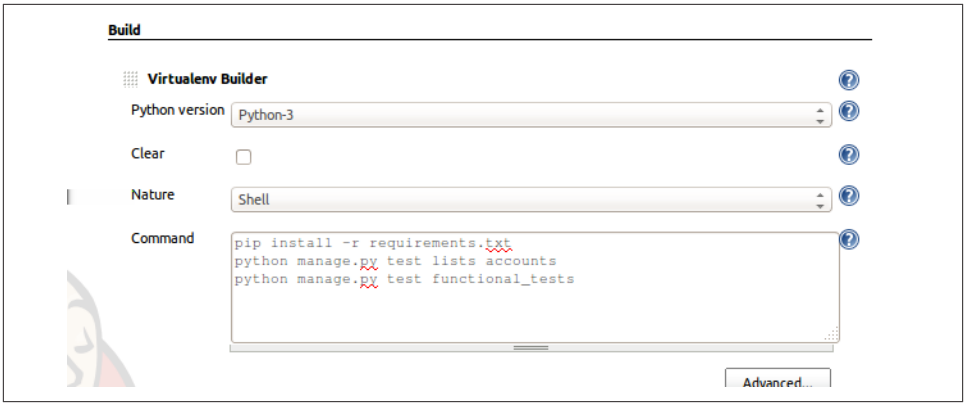


Figure 20-7. Virtualenv build steps

First Build!

Hit "Build Now!", then go and take a look at the "Console Output". You should see something like this:

```
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==1.7 in
/var/lib/jenkins/shiningpanda/jobs/ddc1aed1/virtualenvs/d41d8cd9/lib/python3.3/site-packages
(from -r requirements.txt (line 1))
Downloading/unpacking South==0.8.2 (from -r requirements.txt (line 2))
  Running setup.py egg_info for package South

Requirement already satisfied (use --upgrade to upgrade): gunicorn==17.5 in
/var/lib/jenkins/shiningpanda/jobs/ddc1aed1/virtualenvs/d41d8cd9/lib/python3.3/site-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: South, requests
  Running setup.py install for South

  Running setup.py install for requests

Successfully installed South requests
Cleaning up...
+ python manage.py test lists accounts
.................................................
 ----------------------------------------------------------------------
Ran 51 tests in 0.323s

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

```
Destroying test database for alias 'default'...
+ python manage.py test functional_tests
ImportError: No module named 'selenium'
Build step 'Virtualenv Builder' marked build as failure
```

Ah. We need Selenium in our virtualenv.

Let's add a manual installation of Selenium to our build steps:[2]

```
pip install -r requirements.txt
pip install selenium==2.39
python manage.py test accounts lists
python manage.py test functional_tests
```

Some people like to use a file called *test-requirements.txt* to specify packages that are needed for the tests, but not the main app.

Now what?

```
    File
    "/var/lib/jenkins/shiningpanda/jobs/ddc1aed1/virtualenvs/d41d8cd9/lib/python3.
    line 100, in _wait_until_connectable
      self._get_firefox_output())
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"'
```

Setting Up a Virtual Display so the FTs Can Run Headless

As you can see from the traceback, Firefox is unable to start because the server doesn't have a display.

There are two ways to deal with this problem. The first is to switch to using a headless browser, like PhantomJS or SlimerJS. Those tools definitely have their place—they're faster, for one thing—but they also have disadvantages. The first is that they're not "real" web browsers, so you can't be sure you're going to catch all the strange quirks and behaviours of the actual browsers your users use. The second is that they behave quite differently inside Selenium, and will require substantial amounts of rewriting of FT code.

2. At the time of writing, the latest Selenium (2.41) was causing me some trouble, so that's why I'm pinning it to 2.39 here. By all means experiment with newer versions!

I would look into using headless browsers as a "dev-only" tool, to speed up the running of FTs on the developer's machine, while the tests on the CI server use actual browsers.

The alternative is to set up a virtual display: we get the server to pretend it has a screen attached to it, so Firefox runs happily. There's a few tools out there to do this; we'll use one called "Xvfb" (X Virtual Framebuffer)[3] because it's easy to install and use, and because it has a convenient Jenkins plugin.

We go back to our project and hit "Configure" again, then find the section called "Build Environment". Using the virtual display is as simple as ticking the box marked "Start Xvfb before the build, and shut it down after," as in Figure 20-8.

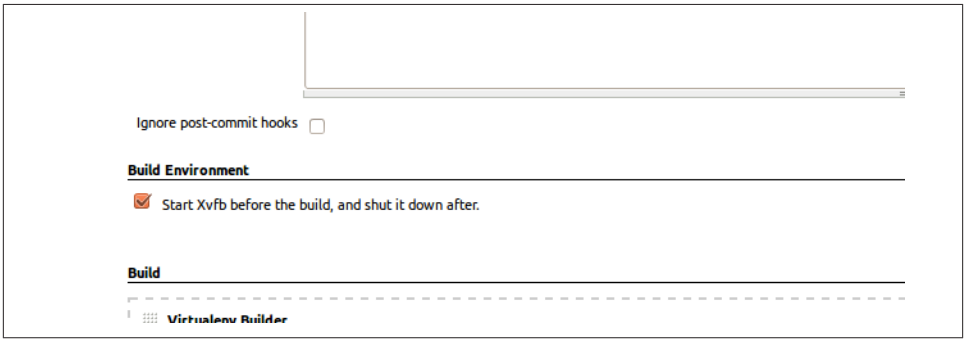


Figure 20-8. Sometimes config is easy

The build does much better now:

```
[...]
Xvfb starting$ /usr/bin/Xvfb :2 -screen 0 1024x768x24 -fbdir
/var/lib/jenkins/2013-11-04_03-27-2215100124277394709 28xvfb
[...]
+ python manage.py test lists accounts
.................................................
 ----------------------------------------------------------------------
Ran 51 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/python3.3/site-packages
```

3. Check out pyvirtualdisplay as a way of controlling virtual displays from Python.

```
Cleaning up...

+ python manage.py test functional_tests
.....F.
======================================================================
FAIL: test_logged_in_users_lists_are_saved_as_my_lists
(functional_tests.test_my_lists.MyListsTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File
"/var/lib/jenkins/jobs/Superlists/workspace/functional_tests/test_my_lists.py",
line 44, in test_logged_in_users_lists_are_saved_as_my_lists
    self.assertEqual(self.browser.current_url, first_list_url)
AssertionError: 'http://localhost:8081/accounts/edith@example.com/' !=
'http://localhost:8081/lists/1/'
- http://localhost:8081/accounts/edith@example.com/
+ http://localhost:8081/lists/1/

 ----------------------------------------------------------------------
Ran 7 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
```

Pretty close! To debug that failure, we'll need screenshots though.

As we'll see, this error is due to a race condition, which means it's not always reproducible. You may see a different error, or none at all. In any case, the tools below for taking screenshots and dealing with race conditions will come in useful. Read on!

Taking Screenshots

To be able to debug unexpected failures that happen on a remote PC, it would be good to see a picture of the screen at the moment of the failure, and maybe also a dump of the HTML of the page. We can do that using some custom logic in our FT class `tearDown`. We have to do a bit of introspection of `unittest` internals, a private attribute called `_outcomeForDoCleanups`, but this will work:

functional_tests/base.py (ch20l006).

```
import os
from datetime import datetime
SCREEN_DUMP_LOCATION = os.path.abspath(
    os.path.join(os.path.dirname(__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):
        # for 3.4. In 3.3, can just use self._outcomeForDoCleanups.success:
        for method, error in self._outcome.errors:
            if error:
                return True
        return False
```

We first create a directory for our screenshots if necessary. Then we iterate through all the open browser tabs and pages, and use some Selenium methods, `get_screen shot_as_file` and `browser.page_source`, for our image and HTML dumps:

functional_tests/base.py (ch20l007).

```
    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)
```

And finally here's a way of generating a unique filename identifier, which includes the name of the test and its class, as well as a timestamp:

functional_tests/base.py (ch20l008).

```
def _get_filename(self):
    timestamp = datetime.now().isoformat().replace(':', '.')[: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
    )
```

You can test this first locally by deliberately breaking one of the tests, with a `self.fail()` for example, and you'll see something like this:

```
[...]
screenshotting to /workspace/superlists/functional_tests/screendumps/MyListsTes
t.test_logged_in_users_lists_are_saved_as_my_lists-window0-2014-03-09T11.19.12.
png
dumping page HTML to /workspace/superlists/functional_tests/screendumps/MyLists
Test.test_logged_in_users_lists_are_saved_as_my_lists-window0-2014-03-09T11.19.
12.html
```

Revert the `self.fail()`, then commit and push:

```
$ git diff  # changes in base.py
$ echo "functional_tests/screendumps" >> .gitignore
$ git commit -am "add screenshot on failure to FT runner"
$ git push
```

And when we rerun the build on Jenkins, we see something like this:

```
screenshotting to /var/lib/jenkins/jobs/Superlists/workspace/functional_tests/
screendumps/LoginTest.test_login_with_persona-window0-2014-01-22T17.45.12.png
dumping page HTML to /var/lib/jenkins/jobs/Superlists/workspace/functional_tests/
screendumps/LoginTest.test_login_with_persona-window0-2014-01-22T17.45.12.html
```

We can go and visit these in the "workspace", which is the folder which Jenkins uses to store our source code and run the tests in, as in Figure 20-9.

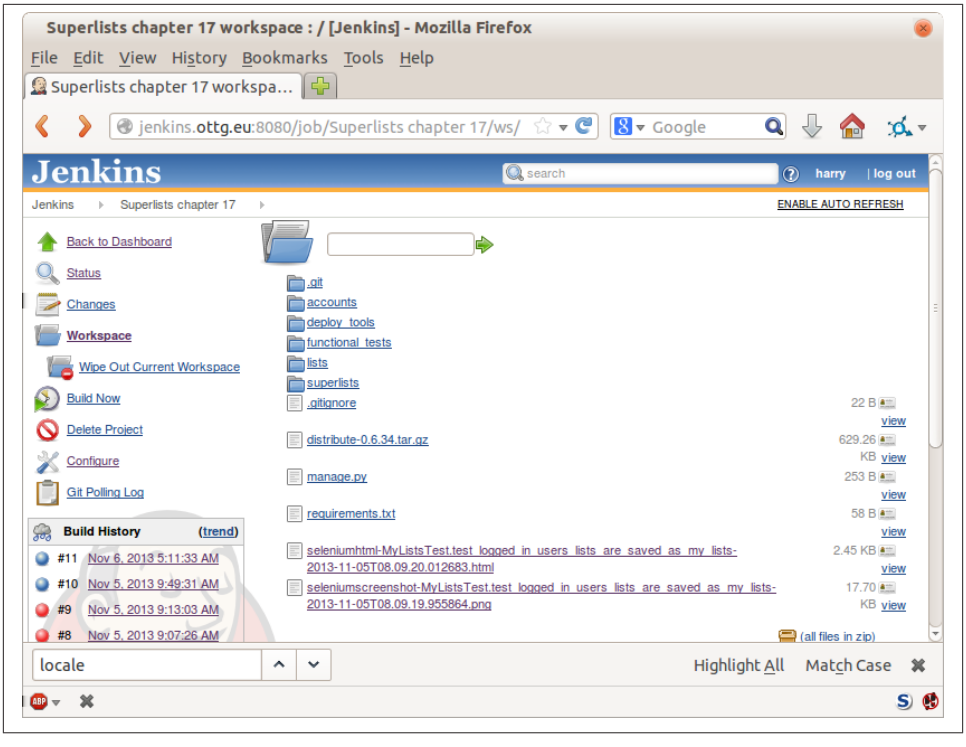


Figure 20-9. Visiting the project workspace

And then we look at the screenshot, as shown in Figure 20-10.

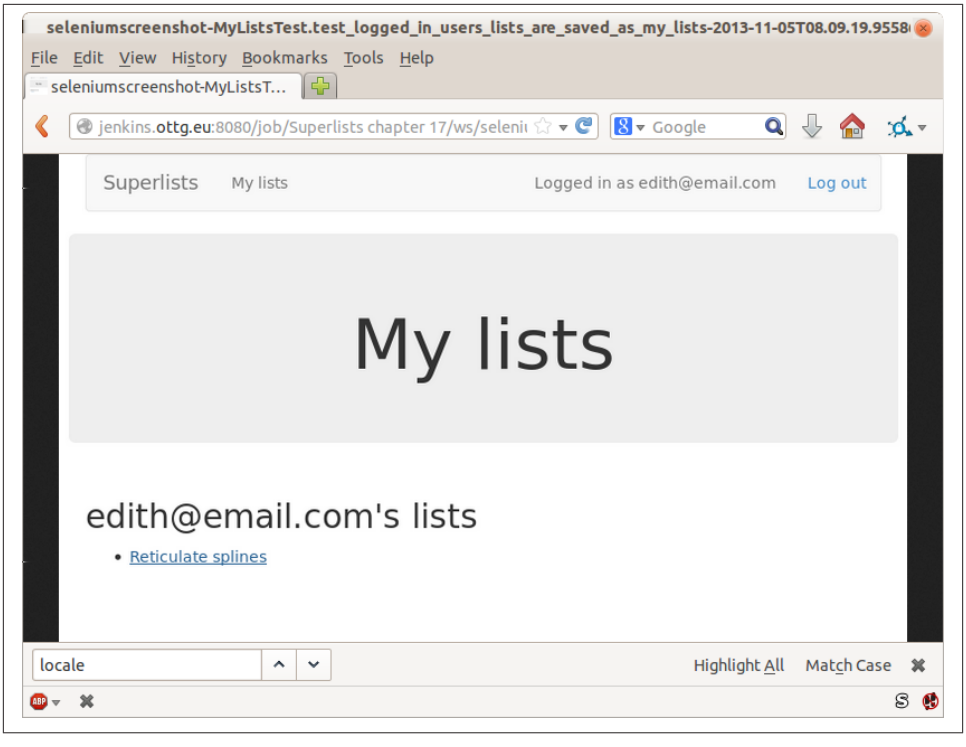


Figure 20-10. Screenshot looking normal

Well, that didn't help much.

A Common Selenium Problem: Race Conditions

Whenever you see an inexplicable failure in a Selenium test, one of the most likely explanations is a hidden race condition. Let's look at the line that failed:

functional_tests/test_my_lists.py.

