

*Fewer Bugs and Less Stress with Selenium,  
Django, and JavaScript*



**Early Release**

**RAW & UNEDITED**

# Test-Driven Web Development with Python

O'REILLY®

*Harry Percival*

---

# Test-Driven Web Development with Python

*Harry Percival*

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo



# Test-Driven Web Development with Python

by Harry Percival

Copyright © 2010 Harry Percival. All rights reserved.

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](mailto:corporate@oreilly.com).

**Editor:** Meghan Blanchette

**Production Editor:** FIX ME!

**Copyeditor:** FIX ME!

**Proofreader:** FIX ME!

**Indexer:** FIX ME!

**Cover Designer:** Karen Montgomery

**Interior Designer:** David Futato

**Illustrator:** Robert Romano

January 2014: First Edition

## Revision History for the First Edition:

2013-03-12: Early release revision 1

2013-05-08: Early release revision 2

2013-08-06: Early release revision 3

2013-10-30: Early release revision 4

2013-12-02: Early release revision 5

2014-01-27: Early release revision 6

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

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. !!FILL THIS IN!! and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-449-36482-3

[?]

---

# Table of Contents

<b>Preface.....</b>	<b>ix</b>
<b>1. Getting Django set up using a Functional Test.....</b>	<b>1</b>
Obey the Testing Goat! Do nothing until you have a test	1
Getting Django up and running	4
Starting a Git repository	5
<b>2. Extending our Functional Test using the unittest module.....</b>	<b>9</b>
Using the Functional Test to scope out a minimum viable app	9
The Python standard library's unittest module	12
Implicitly wait	14
Commit	14
<b>3. Testing a simple home page with unit tests.....</b>	<b>17</b>
Our first Django app, and our first unit test	18
Unit tests, and how they differ from Functional tests	18
Unit testing in Django	19
Django's MVC, URLs and view functions	20
At last! We actually write some application code!	22
Reading tracebacks	23
urls.py	24
Unit testing a view	26
The unit test / code cycle	27
<b>4. What are we doing with all these tests?.....</b>	<b>31</b>
Programming is like pulling a bucket of water up from a well	32
On the merits of trivial tests for trivial functions	33
Using Selenium to test user interactions	33
The “Don't test constants” rule, and templates to the rescue	36

On refactoring	40
A little more of our front page	41
Recap: the TDD process	42
<b>5. Saving user input. ....</b>	<b>47</b>
Wiring up our form to send a POST request	47
Processing a POST request on the server	50
3 strikes and refactor	55
The Django ORM & our first model	56
Saving the POST to the database	59
Redirect after a POST	61
Better unit testing practice: each test should test one thing	62
Rendering items in the template	63
Creating our production database with syncdb	65
<b>6. Getting to the minimum viable site. ....</b>	<b>71</b>
Ensuring test isolation in functional tests	71
Running just the unit tests	74
Small Design When Necessary	76
YAGNI!	76
REST	77
Implementing the new design using TDD	78
Iterating towards the new design	80
Testing views, templates and URLs together with the Django Test Client	81
A URL and view for adding list items	86
Adjusting our models	89
Each list should have its own URL	93
One more view to handle adding items to an existing list	95
A final refactor using URL includes	99
<b>7. Prettification: layout and styling, and what to test about it. ....</b>	<b>101</b>
What to functionally test about layout and style	101
Prettification: Using a CSS framework	104
Django template inheritance	106
Integrating Bootstrap	108
Static files in Django	109
What we skipped over: collectstatic and other static directories	113
<b>8. Testing deployment using a staging site. ....</b>	<b>117</b>
TDD and the Danger Areas of deployment	118
As always, start with a test	119
Getting a domain name	121

Manually provisioning a server to host our site	121
Choosing where to host our site	122
Spinning up a server	123
User accounts, SSH and privileges	123
Installing Nginx	124
Configuring domains for staging and live	125
Deploying our code manually	126
Adjusting the database location	127
Creating a virtualenv	128
Simple nginx configuration	130
Creating the database with syncdb	133
Switching to Gunicorn	134
Getting Nginx to serve static files	135
Switching to using Unix sockets	136
Switching DEBUG to False and setting ALLOWED_HOSTS	137
Using upstart to make sure gunicorn starts on boot	138
Saving our changes: adding gunicorn to our requirements.txt	138
Automating:	139
Automating deployment with fabric	141
Git tag the release	150
Recap:	150
Further reading:	151
Todos	151
<b>9. Input validation and test organisation.....</b>	<b>153</b>
Validation FT: preventing blank items	153
Skipping a test	154
Splitting functional tests out into many files	155
Running a single test file	158
Fleshing out the FT	158
Using model-layer validation	159
Refactoring unit tests into several files	159
Unit testing model validation and the self.assertRaises context manager	161
Overriding the save method on a model to ensure validation	162
Handling model validation errors in the view:	163
Django pattern: processing POST request in the same view as renders the form	165
Refactor: Removing hard-coded URLs	169
The {% url %} template tag	169
Using get_absolute_url for redirects	170
<b>10. A simple form.....</b>	<b>173</b>