```
        # She sees that her list is in there, named according to its
        # first list item
        self.browser.find_element_by_link_text('Reticulate splines').click()
        self.assertEqual(self.browser.current_url, first_list_url)
```

Immediately after we click the "Reticulate splines" link, we ask Selenium to check whether the current URL matches the URL for our first list. But it doesn't:

```
AssertionError: 'http://localhost:8081/accounts/edith@example.com/' !=
'http://localhost:8081/lists/1/'
```

It looks like the current URL is still the URL of the "My Lists" page. What's going on?

Do you remember that we set an `implicitly_wait` on the browser, way back in Chapter 2? Do you remember I mentioned it was unreliable?

`implicitly_wait` works reasonably well for any calls to any of the Selenium `find_element_` calls, but it doesn't apply to `browser.current_url`. Selenium doesn't "wait" after you tell it to click an element, so what's happened is that the browser hasn't finished loading the new page yet, so `current_url` is still the old page. We need to use some more wait code, like we did for the various Persona pages.

At this point it's time for a "wait for" helper function. To see how this is going to work, it helps to see how I expect to use it (outside-in!):

functional_tests/test_my_lists.py (ch20l012).

```
        # She sees that her list is in there, named according to its
        # first list item
        self.browser.find_element_by_link_text('Reticulate splines').click()
        self.wait_for(
            lambda: self.assertEqual(self.browser.current_url, first_list_url)
        )
```

We're going to take our `assertEqual` call and turn it into a lambda function, then pass it into our `wait_for` helper.

functional_tests/base.py (ch20l013).

```
import time
from selenium.common.exceptions import WebDriverException
[...]

    def wait_for(self, function_with_assertion, timeout=DEFAULT_WAIT):
        start_time = time.time()
        while time.time() - start_time < timeout:
            try:
                return function_with_assertion()
            except (AssertionError, WebDriverException):
                time.sleep(0.1)
        # one more try, which will raise any errors if they are outstanding
        return function_with_assertion()
```

`wait_for` then tries to execute that function, but instead of letting the test fail if the assertion fails, it catches the `AssertionError` that `assertEqual` would ordinarily raise, waits for a brief moment, and then loops around retrying it. The `while` loop lasts until a given timeout. It also catches any `WebDriverException` that might happen if, say, an element hasn't appeared on the page yet. It tries one last time after the timeout has expired, this time without the `try/except`, so that if there really is still an `AssertionError`, the test will fail appropriately.

We've seen that Selenium provides `WebdriverWait` as a tool for doing waits, but it's a little restrictive. This hand-rolled version lets us pass a function that does a `unittest` assertion, with all the benefits of the readable error messages that it gives us.

I've added the timeout there as an optional argument, and I'm basing it on a constant we'll add to *base.py*. We'll also use it in our original `implicitly_wait`:

functional_tests/base.py (ch20l014).

```
[...]
DEFAULT_WAIT = 5
SCREEN_DUMP_LOCATION = os.path.abspath(
    os.path.join(os.path.dirname(__file__), 'screendumps')
)


class FunctionalTest(StaticLiveServerCase):

    [...]

    def setUp(self):
        self.browser = webdriver.Firefox()
        self.browser.implicitly_wait(DEFAULT_WAIT)
```

Now we can rerun the test to confirm it still works locally:

```
$ python3 manage.py test functional_tests.test_my_lists
[...]
.

Ran 1 test in 9.594s

OK
```

And, just to be sure, we'll deliberately break our test to see it fail too:

functional_tests/test_my_lists.py (ch20l015).

```
        self.wait_for(
            lambda: self.assertEqual(self.browser.current_url, 'barf')
        )
```

Sure enough, that gives:

```
$ python3 manage.py test functional_tests.test_my_lists
[...]
AssertionError: 'http://localhost:8081/lists/1/' != 'barf'
```

And we see it pause on the page for three seconds. Let's revert that last change, and then commit our changes:

```
$ git diff # base.py, test_my_lists.py
$ git commit -am"use wait_for function for URL checks in my_lists"
$ git push
```

Then we can rerun the build on Jenkins using "Build now", and confirm it now works, as in Figure 20-11.

Figure 20-11. The outlook is brighter

Jenkins uses blue to indicate passing builds rather than green, which is a bit disappointing, but look at the sun peeking through the clouds: that's cheery! It's an indicator of a moving average ratio of passing builds to failing builds. Things are looking up!

Running Our QUnit JavaScript Tests in Jenkins with PhantomJS

There's a set of tests we almost forgot—the JavaScript tests. Currently our "test runner" is an actual web browser. To get Jenkins to run them, we need a command-line test runner. Here's a chance to use PhantomJS.

Installing node

It's time to stop pretending we're not in the JavaScript game. We're doing web development. That means we do JavaScript. That means we're going to end up with node.js on our computers. It's just the way it has to be.

Follow the instructions on the node.js download page. There are installers for Windows and Mac, and repositories for popular Linux distros.[4]

Once we have node, we can install phantom:

```
$ npm install -g phantomjs  # the -g means "system-wide". May need sudo.
```

Next we pull down a QUnit/PhantomJS test runner. There are several out there (I even wrote a basic one to be able to test the QUnit listings in this book), but the best one to get is probably the one that's linked from the QUnit plugins page. At the time of writing, its repo was at *https://github.com/jonkemp/qunit-phantomjs-runner*. The only file you need is *runner.js*.

You should end up with this:

```
$ tree superlists/static/tests/
superlists/static/tests/
├── qunit.css
├── qunit.js
├── runner.js
└── sinon.js

0 directories, 4 files
```

Let's try it out:

```
$ phantomjs superlists/static/tests/runner.js lists/static/tests/tests.html
Took 24ms to run 2 tests. 2 passed, 0 failed.
$ phantomjs superlists/static/tests/runner.js accounts/static/tests/tests.html
Took 29ms to run 11 tests. 11 passed, 0 failed.
```

Just to be sure, let's deliberately break something:

lists/static/list.js (ch20l019).

```
$('input').on('keypress', function () {
    //$('.has-error').hide();
});
```

Sure enough:

```
$ phantomjs superlists/static/tests/runner.js lists/static/tests/tests.html
Test failed: undefined: errors should be hidden on keypress
    Failed assertion: expected: false, but was: true
    at file:///workspace/superlists/superlists/static/tests/qunit.js:556
    at file:///workspace/superlists/lists/static/tests/tests.html:26
```

4. Make sure you get the latest version. On Ubuntu, use the PPA rather than the default package.

```
    at file:///workspace/superlists/superlists/static/tests/qunit.js:203
    at file:///workspace/superlists/superlists/static/tests/qunit.js:361
    at process
(file:///workspace/superlists/superlists/static/tests/qunit.js:1453)
    at file:///workspace/superlists/superlists/static/tests/qunit.js:479
Took 27ms to run 2 tests. 1 passed, 1 failed.
```

All right! Let's unbreak that, commit and push the runner, and then add it to our Jenkins build:

```
$ git checkout lists/static/list.js
$ git add superlists/static/tests/runner.js
$ git commit -m"Add phantomjs test runner for javascript tests"
$ git push
```

Adding the Build Steps to Jenkins

Edit the project configuration again, and add a step for each set of JavaScript tests, as per Figure 20-12.

Figure 20-12. Add a build step for our JavaScript unit tests

You'll also need to install PhantomJS on the server:

```
elspeth@server:$ sudo add-apt-repository -y ppa:chris-lea/node.js
elspeth@server:$ sudo apt-get update
elspeth@server:$ sudo apt-get install nodejs
elspeth@server:$ sudo npm install -g phantomjs
```

And there we are! A complete CI build featuring all of our tests!

```
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 33 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 superlists/static/tests/runner.js lists/static/tests/tests.html
Took 32ms to run 2 tests. 2 passed, 0 failed.
+ phantomjs superlists/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 7 tests in 76.804s

OK
```

Nice to know that, no matter how lazy I get about running the full test suite on my own machine, the CI server will catch me. Another one of the Testing Goat's agents in cyberspace, watching over us…

More Things to Do with a CI Server

I've only scratched the surface of what you can do with Jenkins and CI servers. For example, you can make it much smarter about how it monitors your repo for new commits.

Perhaps more interestingly, you can use your CI server to automate your staging tests as well as your normal functional tests. If all the FTs pass, you can add a build step that

deploys the code to staging, and then reruns the FTs against that—automating one more step of the process, and ensuring that your staging server is automatically kept up to date with the latest code.

Some people even use a CI server as the way of deploying their production releases!

Tips on CI and Selenium Best Practices

Set up CI as soon as possible for your project
: As soon as your functional tests take more than a few seconds to run, you'll find yourself avoiding running them all. Give this job to a CI server, to make sure that all your tests are getting run somewhere.

Set up screenshots and HTML dumps for failures
: Debugging test failures is easier if you can see what the page looked at when the failure occurs. This is particularly useful for debugging CI failures, but it's also very useful for tests that you run locally.

Use waits in Selenium tests
: Selenium's `implicitly_wait` only applies to uses of its `find_element` functions, and even that can be unreliable (it can find an element that's still on the old page). Build a `wait_for` helper function, and alternate between actions on the site, and then some sort of wait to see that they've taken effect.

Look in to hooking up CI and staging
: Tests that use `LiveServerTestCase` are all very well for dev boxes, but the true reassurance comes from running your tests against a real server. Look into getting your CI server to deploy to your staging server, and run the functional tests against that instead. It has the side benefit of testing your automated deploy scripts.

CHAPTER 21

The Token Social Bit, the Page Pattern, and an Exercise for the Reader

Are jokes about how "everything has to be social now" slightly old hat? Everything has to be all A/B tested big data get-more-clicks lists of 10 Things This Inspiring Teacher Said That Will Make You Change Your Mind About Blah Blah now … anyway. Lists, be they Inspirational or otherwise, are often better shared. Let's allow our users to collaborate on their lists with other users.

Along the way we'll improve our FTs by starting to implement the interact/wait Selenium pattern that we learned in the last chapter. We'll also experiment with something called the Page Object pattern.

Then, rather than showing you explicitly what to do, I'm going to let you write your unit tests and application code by yourself. Don't worry, you won't be totally on your own! I'll give an outline of the steps to take, as well as some hints and tips.

An FT with Multiple Users, and addCleanup

Let's get started—we'll need two users for this FT:

functional_tests/test_sharing.py.

```
from selenium import webdriver
from .base import FunctionalTest

def quit_if_possible(browser):
    try: browser.quit()
    except: pass


class SharingTest(FunctionalTest):

    def test_logged_in_users_lists_are_saved_as_my_lists(self):
        # Edith is a logged-in user
        self.create_pre_authenticated_session('edith@example.com')
```

```
        edith_browser = self.browser
        self.addCleanup(lambda: quit_if_possible(edith_browser))

        # Her friend Oniciferous is also hanging out on the lists site
        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 goes to the home page and starts a list
        self.browser = edith_browser
        self.browser.get(self.server_url)
        self.get_item_input_box().send_keys('Get help\n')

        # She notices a "Share this list" option
        share_box = self.browser.find_element_by_css_selector('input[name=email]')
        self.assertEqual(
            share_box.get_attribute('placeholder'),
            'your-friend@example.com'
        )
```

The interesting feature to note about this section is the `addCleanup` function, whose documentation you can find here. It can be used as an alternative to the `tearDown` function as a way of cleaning up resources used during the test. It's most useful when the resource is only allocated halfway through a test, so you don't have to spend time in `tearDown` figuring out what does or doesn't need cleaning up.

`addCleanup` is run after `tearDown`, which is why we need that `try/except` formulation for `quit_if_possible`—whichever of `edith_browser` and `oni_browser` is also assigned to `self.browser` at the point at which the test ends will already have been quit by the `tearDown` function.

We'll also need to move `create_pre_authenticated_session` from *test_my_lists.py* into *base.py*.

OK, let's see if that all works:

```
$ python3 manage.py test functional_tests.test_sharing
[...]
Traceback (most recent call last):
  File "/workspace/superlists/functional_tests/test_sharing.py", line 29, in
test_logged_in_users_lists_are_saved_as_my_lists
    share_box = self.browser.find_element_by_css_selector('input[name=email]')
[...]
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"css selector","selector":"input[name=email]"}' ;
```

Great! It seems to have got through creating the two user sessions, and it gets onto an expected failure—there is no input for an email address of a person to share a list with on the page.

Let's do a commit at this point, because we've got at least a placeholder for our FT, we've got a useful modification of the `create_pre_authenticated_session` function, and we're about to embark on a bit of an FT refactor:

```
$ git add functional_tests
$ git commit -m "New FT for sharing, move session creation stuff to base"
```

Implementing the Selenium Interact/Wait Pattern

Before we continue, let's take a closer look at the interactions with the site which we have in our FT so far:

functional_tests/test_sharing.py

```
        # Edith goes to the home page and starts a list
        self.browser = edith_browser
        self.browser.get(self.server_url)
        self.get_item_input_box().send_keys('Get help\n')  #❶

        # She notices a "Share this list" option
        share_box = self.browser.find_element_by_css_selector('input[name=email]')  #❷
        self.assertEqual(
            share_box.get_attribute('placeholder'),
            'your-friend@example.com'
        )
```

❶ Interaction with site

❷ Assumption about updated state of page

We learned in the last chapter that it's dangerous to assume too much about the state of the browser after we do an interaction (like `send_keys`). In theory, `implicitly_wait` will make sure that, if the call to `find_element_by_css_selector` doesn't find our `input[name=email]` at first, it will silently retry a few times. But it can also go wrong—imagine if there was an input on the previous page, with the same `name=email`, but a different placeholder text? We'd get a strange failure, because Selenium could theoretically pick up the element from the previous page while the new page is loading. That tends to raise a `StaleElementException`.

Unexpected `StaleElementException` errors from Selenium often mean you have some kind of race condition. You should probably specify an explicit interaction/wait pattern.

Instead, it's always prudent to follow the "wait-for" pattern whenever we want to check on the effects of an interaction that we've just triggered. Something like this:

functional_tests/test_sharing.py

```
        self.get_item_input_box().send_keys('Get help\n')
```

```
        # She notices a "Share this list" option
        self.wait_for(
            lambda:  self.assertEqual(
                self.browser.find_element_by_css_selector(
                    'input[name=email]'
                ).get_attribute('placeholder'),
                'your-friend@example.com'
            )
        )
```

The Page Pattern

But do you know what would be even better? This is an occasion for a "three strikes and refactor". This test, and many others, all begin off with the user starting a new list. What if we had a helper function called "start new list" that would do the `wait_for` as well as the list item input?

We've already seen how to use helper methods on the base `FunctionalTest` class, but if we continue using too many of them, it's going to get very crowded. I've worked on a base FT class that was over 1,500 lines long, and that got pretty unwieldy.

One accepted pattern for splitting up your FT helper code is called the Page pattern, and it involves having objects to represent the different pages on your site, and to be the single place to store information about how to interact with them.

Let's see how we would create Page objects for the home and lists pages. Here's one for the home page:

functional_tests/home_and_list_pages.py.

```
class HomePage(object):

    def __init__(self, test):
        self.test = test  #❶


    def go_to_home_page(self):  #❷
        self.test.browser.get(self.test.server_url)
        self.test.wait_for(self.get_item_input)
        return self  #❸


    def get_item_input(self):
        return self.test.browser.find_element_by_id('id_text')


    def start_new_list(self, item_text):  #❹
        self.go_to_home_page()
        inputbox = self.get_item_input()
        inputbox.send_keys(item_text + '\n')
        list_page = ListPage(self.test)  #❺
```

```
        list_page.wait_for_new_item_in_list(item_text, 1)  #❻
        return list_page  #❼
```

❶ It's initialised with an object that represents the current test. That gives us the ability to make assertions, access the browser instance via `self.test.browser`, and use the `wait_for` function.

❷ Most Page objects have a "go to this page" function. Notice that it implements the interaction/wait pattern—first we `get` the page URL, then we wait for an element that we know is on the home page.

❸ Returning `self` is just a convenience. It enables method chaining.

❹ Here's our function that starts a new list. It goes to the home page, finds the input box, and sends the new item text to it, as well as a carriage return. Then it uses a wait to check that the interaction has completed, but as you can see that wait is actually on a different Page object:

❺ The `ListPage`, which we'll see the code for shortly. It's initialised just like a `HomePage`.

❻ We use the `ListPage` to `wait_for_new_item_in_list`. We specify the expected text of the item, and its expected position in the list.

❼ Finally, we return the `list_page` object to the caller, because they will probably find it useful (as we'll see shortly).

Here's how `ListPage` looks:

functional_tests/home_and_list_pages.py (ch21l006).

```
[...]

class ListPage(object):

    def __init__(self, test):
        self.test = test

    def get_list_table_rows(self):
        return self.test.browser.find_elements_by_css_selector(
            '#id_list_table tr'
        )

    def wait_for_new_item_in_list(self, item_text, position):
        expected_row = '{}: {}'.format(position, item_text)
        self.test.wait_for(lambda: self.test.assertIn(
            expected_row,
            [row.text for row in self.get_list_table_rows()]
        ))
```

It's usually best to have a separate file for each Page object. In this case, `HomePage` and `ListPage` are so closely related it's easier to keep them together.

Let's see how to use it in our test:

functional_tests/test_sharing.py (ch21l007).

```
from .home_and_list_pages import HomePage
[...]

        # Edith goes to the home page and starts a list
        self.browser = edith_browser
        list_page = HomePage(self).start_new_list('Get help')
```

Let's continue rewriting our test, using the Page object whenever we want to access elements from the lists page:

functional_tests/test_sharing.py (ch21l008).

```
        # She notices a "Share this list" option
        share_box = list_page.get_share_box()
        self.assertEqual(
            share_box.get_attribute('placeholder'),
            'your-friend@example.com'
        )

        # She shares her list.
        # The page updates to say that it's shared with Oniciferous:
        list_page.share_list_with('oniciferous@example.com')
```

We add the following three functions to our `ListPage`:

functional_tests/home_and_list_pages.py (ch21l009).

```
    def get_share_box(self):
        return self.test.browser.find_element_by_css_selector(
            'input[name=email]'
        )


    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 + '\n')
        self.test.wait_for(lambda: self.test.assertIn(
            email,
            [item.text for item in self.get_shared_with_list()]
        ))
```

The idea behind the Page pattern is that it should capture all the information about a particular page in your site, so that if, later, you want to go and make changes to that page—even just simple tweaks to its HTML layout for example—you have a single place to go and look for to adjust your functional tests, rather than having to dig through dozens of FTs.

The next step would be to pursue the FT refactor through our other tests. I'm not going to show that here, but it's something you could do, for practice, to get a feel for what the trade-offs between D.R.Y. and test readability are like…

Extend the FT to a Second User, and the "My Lists" Page

Let's spec out just a little more detail of what we want our sharing user story to be. Edith has seen on her list page that the list is now "shared with" Oniciferous, and then we can have Oni log in and see the list on his "My Lists" page, maybe in a section called "lists shared with me":

functional_tests/test_sharing.py (ch21l010).

```
        [...]
        list_page.share_list_with('oniciferous@example.com')

        # Oniciferous now goes to the lists page with his browser
        self.browser = oni_browser
        HomePage(self).go_to_home_page().go_to_my_lists_page()

        # He sees Edith's list in there!
        self.browser.find_element_by_link_text('Get help').click()
```

That means another function in our `HomePage` class:

functional_tests/home_and_list_pages.py (ch21l011).

```
class HomePage(object):

    [...]

    def go_to_my_lists_page(self):
        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'
        ))
```

Once again, this is a function that would be good to carry across into *test_my_lists.py*, along with maybe a `MyListsPage` object. Exercise for the reader!

In the meantime, Oniciferous can also add things to the list:

functional_tests/test_sharing.py (ch21l012).

```
        # On the list page, Oniciferous can see says that it's Edith's list
        self.wait_for(lambda: self.assertEqual(
            list_page.get_list_owner(),
            'edith@example.com'
```

```
        ))

        # He adds an item to the list
        list_page.add_new_item('Hi Edith!')

        # When Edith refreshes the page, she sees Oniciferous's addition
        self.browser = edith_browser
        self.browser.refresh()
        list_page.wait_for_new_item_in_list('Hi Edith!', 2)
```

That's a couple more additions to our Page object:

functional_tests/home_and_list_pages.py (ch21l013).

```
ITEM_INPUT_ID = 'id_text'
[...]

class HomePage(object):
    [...]

    def get_item_input(self):
        return self.test.browser.find_element_by_id(ITEM_INPUT_ID)


class ListPage(object):
    [...]

    def get_item_input(self):
        return self.test.browser.find_element_by_id(ITEM_INPUT_ID)


    def add_new_item(self, item_text):
        current_pos = len(self.get_list_table_rows())
        self.get_item_input().send_keys(item_text + '\n')
        self.wait_for_new_item_in_list(item_text, current_pos + 1)


    def get_list_owner(self):
        return self.test.browser.find_element_by_id('id_list_owner').text
```

It's long past time to run the FT and check if all of this works!

```
$ python3 manage.py test functional_tests.test_sharing

    share_box = list_page.get_share_box()
    [...]
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"css selector","selector":"input[name=email]"}' ;
```

That's the expected failure; we don't have an input for email addresses of people to share with. Let's do a commit:

```
$ git add functional_tests
$ git commit -m "Create Page objects for Home and List pages, use in sharing FT"
```

An Exercise for the Reader

Here's an outline of the steps to take to get this new feature implemented:

1. We'll need a new section in *list.html*, with, at first, a form with an input box for an email address. That should get the FT one step further.
2. Next, we'll need a view for the form to submit to. Start by defining the URL in the template, maybe something like *lists/<list_id>/share*.
3. Then, our first unit test. It can be just enough to get a placeholder view in. We want the view to respond to POST requests, and it should respond with a redirect back to the list page, so the test could be called something like `ShareListTest.test_post_redirects_to_lists_page`.
4. We build out our placeholder view, as just a two-liner that finds a list and redirects to it.
5. We can then write a new unit test which creates a user and a list, does a POST with their email address, and checks the user is added to `list_.shared_with.all()` (a similar ORM usage to "My Lists"). That `shared_with` attribute won't exist yet, we're going outside-in.
6. So before we can get this test to pass, we have to move down to the model layer. The next test, in *test_models.py*, can check that a list has a `shared_with.add` method, which can be called with a user's email address and then check the lists' `shared_with.all()` queryset, which will subsequently contain that user.
7. You'll then need a `ManyToManyField`. You'll probably see an error message about a clashing `related_name`, which you'll find a solution to if you look around the Django docs.
8. It will need a database migration.
9. That should get the model tests passing. Pop back up to fix the view test.
10. You may find the redirect view test fails, because it's not sending a valid POST request. You can either choose to ignore invalid inputs, or adjust the test to send a valid POST.
11. Then back up to the template level; on the "My Lists" page we'll want a `<ul>` with a for loop of the lists shared with the user. On the lists page, we also want to show who the list is shared with, as well as mention of who the list owner is. Look back at the FT for the correct classes and IDs to use. You could have brief unit tests for each of these if you like, as well.
12. You might find that spinning up the site with `runserver` will help you iron out any bugs, as well as fine-tune the layout and aesthetics. If you use a private browser session, you'll be able to log multiple users in.

By the end, you might end up with something that looks like Figure 21-1.

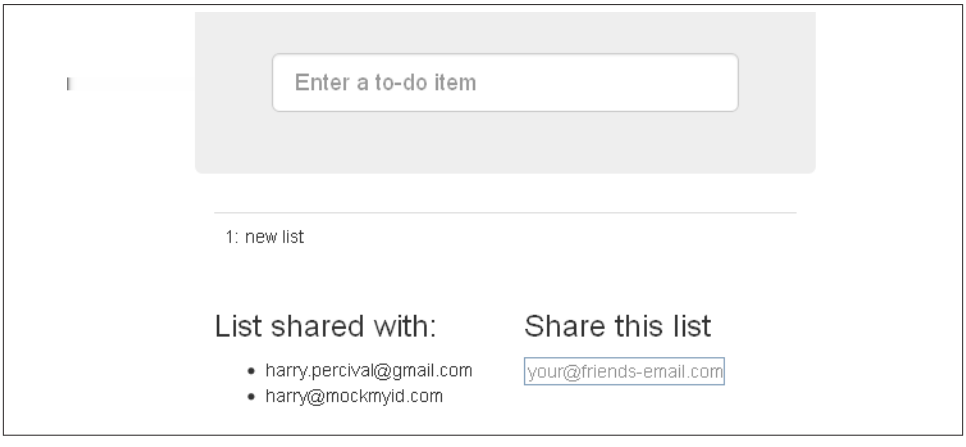


Figure 21-1. Sharing lists

The Page Pattern, and the Real Exercise for the Reader

Apply DRY to your functional tests
: Once your FT suite starts to grow, you'll find that different tests will inevitably find themselves using similar parts of the UI. Try to avoid having constants, like the HTML IDs or classes of particular UI elements duplicated between your FTs.

The Page pattern
: Moving helper methods into a base `FunctionalTest` class can become unwieldy. Consider using individual Page objects to hold all the logic for dealing with particular parts of your site.

An exercise for the reader
: I hope you've actually tried this out! Try to follow the "Outside-In" method, and occasionally try things out manually if you get stuck. The real exercise for the reader, of course, is to apply TDD to your next project. I hope you'll enjoy it!

In the next chapter, we'll wrap up with a discussion of testing "best practices".

CHAPTER 22

Fast Tests, Slow Tests, and Hot Lava

The database is Hot Lava!

— Casey Kinsey

Right up until Chapter 19, almost all of the "unit" tests in the book should perhaps have been called *integrated* tests, because they either rely on the database, or they use the Django test client, which does too much magic with the middleware layers that sit between requests, responses, and view functions.

There is an argument that a true unit test should always be isolated, because it's meant to test a single unit of software.

Some TDD veterans say you should strive to write "pure", isolated unit tests wherever possible, instead of writing integrated tests. It's one of the ongoing (occasionally heated) debates in the testing community.

Being merely a young whippersnapper myself, I'm only part way towards all the subtleties of the argument. But in this chapter, I'd like to try and talk about why people feel strongly about it, and try and give you some idea of when you can get away with muddling through with integrated tests (which I confess I do a lot of!), and when it's worth striving for more "pure" unit tests.

Terminology: Different Types of Test

Isolated tests ("pure" unit tests) vs. integrated tests

The primary purpose of a unit test should be to verify the correctness of the logic of your application. An *isolated* test is one that tests exactly one chunk of code, and whose success or failure does not depend on any other external code. This is what I call a "pure" unit test: a test for a single function, for example, written in such a way that only that function can make it fail. If the function depends on another system, and breaking that system breaks our test, we have an *integrated* test. That

system could be an external system, like a database, but it could also be another function which we don't control. In either case, if breaking the system makes our test fail, our test is not properly isolated, it is not a "pure" unit test. That's not necessarily a bad thing, but it may mean the test is doing two jobs at once.

Integration tests
: An integration test checks that the code you control is integrated correctly with some external system which you don't control. *Integration* tests are typically also *integrated* tests.

System tests
: If an integration test checks the integration with one external system, a system test checks the integration of multiple systems in your application—for example, checking that we've wired up our database, static files, and server config together in such a way that they all work.

Functional tests and acceptance tests
: An acceptance test is meant to test that our system works from the point of view of the user ("would the user accept this behaviour?"). It's hard to write an acceptance test that's not a full-stack, end-to-end test. We've been using our functional tests to play the role of both acceptance tests and system tests.

If you'll forgive me the pretentious philosophical terminology, I'm going to follow a Hegelian dialectical structure:

- The Thesis: the case for "pure" unit tests that are fast.
- The Antithesis: some of the risks associated with a (naive) pure unit testing approach.
- The Synthesis: a discussion of best practices like "Ports and Adapters" or "Functional Core, Imperative Shell", and of just what it is that we want from our tests, anyway.

Thesis: Unit Tests Are Superfast and Good Besides That

One of the things you often hear about unit tests is that they're much faster. I don't think that's actually the primary benefit of unit tests, but it's worth exploring the theme of speed.

Faster Tests Mean Faster Development

Other things being equal, the faster your unit tests run, the better. To a lesser extent, the faster *all* your tests run, the better.

I've outlined the TDD test/code cycle in this book. You've started to get a feel for the TDD workflow, the way you flick between writing tiny amounts of code, and running your tests. You end up running your unit tests several times a minute, and your functional tests several times a day.

So, on a very basic level, the longer they take, the more time you spend waiting for your tests, and that will slow down your development. But there's more to it than that.

The Holy Flow State

Thinking sociology for a moment, we programmers have our own culture, and our own tribal religion in a way. It has many congregations within it, such as the cult of TDD to which you are now initiated. There are the followers of vi and the heretics of emacs. But one thing we all agree on, one particular spiritual practice, our own transcendental meditation, is the holy flow state. That feeling of pure focus, of concentration, where hours pass like no time at all, where code flows naturally from our fingers, where problems are just tricky enough to be interesting but not so hard that they defeat us…

There is absolutely no hope of achieving flow if you spend your time waiting for a slow test suite to run. Anything longer than a few seconds and you're going to let your attention wander, you context-switch, and the flow state is gone. And the flow state is a fragile dream. Once it's gone, it takes at least 15 minutes to live again.

Slow Tests Don't Get Run as Often, Which Causes Bad Code

If your test suite is slow and ruins your concentration, the danger is that you'll start to avoid running your tests, which may lead to bugs getting through. Or, it may lead to our being shy of refactoring the code, since we know that any refactor will mean having to wait ages while all the tests run. In either case, bad code can be the result.

We're Fine Now, but Integrated Tests Get Slower Over Time

You might be thinking, OK, but our test suite has lots of integrated tests in it—over 50 of them, and it only takes 0.2 seconds to run.

But remember, we've got a very simple app. Once it starts to get more complex, as your database grows more and more tables and columns, integrated tests will get slower and slower. Having Django reset the database between each test will take longer and longer.

Don't Take It from Me

Gary Bernhardt, a man with far more experience of testing than me, put these points eloquently in a talk called Fast Test, Slow Test. I encourage you to watch it.

And Unit Tests Drive Good Design

But perhaps more importantly than any of this, remember the lesson from Chapter 19. Going through the process of writing good, isolated unit tests can help us drive out better designs for our code, by forcing us to identify dependencies, and encouraging us towards a decoupled architecture in a way that integrated tests don't.

The Problems with "Pure" Unit Tests

All of this comes with a huge "but". Writing isolated united tests comes with its own hazards, particularly if, like you or I, we are not yet advanced TDD'ers.

Isolated Tests Can Be Harder to Read and Write

Cast your mind back to the first isolated unit test we wrote. Wasn't it ugly? Admittedly, things improved when we refactored things out into the forms, but imagine if we hadn't followed through? We'd have been left with a rather unreadable test in our codebase. And even the final version of the tests we ended up with contain some pretty mind-bending bits.

Isolated Tests Don't Automatically Test Integration

As we saw a little later on, isolated tests by their nature only test the unit under test, in isolation. They won't test the integration between your units.

This problem is well known, and there are ways of mitigating it. But, as we saw, those mitigations involve a fair bit of hard work on the part of the programmer—you need to remember to keep track of the interfaces between your units, to identify the implicit contract that each component needs to honour, and you need to write tests for those contracts as well as for the internal functionality of your unit.

Unit Tests Seldom Catch Unexpected Bugs

Unit tests will help you catch off-by-one errors and logic snafus, which are the kinds of bugs we know we introduce all the time, so in a way we are expecting them. But they don't warn you about some of the more unexpected bugs. They won't remind you when you forgot to create a database migration. They won't tell you when the middleware layer is doing some clever HTML-entity escaping that's interfering with the way your data is rendered … something like Donald Rumsfeld's unknown unknowns?

Mocky Tests Can Become Closely Tied to Implementation

And finally, mocky tests can become very tightly coupled with the implementation. If you choose to use `List.objects.create()` to build your objects but your mocks are

expecting you to use `List()` and `.save()`, you'll get failing tests even though the actual effect of the code would be the same. If you're not careful, this can start to work against one of the supposed benefits of having tests, which was to encourage refactoring. You can find yourself having to change dozens of mocky tests and contract tests when you want to change an internal API.

Notice that this may be more of a problem when you're dealing with an API you don't control. You may remember the contortions we had to go through to test our form, mocking out two Django model classes and using `side_effect` to check on the state of the world. If you're writing code that's totally under your own control, you're likely to design your internal APIs so that they are cleaner and require less contortions to test.

But All These Problems Can Be Overcome

But, isolation advocates will come back and say, all that stuff can be mitigated, you just need to get better at writing isolated tests, and, remember the holy flow state? The holy flow state!

So where are we?

Synthesis: What Do We Want from Our Tests, Anyway?

Let's step back and have a think about what benefits we want our tests to deliver. Why are we writing them in the first place?

Correctness

We want our application to be free of bugs—both low-level logic errors, like off-by-one errors, and high-level bugs like the software ultimately should deliver what our users want. We want to find out if we ever introduce regressions which break something that used to work, and we want to find that out before our users see something broken. We expect our tests to tell us our application is correct.

Clean, Maintainable Code

We want our code to obey rules like "YAGNI" and "DRY". We want code that clearly expresses its intentions, which is broken up into sensible components that have well-defined responsibilities and are easily understood. We expect our tests to give us the confidence to refactor our application constantly, so that we're never scared to try and improve its design, and we would also like it if they would actively help us to find the right design.

Productive Workflow

Finally, we want our tests to help enable a fast and productive workflow. We want them to help take some of the stress out of development, we want them to protect us from stupid mistakes. We want them to help keep us in the "flow" state not just because we enjoy it, but because it's highly productive. We want our tests to give us feedback about our work as quickly as possible, so that we can try out new ideas and evolve them quickly. And we don't want to feel like our tests are more of a hindrance than a help when it comes to evolving our codebase.

Evaluate Your Tests Against the Benefits You Want from Them

I don't think there are any universal rules about how many tests you should write and what the correct balance between functional, integrated, and isolated tests should be. Circumstances vary between projects. But, by thinking about all of your tests and asking whether they are delivering the benefits you want, you can make some decisions.

Objective	Some considerations
Correctness	• Do I have enough functional tests to reassure myself that my application *really* works, from the point of view of the user? • Am I testing all the edge cases thoroughly? This feels like a job for low-level, isolated tests. • Do I have tests that check whether all my components fit together properly? Could some integrated tests do this, or are functional tests enough?
Clean, maintainable code	• Are my tests giving me the confidence to refactor my code, fearlessly and frequently? • Are my tests helping me to drive out a good design? If I have a lot of integrated tests and few isolated tests, are there any parts of my application where putting in the effort to write more isolated tests would give me better feedback about my design?
Productive workflow	• Are my feedback cycles as fast as I would like them? When do I get warned about bugs, and is there any practical way to make that happen sooner? • If I have a lot of high-level, functional tests, that take a long time to run, and I have to wait overnight to get feedback about accidental regressions, is there some way I could write some faster tests, integrated tests perhaps, that would get me feedback quicker? • Can I run a subset of the full test suite when I need to? • Am I spending too much time waiting for tests to run, and thus less time in a productive flow state?

Architectural Solutions

There are also some architectural solutions that can help to get the most out of your test suite, and particularly that help avoid some of the disadvantages of isolated tests.

Mainly these involve trying to identify the boundaries of your system—the points at which your code interacts with external systems, like the database or the filesystem, or

the Internet, or the UI—and trying to keep them separate from the core business logic of your application.

Ports and Adapters/Hexagonal/Clean Architecture

Integrated tests are most useful at the *boundaries* of a system—at the points where our code integrates with external systems, like a database, filesystem, or UI components.

Similarly, it's at the boundaries that the downsides of test isolation and mocks are at their worst, because it's at the boundaries that you're most likely to be annoyed if your tests are tightly coupled to an implementation, or to need more reassurance that things are integrated properly.

Conversely, code at the *core* of our application—code that's purely concerned with our business domain and business rules, code that's entirely under our control—this code has less need for integrated tests, since we control and understand all of it.

So one way of getting what we want is to try and minimise the amount of our code that has to deal with boundaries. Then we test our core business logic with isolated tests and test our integration points with integrated tests.

Steve Freeman and Nat Pryce, in their book Growing Object-Oriented Software, call this approach "Ports and Adapters" (see Figure 22-1).

We actually started moving towards a ports and adapters architecture in Chapter 19, when we found that writing isolated unit tests was encouraging us to push ORM code out of the main application, and hide it in helper functions from the model layer.

This pattern is also sometimes known as "The Clean architecture" or "Hexagonal Architecture". See the further reading section at the end for more info.

Functional Core, Imperative Shell

Gary Bernhardt pushes this further, recommending an architecture he calls "Functional Core, Imperative Shell", whereby the "shell" of the application, the place where interaction with boundaries happens, follows the imperative programming paradigm, and can be tested by integrated tests, acceptance tests, or even (gasp!) not at all, if it's kept minimal enough. But the core of the application is actually written following the functional programming paradigm (complete with the "no side effects" corollary), which actually allows fully isolated, "pure" unit tests, *entirely without mocks*.

Check out Gary's presentation titled "Boundaries" for more on this approach.

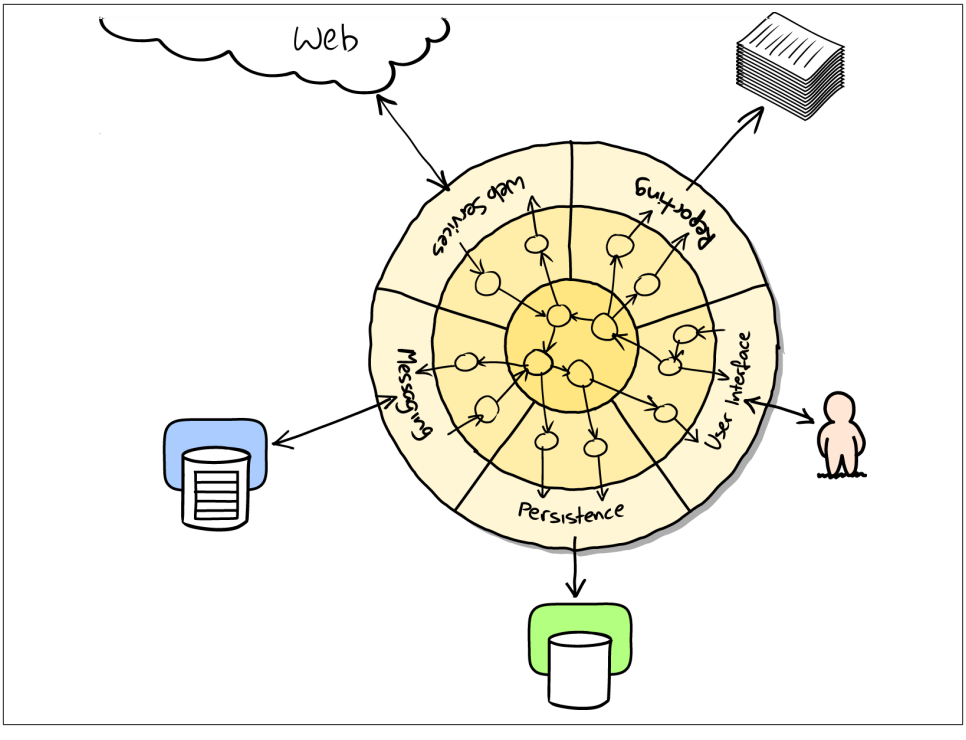


Figure 22-1. Ports and Adapters (diagram by Nat Pryce)

Conclusion

I've tried to give an overview of some of the more advanced considerations that come into the TDD process. Mastery of these topics is something that comes from long years of practice, and therefore I'm grossly underqualified to talk about these things. So I heartily encourage you to take everything I've said with a pinch of salt, to go out there and find out what works for you, and most importantly to go and find the opinions of some real experts!

Here are some places to go for further reading.

Further Reading

Fast Test, Slow Test and Boundaries
: Gary Bernhardt's talks from Pycon 2012 and 2013. His screencasts are also well worth a look.

Ports and Adapters
: Steve Freeman and Nat Pryce wrote about this in their book. You can also catch a good discussion of the idea in this talk. See also Uncle Bob's description of the clean architecture, and Alistair Cockburn coining the term Hexagonal Architecture.

Hot Lava
: Casey Kinsey's memorable warning about avoiding the database whenever you can.

Inverting the Pyramid
: The idea that projects end up with too great a ratio of slow, high-level tests to unit tests, and a visual metaphor for the effort to invert that ratio.

Integrated tests are a scam
: J.B. Rainsberger has a famous rant about the way integrated tests will ruin your life, here. Watch the video presentation here or here (there are two videos available, though neither has perfect cinematography). Then check out a couple of follow-up posts, particularly this defence of acceptance tests (what I call functional tests), and this analysis of how slow tests kill productivity.

A pragmatic view
: Martin Fowler (author of *Refactoring*) presents a reasonably balanced, pragmatic approach.

Obey the Testing Goat!

Back to the Testing Goat.

Groan, I hear you say, *Harry, the Testing Goat stopped being funny about 17 chapters ago*. Bear with me, I'm going to use it to make a serious point.

Testing Is Hard

I think the reason the phrase "Obey the Testing Goat" first grabbed me when I saw it was that it really spoke to the fact that testing is hard—not hard to do in and of itself, but hard to *stick to*, and hard to keep doing.

It always feels easier to cut corners and skip a few tests. And it's doubly hard psychologically because the payoff is so disconnected from the point at which you put in the effort. A test you spend time writing now doesn't reward you immediately, it only helps much later—perhaps months later when it saves you from introducing a bug while refactoring, or catches a regression when you upgrade a dependency. Or, perhaps it pays you back in a way that's hard to measure, by encouraging you to write better designed code, but you convince yourself you could have written it just as elegantly without tests.

I myself started slipping when I was writing the test framework for this book. Being a quite complex beast, it has tests of its own, but I cut several corners, coverage isn't perfect, and I now regret it because it's turned out quite unwieldy and ugly (I'll open source it one day so you can all point and laugh).

Keep Your CI Builds Green

Another area that takes real hard work is continuous integration. You saw in Chapter 20 that strange and unpredictable bugs sometimes occur on CI. When you're looking at these and thinking "it works fine on my machine", there's a strong temptation to just ignore them … but, if you're not careful, you start to tolerate a failing test suite in CI, and pretty soon your CI build is actually useless, and it feels like too much work to get

it going again. Don't fall into that trap. Persist, and you'll find the reason that your test is failing, and you'll find a way to lock it down and make it deterministic, and green, again.

Take Pride in Your Tests, as You Do in Your Code

One of the things that helps is to stop thinking of your tests as being an incidental add-on to the "real" code, and to start thinking of them as being a part of the finished product that you're building—a part that should be just as finely polished, just as aesthetically pleasing, and a part you can be justly proud of delivering…

So do it because the Testing Goat says so. Do it because you know the payoff will be worth it, even if it's not immediate. Do it out of a sense of duty, or professionalism, or OCD, or sheer bloody-mindedness. Do it because it's a good thing to practice. And, eventually, do it because it makes software development more fun.

Remember to Tip the Bar Staff

This book wouldn't have been possible without the backing of my publisher, the wonderful O'Reilly Media. If you're reading the free edition online, I hope you'll consider buying a real copy … if you don't need one for yourself, then maybe as a gift for a friend?

Don't Be a Stranger!

I hope you enjoyed the book. Do get in touch and tell me what you thought!

Harry.

- @hjwp
- *obeythetestinggoat@gmail.com*

APPENDIX A
PythonAnywhere

Are you planning to use PythonAnywhere to follow along with this book? Here's a few notes on how to get things working, specifically with regards to Selenium/Firefox tests, running the test server, and screenshots.

If you haven't already, you'll need to sign up for a PythonAnywhere account. A free one should be fine.

Running Firefox Selenium Sessions with Xvfb

The next thing is that PythonAnywhere is a console-only environment, so it doesn't have a display in which to pop up Firefox. But we can use a virtual display.

In Chapter 1, when we write our first ever test, you'll find things don't work as expected. The first test looks like this, and you can type it in using the PythonAnywhere editor just fine:

```
from selenium import webdriver
browser = webdriver.Firefox()
browser.get('http://localhost:8000')
assert 'Django' in browser.title
```

But when you try and run it (in a *Bash console*), you'll get an error:

```
$ python3 functional_tests.py
Traceback (most recent call last):
File "tests.py", line 3, in <module>
browser = webdriver.Firefox()
File "/usr/local/lib/python3.3/site-packages/selenium/webdriver/firefox/webdrive
self.binary, timeout),
File "/usr/local/lib/python3.3/site-packages/selenium/webdriver/firefox/extensio
self.binary.launch_browser(self.profile)
File "/usr/local/lib/python3.3/site-packages/selenium/webdriver/firefox/firefox_
self._wait_until_connectable()
File "/usr/local/lib/python3.3/site-packages/selenium/webdriver/firefox/firefox_
```

```
    self._get_firefox_output())
selenium.common.exceptions.WebDriverException: Message: 'The browser appears to
have exited before we could connect. The output was: Error: no display
specified\n'
```

The fix is to use *Xvfb*, which stands for X Virtual Framebuffer. It will start up a "virtual" display, which Firefox can use even though the server doesn't have a real one.

If, instead, you see "`ImportError, no module named selenium`", do a `pip3 install --user selenium`.

The command `xvfb-run` will run the next command in Xvfb. Using that will give us our expected failure:

```
$ xvfb-run python3 functional_tests.py
Traceback (most recent call last):
File "tests.py", line 11, in <module>
assert 'Django' in browser.title
AssertionError
```

Setting Up Django as a PythonAnywhere Web App

Shortly after that, we set up Django. Rather than using the `django-admin.py start project` command, I recommend you use the PythonAnywhere quick-start option in the Web tab. Add a new web app, choose Django, Python 3, and then use *superlists* as the project name.

Then, instead of running the test server from a console on `localhost:8000`, you can use the real URL of your PythonAnywhere web app:

```
browser.get('http://my-username.pythonanywhere.com')
```

You'll need to remember to hit "Reload Web App" whenever you make changes to the code, to update the site.

That should work better.[1]

1. You *could* run the Django dev server from a console instead, but the problem is that PythonAnywhere consoles don't always run on the same server, so there's no guarantee that the console you're running your tests in is the same as the one you're running the server in. Plus, when it's running in the console, there's no easy way of visually inspecting how the site looks.

Cleaning Up /tmp

Selenium and Xvfb tend to leave a lot of junk lying around in */tmp*, especially when they're not shut down tidily (that's why I included a `try/finally` earlier).

In fact they leave so much stuff lying around that they might max out your storage quota. So do a tidy-up in */tmp* every so often:

```
$ rm -rf /tmp/*
```

Screenshots

In Chapter 5, I suggest using a `time.sleep` to pause the FT as it runs, so that we can see what the Selenium browser is showing on screen. We can't do that on PythonAnywhere, because the browser runs in a virtual display. Instead, you can inspect the live site, or you could "take my word for it" regarding what you should see.

The best way of doing visual inspections of tests that run in a virtual display is to use screenshots. Take a look at Chapter 20 if you're curious—there's some example code in there.

The Deployment Chapter

When you hit Chapter 8, you'll have the choice of continuing to use PythonAnywhere, or of learning how to build a "real" server. I recommend the latter, because you'll get the most out of it.

If you really want to stick with PythonAnywhere, one option would be deploy a second copy of your app on a different domain. You'll need your own domain name, and a paid account on PythonAnywhere. But even if you don't do that, you should still make sure you can run the FTs in "staging" mode against the real site, rather than using the threaded server from `LiveServerTestCase`.

If you are using PythonAnywhere to follow through with the book, I'd love to hear how you get on! Do send me an email at *obeythetestinggoat@gmail.com*.

APPENDIX B

Django Class-Based Views

This appendix follows on from Chapter 12, in which we implemented Django forms for validation, and refactored our views. By the end of that chapter, our views were still using functions.

The new shiny in the Django world, however, is class-based views. In this appendix, we'll refactor our application to use them instead of view functions. More specifically, we'll have a go at using class-based *generic* views.

Class-Based Generic Views

There's a difference between class-based views and class-based *generic* views. Class-based views are just another way of defining view functions. They make few assumptions about what your views will do, and they offer one main advantage over view functions, which is that they can be subclassed. This comes, arguably, at the expense of being less readable than traditional function-based views. The main use case for *plain* class-based views is when you have several views that reuse the same logic. We want to obey the DRY principle. With function-based views, you would use helper functions or decorators. The theory is that using a class structure may give you a more elegant solution.

Class-based *generic* views are class-based views that attempt to provide ready-made solutions to common use cases: fetching an object from the database and passing it to a template, fetching a list of objects, saving user input from a POST request using a ModelForm, and so on. These sound very much like our use cases, but as we'll soon see, the devil is in the detail.

I should say at this point that I've not used either kind of class-based views much. I can definitely see the sense in them, and there are potentially many use cases in Django apps where CBGVs would fit in perfectly. However, as soon as your use case is slightly outside the basics—as soon as you have more than one model you want to use, for example—I

find that using class-based views can (again, debatably) lead to code that's much harder to read than a classic view function.

Still, because we're forced to use several of the customisation options for class-based views, implementing them in this case can teach us a lot about how they work, and how we can unit test them.

My hope is that the same unit tests we use for function-based views should work just as well for class-based views. Let's see how we get on.

The Home Page as a FormView

Our home page just displays a form on a template:

```
def home_page(request):
    return render(request, 'home.html', {'form': ItemForm()})
```

Looking through the options, Django has a generic view called `FormView`—let's see how that goes:

lists/views.py (ch31l001).

```
from django.views.generic import FormView
[...]

class HomePageView(FormView):
    template_name = 'home.html'
    form_class = ItemForm
```

We tell it what template we want to use, and which form. Then, we just need to update *urls.py*, replacing the line that used to say `lists.views.home_page`:

superlists/urls.py (ch31l002).

```
    url(r'^$', HomePageView.as_view(), name='home'),
```

And the tests all check out! That was easy…

```
$ python3 manage.py test lists
[...]

Ran 34 tests in 0.119s
OK

$ python3 manage.py test functional_tests
[...]

Ran 4 tests in 15.160s
OK
```

So far so good. We've replaced a one-line view function with a two-line class, but it's still very readable. This would be a good time for a commit…

Using form_valid to Customise a CreateView

Next we have a crack at the view we use to create a brand new list, currently the `new_list` function. Here's what it looks like now:

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})
```

Looking through the possible CBGVs, we probably want a `CreateView`, and we know we're using the `ItemForm` class, so let's see how we get on with them, and whether the tests will help us:

lists/views.py (ch31l003).

```
from django.views.generic import FormView, CreateView
[...]

class NewListView(CreateView):
    form_class = ItemForm

def new_list(request):
    [...]
```

I'm going to leave the old view function in *views.py*, so that we can copy code across from it. We can delete it once everything is working. It's harmless as soon as we switch over the URL mappings, this time in:

lists/urls.py (ch31l004).

```
from django.conf.urls import patterns, url
from lists.views import NewListView

urlpatterns = patterns('',
    url(r'^(\d+)/$', 'lists.views.view_list', name='view_list'),
    url(r'^new$', NewListView.as_view(), name='new_list'),
)
```

Now running the tests gives three errors:

```
$ python3 manage.py test lists

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_saving_a_POST_request (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()'

Ran 34 tests in 0.125s

FAILED (errors=6)
```

Let's start with the third—maybe we can just add the template?

lists/views.py (ch31l005).

```
class NewListView(CreateView):
    form_class = ItemForm
    template_name = 'home.html'
```

That gets us down to just two failures: we can see they're both happening in the generic view's `form_valid` function, and that's one of the ones that you can override to provide custom behaviour in a CBGV. As its name implies, it's run when the view has detected a valid form. We can just copy some of the code from our old view function, that used to live after `if form.is_valid():`:

lists/views.py (ch31l005).

```
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_)
```

That gets us a full pass!

```
$ python3 manage.py test lists
Ran 34 tests in 0.119s
OK
$ python3 manage.py test functional_tests
Ran 4 tests in 15.157s
OK
```

And we *could* even save two more lines, trying to obey "DRY", by using one of the main advantages of CBVs: inheritance!

lists/views.py (ch31l007).

```
class NewListView(CreateView, HomePageView):

    def form_valid(self, form):
        list = List.objects.create()
        Item.objects.create(text=form.cleaned_data['text'], list=list)
        return redirect('/lists/%d/' % (list.id,))
```

And all the tests would still pass:

```
OK
```

This is not really good object-oriented practice. Inheritance implies an "is-a" relationship, and it's probably not meaningful to say that our new list view "is-a" home page view … so, probably best not to do this.

With or without that last step, how does it compare to the old version? I'd say that's not bad. We save some boilerplate code, and the view is still fairly legible. So far, I'd say we've got one point for CBGVs, and one draw.

A More Complex View to Handle Both Viewing and Adding to a List

This took me *several* attempts. And I have to say that, although the tests told me when I got it right, they didn't really help me to figure out the steps to get there … mostly it was just trial and error, hacking about in functions like `get_context_data`, `get_form_kwargs`, and so on.

One thing it did made me realise was the value of having lots of individual tests, each testing one thing. I went back and rewrote some of Chapters 10–12 as a result.

The Tests Guide Us, for a While

Here's how things might go. Start by thinking we want a `DetailView`, something that shows you the detail of an object:

lists/views.py.

```
from django.views.generic import FormView, CreateView, DetailView
[...]

class ViewAndAddToList(DetailView):
    model = List
```

That gives:

```
[...]
AttributeError: Generic detail view ViewAndAddToList must be called with either
an object pk or a slug.
```

```
FAILED (failures=5, errors=6)
```

Not totally obvious, but a bit of Googling around led me to understand that I needed to use a "named" regex capture group:

lists/urls.py (ch31l011).

```
@@ -1,7 +1,7 @@
 from django.conf.urls import patterns, url
-from lists.views import NewListView
+from lists.views import NewListView, ViewAndAddToList

 urlpatterns = patterns('',
-    url(r'^(\d+)/$', 'lists.views.view_list', name='view_list'),
+    url(r'^(?P<pk>\d+)/$', ViewAndAddToList.as_view(), name='view_list'),
     url(r'^new$', NewListView.as_view(), name='new_list'),
 )
```

The next error was fairly helpful:

```
[...]
django.template.base.TemplateDoesNotExist: lists/list_detail.html

FAILED (failures=5, errors=6)
```

That's easily solved:

lists/views.py.

```
class ViewAndAddToList(DetailView):
    model = List
    template_name = 'list.html'
```

That takes us down three errors:

```
[...]
ERROR: test_displays_item_form (lists.tests.test_views.ListViewTest)
KeyError: 'form'

FAILED (failures=5, errors=2)
```

Until We're Left with Trial and Error

So I figured, our view doesn't just show us the detail of an object, it also allows us to create new ones. Let's make it both a `DetailView` *and* a `CreateView`:

lists/views.py.

```
class ViewAndAddToList(DetailView, CreateView):
    model = List
    template_name = 'list.html'
    form_class = ExistingListItemForm
```

But that gives us a lot of errors saying:

```
[...]
TypeError: __init__() missing 1 required positional argument: 'for_list'
```

And the `KeyError: 'form'` was still there too!

At this point the errors stopped being quite as helpful, and it was no longer obvious what to do next. I had to resort to trial and error. Still, the tests did at least tell me when I was getting things more right or more wrong.

My first attempts to use `get_form_kwargs` didn't really work, but I found that I could use `get_form`:

lists/views.py.

```
    def get_form(self, form_class):
        self.object = self.get_object()
        return form_class(for_list=self.object, data=self.request.POST)
```

But it would only work if I also assigned to `self.object`, as a side effect, along the way, which was a bit upsetting. Still, that takes us down to just three errors, but we're still apparently not passing that form to the template!

```
KeyError: 'form'

FAILED (errors=3)
```

Back on Track

A bit more experimenting led me to swap out the `DetailView` for a `SingleObjectMixin` (the docs had some useful pointers here):

```
from django.views.generic.detail import SingleObjectMixin
[...]

class ViewAndAddToList(CreateView, SingleObjectMixin):
    [...]
```

That takes us down to just two errors:

```
django.core.exceptions.ImproperlyConfigured: No URL to redirect to.  Either
provide a url or define a get_absolute_url method on the Model.
```

And for this final failure, the tests are being helpful again. It's quite easy to define a `get_absolute_url` on the `Item` class, such that items point to their parent list's page:

lists/models.py.

```
class Item(models.Model):
    [...]

    def get_absolute_url(self):
        return reverse('view_list', args=[self.list.id])
```

Is That Your Final Answer?

We end up with a view class that looks like this:

lists/views.py (ch31l010).

```
class ViewAndAddToList(CreateView, SingleObjectMixin):
    template_name = 'list.html'
    model = List
    form_class = ExistingListItemForm

    def get_form(self, form_class):
        self.object = self.get_object()
        return form_class(for_list=self.object, data=self.request.POST)
```

Compare Old and New

Let's see the old version for comparison?

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})
```

Well, it has reduced the number of lines of code from nine to seven. Still, I find the function-based version a little easier to understand, in that it has a little bit less magic —"explicit is better than implicit", as the Zen of Python would have it. I mean … `SingleObjectMixin`? What? And, more offensively, the whole thing falls apart if we don't assign to `self.object` inside `get_form`? Yuck.

Still, I guess some of it is in the eye of the beholder.

Best Practices for Unit Testing CBGVs?

As I was working through this, I felt like my "unit" tests were sometimes a little too high-level. This is no surprise, since tests for views that involve the Django test client are probably more properly called integrated tests.

They told me whether I was getting things right or wrong, but they didn't always offer enough clues on exactly how to fix things.

I occasionally wondered whether there might be some mileage in a test that was closer to the implementation—something like this:

```
    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)
```

But the problem is that it requires a lot of knowledge of the internals of Django CBVs to be able to do the right test setup for these kinds of tests. And you still end up getting very confused by the complex inheritance hierarchy.

Take-Home: Having Multiple, Isolated View Tests with Single Assertions Helps

One thing I definitely did conclude from this appendix was that having many short unit tests for views was much more helpful than having few tests with a narrative series of assertions.

Consider this monolithic test:

```
    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)
```

That is definitely less useful than having three individual tests, like this:

```
    def test_invalid_input_means_nothing_saved_to_db(self):
        self.post_invalid_input()
        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))
```

The reason is that, in the first case, an early failure means not all the assertions are checked. So, if the view was accidentally saving to the database on invalid POST, you would get an early fail, and so you wouldn't find out whether it was using the right template or rendering the form. The second formulation makes it much easier to pick out exactly what was or wasn't working.

Lessons Learned from CBGVs

Class-based generic views can do anything
: It might not always be clear what's going on, but you can do just about anything with class-based views.

Single-assertion unit tests help refactoring
: With each unit test providing individual guidance on what works and what doesn't, it's much easier to change the implementation of our views to using this fundamentally different paradigm.

APPENDIX C

Provisioning with Ansible

We used Fabric to automate deploying new versions of the source code to our servers. But provisioning a fresh server, and updating the Nginx and Gunicorn config files, was all left as a manual process.

This is the kind of job that's increasingly given to tools called "Configuration Management" or "Continuous Deployment" tools. Chef and Puppet were the first popular ones, and in the Python world there's Salt and Ansible.

Of all of these, Ansible is the easiest to get started with. We can get it working with just two files:

```
pip install ansible  # Python 2 sadly
```

An "inventory file" at `deploy_tools/inventory.ansible` defines what servers we can run against:

deploy_tools/inventory.ansible.

```
[live]
superlists.ottg.eu

[staging]
superlists-staging.ottg.eu

[local]
localhost ansible_ssh_port=6666 ansible_host=127.0.0.1
```

(The local entry is just an example, in my case a Virtualbox VM, with port forwarding for ports 22 and 80 set up.)

Installing System Packages and Nginx

Next the Ansible "playbook", which defines what to do on the server. This uses a syntax called YAML:

deploy_tools/provision.ansible.yaml.

```
---

- hosts: all

  sudo: yes

  vars:
      host: $inventory_hostname

  tasks:
    - name: make sure required packages are installed
      apt: pkg=nginx,git,python3,python3-pip state=present
    - name: make sure virtualenv is installed
      shell: pip3 install virtualenv

    - 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
```

The vars section defines a variable "host" for convenience, which we can then use in the various filenames and pass to the config files themselves. It comes from `$inventory_hostname`, which is the domain name of the server we're running against at the time.

In this section, we install our required software using `apt`, tweak the Nginx config to allow long hostnames using a regular expression replacer, and then we write the Nginx config file using a template. This is a modified version of the template file we saved into *deploy_tools/nginx.template.conf* in Chapter 8, but it now uses a specific templating syntax—Jinja2, which is actually a lot like the Django template syntax:

deploy_tools/nginx.conf.j2.

```
server {
    listen 80;
    server_name {{ host }};

    location /static {
```

```
        alias /home/harry/sites/{{ host }}/static;
    }

    location / {
        proxy_set_header Host $host;
        proxy_pass http://unix:/tmp/{{ host }}.socket;
    }
}
```

Configuring Gunicorn, and Using Handlers to Restart Services

Here's the second half of our playbook:

deploy_tools/provision.ansible.yaml.

```
    - name: write gunicorn init script
      template: src=./gunicorn-upstart.conf.j2
                dest=/etc/init/gunicorn-{{ host }}.conf
      notify:
          - restart gunicorn

    - name: make sure nginx is running
      service: name=nginx state=running
    - name: make sure gunicorn is running
      service: name=gunicorn-{{ host }} state=running

  handlers:
    - name: restart nginx
      service:  name=nginx state=restarted

    - name: restart gunicorn
      service:  name=gunicorn-{{ host }} state=restarted
```

Once again we use a template for our Gunicorn config:

deploy_tools/gunicorn.upstart.conf.j2.

```
description "Gunicorn server for {{ host }}"

start on net-device-up
stop on shutdown

respawn

chdir /home/harry/sites/{{ host }}/source
exec ../virtualenv/bin/gunicorn \
    --bind unix:/tmp/{{ host }}.socket \
    --access-logfile ../access.log \
    --error-logfile ../error.log \
    superlists.wsgi:application
```

Then we have two "handlers" to restart Nginx and Gunicorn. Ansible is clever, so if it sees multiple steps all call the same handlers, it waits until the last one before calling it.

And that's it! The command to kick all these off is:

```
ansible-playbook -i ansible.inventory provision.ansible.yaml --limit=staging
```

Lots more info in the Ansible docs.

What to Do Next

I've just given a little taster of what's possible with Ansible. But the more you automate about your deployments, the more confidence you will have in them. Here's a few more things to look into.

Move Deployment out of Fabric and into Ansible

We've seen that Ansible can help with some aspects of provisioning, but it can also do pretty much all of our deployment for us. See if you can extend the playbook to do everything that we currently do in our fabric deploy script, including notifying the restarts as required.

Use Vagrant to Spin Up a Local VM

Running tests against the staging site gives us the ultimate confidence that things are going to work when we go live, but we can also use a VM on our local machine.

Download Vagrant and Virtualbox, and see if you can get Vagrant to build a dev server on your own PC, using our Ansible playbook to deploy code to it. Rewire the FT runner to be able to test against the local VM.

Having a Vagrant config file is particularly helpful when working in a team—it helps new developers to spin up servers that look exactly like yours.

APPENDIX D

Testing Database Migrations

Django-migrations and its predecessor South have been around for ages, so it's not usually necessary to test database migrations. But it just so happens that we're introducing a dangerous type of migration, ie one that introduces a new integrity constraint on our data. When I first ran the migration script against staging, I saw an error.

On larger projects, where you have sensitive data, you may want the additional confidence that comes from testing your migrations in a safe environment before applying them to production data, so this toy example will hopefully be a useful rehearsal.

Another common reason to want to test migrations is for speed—migrations often involve downtime, and sometimes, when they're applied to very large datasets, they can take time. It's good to know in advance how long that might be.

An Attempted Deploy to Staging

Here's what happened to me when I first tried to deploy our new validation constraints in Chapter 14:

```
$ 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/python3.3/dist-packages/django/db/backends/utils.py",
line 61, in execute
    return self.cursor.execute(sql, params)
  File
"/usr/local/lib/python3.3/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
[...]
```

What happened was that some of the existing data in the database violated the integrity constraint, so the database was complaining when I tried to apply it.

In order to deal with this sort of problem, we'll need to build a "data migration". Let's first set up a local environment to test against.

Running a Test Migration Locally

We'll use a copy of the live database to test our migration against.

Be very, very, very careful when using real data for testing. For example, you may have real customer email addresses in there, and you don't want to accidentally send them a bunch of test emails. Ask me how I know this.

Entering Problematic Data

Start a list with some duplicate items on your live site, as shown in Figure D-1.

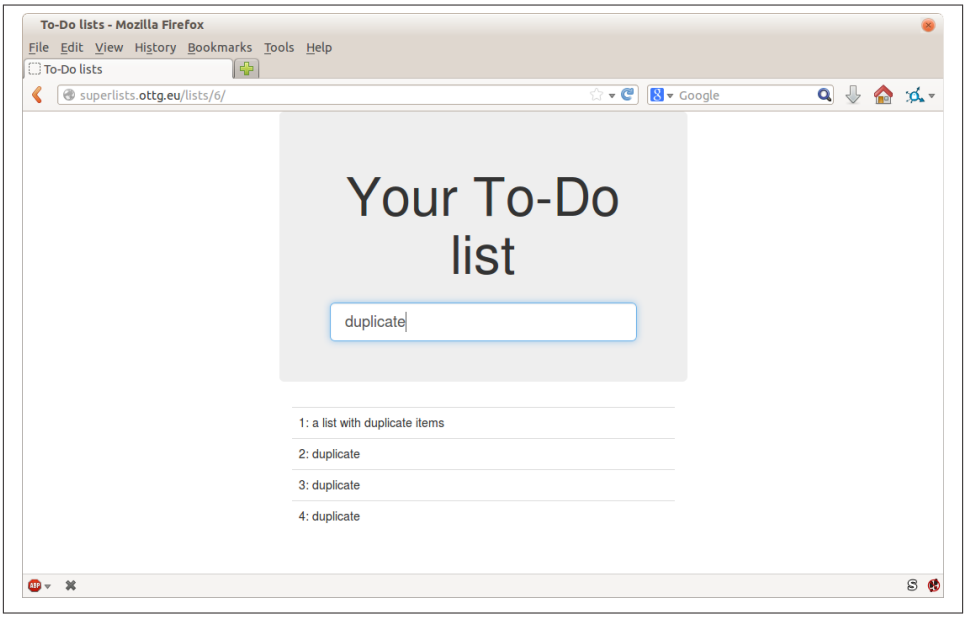


Figure D-1. A list with duplicate items

Copying Test Data from the Live Site

Copy the database down from 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
```

Confirming the Error

We now have a local database that has not been migrated, and that contains some problematic data. We should see an error if we try to run `migrate`:

```
$ python3 manage.py migrate --migrate
python3 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
```

Inserting a Data Migration

Data migrations are a special type of migration that modifies data in the database rather than changing the schema. We need to create one that will run before we apply the integrity constraint, to preventively remove any duplicates. Here's how we can do that:

```
$ git rm lists/migrations/0005_list_item_unique_together.py
$ python3 manage.py makemigrations lists --empty
Migrations for 'lists':
  0005_auto_20140414_2325.py:
$ mv lists/migrations/0005_ lists/migrations/0005_remove_duplicates.py*
```

Check out the Django docs on data migrations for more info, but here's how we add some instructions to change existing data:

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),
    ]
```

Re-creating the Old Migration

We re-create the old migration using `makemigrations`, which will ensure it is now the sixth migration and has an explicit dependency on `0005`, the data migration:

```
$ python3 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
```

Testing the New Migrations Together

We're now ready to run our test against the live data:

```
$ cd deploy_tools
$ fab deploy:host=elspeth@superlists-staging.ottg.eu
[...]
```

We'll need to restart the live Gunicorn job too:

```
elspeth@server:$ sudo restart gunicorn-superlists.ottg.eu
```

And we can now run our FTs against staging:

```
$ python3 manage.py test functional_tests --liveserver=superlists-staging.ottg.eu
Creating test database for alias 'default'...
....
 ---------------------------------------------------------------------
Ran 4 tests in 17.308s

OK
```

Everything seems in order! Let's do it against live:

```
$ fab deploy --host=superlists.ottg.eu
[superlists.ottg.eu] Executing task 'deploy'
[...]
```

And that's a wrap. `git add lists/migrations`, `git commit`, etc.

Conclusions

This exercise was primarily aimed at building a data migration and testing it against some real data. Inevitably, this is only a drop in the ocean of the possible testing you could do for a migration. You could imagine building automated tests to check that all your data was preserved, comparing the database contents before and after. You could write individual unit tests for the helper functions in a data migration. You could spend more time measuring the time taken for migrations, and experiment with ways to speed it up by, eg, breaking up migrations into more or fewer component steps.

Remember that this should be a relatively rare case. In my experience, I haven't felt the need to test 99% of the migrations I've worked on. But, should you ever feel the need on your project, I hope you've found a few pointers here to get started with.

On Testing Database Migrations

Be wary of migrations which introduce constraints
: 99% of migrations happen without a hitch, but be wary of any situations, like this one, where you are introducing a new constraint on columns that already exist.

Test migrations for speed
: Once you have a larger project, you should think about testing how long your migrations are going to take. Database migrations typically involve downtime, as, depending on your database, the schema update operation may lock the table it's working on until it completes. It's a good idea to use your staging site to find out how long a migration will take.

Be extremely careful if using a dump of production data
: In order to do so, you'll want fill your staging site's database with an amount of data that's commensurate to the size of your production data. Explaining how to do that is outside of the scope of this book, but I will say this: if you're tempted to just take a dump of your production database and load it into staging, be *very* careful. Production data contains real customer details, and I've personally been responsible for accidentally sending out a few hundred incorrect invoices after an automated process on my staging server started processing the copied production data I'd just loaded into it. Not a fun afternoon.

APPENDIX E

What to Do Next

Here follow a few suggestions for things to investigate next, to develop your testing skills, and to apply them to some of the cool new technologies in web development (at the time of writing!).

I hope to turn each one of these into at least some sort of blog post, if not a future appendix to the book. I hope to also produce code examples for all of them, as time goes by. So do check out *http://www.obeythetestinggoat.com*, and see if there are any updates.

Or, why not try and beat me to it, and write your own blog post chronicling your attempt at any one of these?

I'm very happy to answer questions and provide tips and guidance on all these topics, so if you find yourself attempting one and getting stuck, please don't hesitate to get in touch at *obeythetestinggoat@gmail.com*!

Notifications—Both on the Site and by Email

It would be nice if users were notified when someone shares a list with them.

You can use django-notifications to show a message to users the next time they refresh the screen. You'll need two browsers in your FT for this.

And/or, you could send notifications by email. Investigate Django's email test capabilities. Then, decide this is so critical that you need real tests with real emails. Use the IMAPClient library to fetch actual emails from a test webmail account.

Switch to Postgres

SQLite is a wonderful little database, but it won't deal well once you have more than one web worker process fielding your site's requests. Postgres is everyone's favourite database these days, so find out how to install and configure it.

You'll need to figure out a place to store the usernames and passwords for your local, staging, and production Postgres servers. Since, for security, you probably don't want them in your code repository, look into ways of modifying your deploy scripts to pass them in at the command line. Environment variables are one popular solution for where to keep them…

Experiment with keeping your unit tests running with SQLite, and compare how much faster they are than running against Postgres. Set it up so that your local machine uses SQLite for testing, but your CI server uses Postgres.

Run Your Tests Against Different Browsers

Selenium supports all sorts of different browsers, including Chrome and Internet Exploder. Try them both out and see if your FT suite behaves any differently.

You should also check out a "headless" browser like PhantomJS.

In my experience, switching browsers tends to expose all sorts of race conditions in Selenium tests, and you will probably need to use the interaction/wait pattern a lot more (particularly for PhantomJS).

404 and 500 Tests

A professional site needs good looking error pages. Testing a 404 page is easy, but you'll probably need a custom "raise an exception on purpose" view to test the 500 page.

The Django Admin Site

Imagine a story where a user emails you wanting to "claim" an anonymous list. Let's say we implement a manual solution to this, involving the site administrator manually changing the record using the Django admin site.

Find out how to switch on the admin site, and have a play with it. Write an FT that shows a normal, non-logged-in user creating a list, then have an admin user log in, go to the admin site, and assign the list to the user. The user can then see it in their "My Lists" page.

Investigate a BDD Tool

BDD stands for Behaviour-Driven Development. It's a way of writing tests that should make them more human-readable, by implementing a sort of Domain-Specific Language (DSL) for your FT code. Check out Lettuce (a Python BDD framework) and use it to rewrite an existing FT, or write a new one.

Write Some Security Tests

Expand on the login, my lists, and sharing tests—what do you need to write to assure yourself that users can only do what they're authorized to?

Test for Graceful Degradation

What would happen if Persona went down? Can we at least show an apologetic error message to our users?

- Tip: one way of simulating Persona being down is to hack your hosts file (at */etc/hosts* or *c:\Windows\Sytem32\drivers\etc*). Remember to revert it in the test `tearDown`!
- Think about the server side as well as the client side.

Caching and Performance Testing

Find out how to install and configure `memcached`. Find out how to use Apache's `ab` to run a performance test. How does it perform with and without caching? Can you write an automated test that will fail if caching is not enabled? What about the dreaded problem of cache invalidation? Can tests help you to make sure your cache invalidation logic is solid?

JavaScript MVC Frameworks

JavaScript libraries that let you implement a Model-View-Controller pattern on the client side are all the rage these days. To-do lists are one of the favourite demo applications for them, so it should be pretty easy to convert the site to being a single-page site, where all list additions happen in JavaScript.

Pick a framework—perhaps Backbone.js or Angular.js—and spike in an implementation. Each framework has its own preferences for how to write unit tests, so learn the one that goes along with it, and see how you like it.

Async and Websockets

Supposing two users are working on the same list at the same time. Wouldn't it be nice to see real-time updates, so if the other person adds an item to the list, you see it immediately? A persistent connection between client and server using websockets is the way to get this to work.

Check out one of the Python async web servers—Tornado, gevent, Twisted—and see if you can use it to implement dynamic notifications.

To test it, you'll need two browser instances (like we used for the list sharing tests), and check that notifications of the actions from one appear in the other, without needing to refresh the page…

Switch to Using py.test

py.test lets you write unit tests with less boilerplate. Try converting some of your unit tests to using *py.test*. You may need to use a plugin to get it to play nicely with Django.

Client-Side Encryption

Here's a fun one: what if our users are paranoid about the NSA, and decide they no longer want to trust their lists to The Cloud? Can you build a JavaScript encryption system, where the user can enter a password to encypher their list item text before it gets sent to the server?

One way of testing it might be to have an "administrator" user that goes to the Django admin view to inspect users' lists, and checks that they are stored encrypted in the database.

Your Suggestion Here

What do you think I should put here? Suggestions please!

APPENDIX F

Cheat Sheet

By popular demand, this "cheat sheet" is loosely based on the little recap/summary boxes from the end of each chapter. The idea is to provide a few reminders, and links to the chapters where you can find out more to jog your memory. I hope you find it useful!

Initial Project Setup

- Start with a *User Story* and map it to a first *functional test*.
- Pick a test framework—`unittest` is fine, options like `py.test` and `nose` can also offer some advantages.
- Run the functional test and see your first *expected failure*.
- Pick a web framework such as Django, and find out how to run *unit tests* against it.
- Create your first *unit test* to address the current FT failure, and see it fail.
- Do your *first commit* to a VCS like *Git*.

Relevant chapters: Chapter 1, Chapter 2, Chapter 3

The Basic TDD Workflow

- Double-loop TDD (Figure F-1)
- Red, Green, Refactor
- Trianglulation
- The scratchpad
- "3 Strikes and Refactor"
- "Working State to Working State"

- "YAGNI"

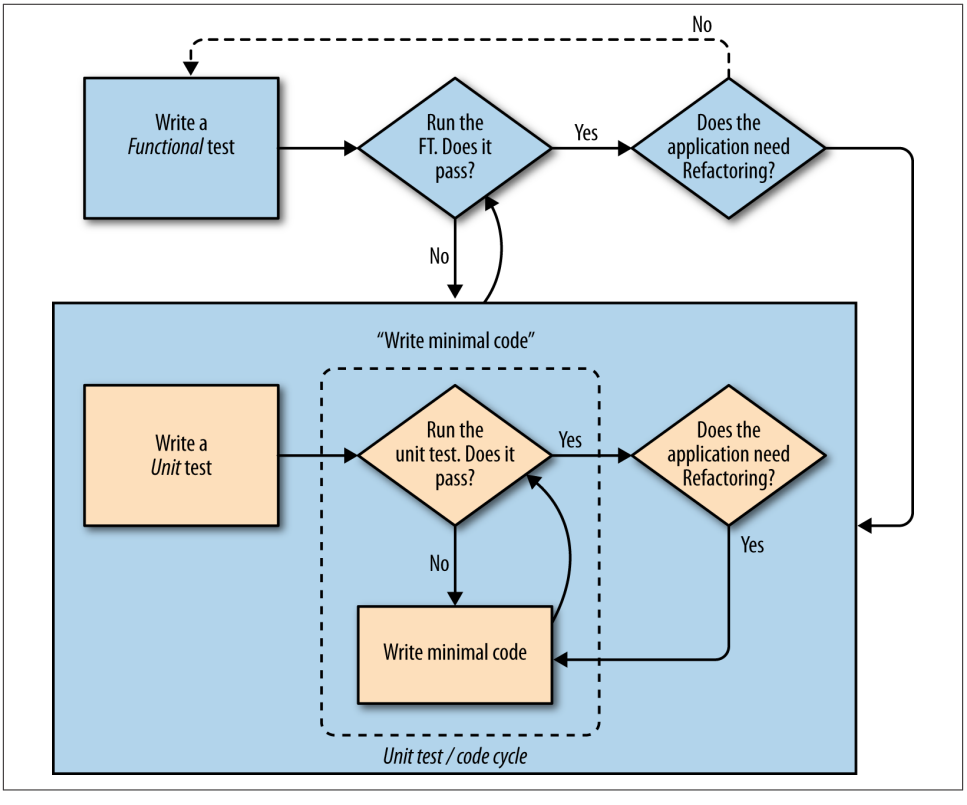


Figure F-1. The TDD process with Functional and Unit tests

Relevant chapters: Chapter 4, Chapter 5, Chapter 6

Moving Beyond dev-only Testing

- Start system testing early. Ensure your components work together: web server, static content, database.
- Build a staging environment to match your production environment, and run your FT suite against it.
- Automate your staging and production environments:
  - — PaaS vs. VPS
  - — Fabric
  - — Configuration management (Chef, Puppett, Salt, Ansible)

— Vagrant

- Think through deployment pain points: the database, static files, dependencies, how to customise settings, etc.
- Build a CI server as soon as possible, so that you don't have to rely on self-discipline to see the tests run.

Relevant chapters: Chapter 8, Chapter 9, Chapter 20, Appendix C

General Testing Best Practices

- Each test should test one thing.
- One test file per application code source file.
- Consider at least a placeholder test for every function and class, no matter how simple.
- "Don't test constants".
- Try to test behaviour rather than implementation.
- Try to think beyond the charmed path through the code, and think through edge cases and error cases.

Relevant chapters: Chapter 4, Chapter 10, Chapter 11

Selenium/Functional Testing Best Practices

- Use explicit rather than implicit waits, and the interaction/wait pattern.
- Avoid duplication of test code—helper methods in base class, or Page pattern are one way to go.
- Avoid double-testing functionality. If you have a test that covers a time-consuming process (eg, login), consider ways of skipping it in other tests (but be aware of unexpected interactions between seemingly unrelated bits of functionality).
- Look into BDD tools as another way of structuring your FTs.

Relevant chapters: Chapter 17, Chapter 20, Chapter 21

Outside-In, Test Isolation Versus Integrated Tests, and Mocking

Be reminded of the reason we write tests in the first place:

- To ensure correctness, and prevent regressions
- To help us to write clean, maintainable code
- To enable a fast, productive workflow

And with those objectives in mind, think of different types of tests, and the tradeoffs between them:

Functional tests

- Provide the best guarantee that your application really works correctly, from the point of view of the user.
- But: it's a slower feedback cycle,
- And they don't necessarily help you write clean code.

Integrated tests (reliant on, eg, the ORM or the Django Test Client)

- Are quick to write,
- Easy to understand,
- Will warn you of any integration issues,
- But may not always drive good design (that's up to you!).
- And are usually slower than isolated tests.

Isolated ("mocky") tests

- These involve the most hard work.
- They can be harder to read and understand,
- But: these are the best ones for guiding you towards better design.
- And they run the fastest.

If you do find yourself writing tests with lots of mocks, and they feel painful, remember *"listen to your tests"*—ugly, mocky tests may be trying to tell you that your code could be simplified.

Relevant chapters: Chapter 18, Chapter 19, Chapter 22

APPENDIX G
Bibliography

- [dip] Mark Pilgrim, *Dive Into Python*: *http://www.diveintopython.net/*
- [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, *TDD By Example*, Addison-Wesley
- [refactoring] Martin Fowler, *Refactoring*, Addison-Wesley
- [seceng] Ross Anderson, *Security Engineering, Second Edition*, Addison-Wesley: *http://www.cl.cam.ac.uk/~rja14/book.html*
- [jsgoodparts] Douglas Crockford, *JavaScript: The Good Parts*, O'Reilly
- [twoscoops] Daniel Greenfield and 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 and Nat Pryce, *Growing Object-Oriented Software Guided by Tests*, Addison-Wesley

Index

We'd like to hear your suggestions for improving our indexes. Send email to index@oreilly.com.

D

E

F

G

H

N

O

P

Q

R

S

T

U

V

W

X

Y

About the Author

After an idyllic childhood spent playing with BASIC on French 8-bit computers like the Thomson T-07 whose keys go "boop" when you press them, Harry spent a few years being deeply unhappy with economics and management consultancy. Soon he rediscovered his true geek nature, and was lucky enough to fall in with a bunch of XP fanatics, working on the pioneering but sadly defunct Resolver One spreadsheet. He now works at PythonAnywhere LLP, and spreads the gospel of TDD worldwide at talks, workshops, and conferences, with all the passion and enthusiasm of a recent convert.

Colophon

The animal on the cover of *Test-Driven Development with Python* is a cashmere goat. Though all goats can produce a cashmere undercoat, only those goats selectively bred to produce cashmere in commercially viable amounts are typically considered "cashmere goats." Cashmere goats thus belong to the domestic goat species *Capra hircus*.

The exceptionally fine, soft hair of the undercoat of a cashmere goat grows alongside an outer coat of coarser hair as part of the goat's double fleece. The cashmere undercoat appears in winter to supplement the protection offered by the outer coat, called *guard hair*. The crimped quality of cashmere hair in the undercoat accounts for its lightweight yet effective insulation properties.

The name "cashmere" is derived from the Kashmir Valley region on the Indian subcontinent where the textile has been manufactured for thousands of years. A diminishing population of cashmere goats in modern Kashmir has led to the cessation of exports of cashmere fiber from the area. Most cashmere wool now originates in Afghanistan, Iran, Outer Mongolia, India, and—predominantly—China.

Cashmere goats grow hair of varying colors and color combinations. Both males and females have horns, which serve to keep the animals cool in summer and provide the goats' owners with effective handles during farming activities.

The cover image is from Wood's Animate Creation. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.