Moving validation logic into a form	173
Switching to a Django ModelForm	175
Testing and customising form validation	176
Using the form in our views	178
Using the form in a view with a GET request	179
A big find & replace	180
Using the form in a view that takes POST requests	182
Using the form in the final view	184
A helper method for several short tests	185
Using the form's own save method	187
The request.POST or None trick	188
<b>11. More advanced Forms.....</b>	<b>191</b>
Another FT for duplicate items	191
Preventing duplicates at the model layer	192
A little digression on Queryset ordering and string representations	194
Handling validation at the views layer	196
A more complex form to handle uniqueness validation	198
Using the existing lists item form in the list view	199
Recap: what to test in views	201
<b>12. Database migrations.....</b>	<b>203</b>
South vs Django migrations	203
Creating an initial migration to match the current live state	204
Migrations: like a VCS for your database	205
Wrap-up: remove fake migration and git tag	209
On testing database migrations	209
Don't test third party code	210
Do test migrations for speed	210
Be extremely careful if using a dump of production data	210
<b>13. Dipping our toes, very tentatively, into JavaScript.....</b>	<b>211</b>
Starting with an FT	211
Setting up a basic JavaScript test runner	212
Using jquery and the fixtures div	215
Building a JavaScript unit test for our desired functionality	218
Columbo says: onload boilerplate and namespacing	220
<b>14. User authentication, integrating 3rd party plugins, and Mocking with JavaScript... ..</b>	<b>223</b>
Mozilla Persona (BrowserID)	224
Exploratory coding, aka “spiking”	224
De-Spiking	231

A common Selenium technique: waiting for	232
Reverting our spiked code	234
Javascript unit tests involving external components. Our first Mocks!	235
Why Mock?	236
Namespacing	236
A simple mock to unit tests our initialize function	237
More advanced mocking	243
Checking call arguments	246
Qunit setup and teardown, testing Ajax	246
More nested callbacks! Testing asynchronous code	250
<b>15. Server-side authentication and mocking in Python. ....</b>	<b>255</b>
Mocking in Python	255
De-spiking our custom authentication back-end: mocking out an internet request	260
1 if = 1 more test	261
patching at the Class level	262
Beware of Mocks in boolean comparisons	265
Creating a user if necessary	266
Tests the get_user method by mocking the Django ORM	266
Testing exception handling	268
A minimal custom user model	270
Tests as documentation	271
A slight disappointment	272
Fixing our cheat	273
The moment of truth: will the FT pass?	273
Finishing off our FT, testing logout	274
<b>16. Test fixtures and server-side debugging. ....</b>	<b>279</b>
Skipping the login process by pre-creating a session	279
Checking it works	280
The proof is in the pudding: using staging to catch final bugs	282
Staging finds an unexpected bug (that's what it's for!)	282
Setting up logging	283
Fixing the Persona bug	284
Managing the test database on staging	286
A Django management command to create sessions	286
An additional hop via subprocess	289
<b>17. Finishing "my lists": Outside-In TDD. ....</b>	<b>293</b>
The FT for "My Lists"	293
Outside-in TDD	294



The outside layer: presentation & templates	294
Moving down one layer to view functions (the controller)	295
Another pass, outside-in	295
A quick re-structure of the template inheritance hierarchy	296
Designing our API using the template	297
Moving down to the next layer: what the view passes to the template	298
The next “requirement” from the views layer	298
Moving down again: to the model layer	300
Final step: feeding through the .name API from the template	302
<b>A. PythonAnywhere.....</b>	<b>305</b>
<b>B. Django Class-Based Views.....</b>	<b>309</b>
<b>C. Provisioning with Ansible.....</b>	<b>317</b>

---

# Preface

If you're reading this sentence, then this is an early release, unfinished draft of this book, so don't be surprised if you find it a little short! I'm working on the rest of it.

With your help, I hope this can be a better book. So please, please, please do get in touch with any comments, feedback and suggestions. You can reach me directly via *obeythe testinggoat@gmail.com*.

You can also check out the website at [www.obeythetestinggoat.com](http://www.obeythetestinggoat.com), and follow me on Twitter via [@hjwp](https://twitter.com/hjwp)

I want to know about typos, about whether you think I'm going too fast or too slow, what you like and don't like, everything. I want to hear from you if you're a beginner, and whether it's working for you from a learning point-of-view. I want to hear from you if you've already done a lot of testing — maybe you think I'm doing it all wrong, maybe you think I'm focusing on the wrong things, maybe you think I've missed something important. I'm not claiming to be the world's foremost authority, so get in touch and help me improve things.

Thanks in advance. I hope you enjoy the book!

PS - if you're reading the free version of the book and you're enjoying it, you know, here's [a link from which you can buy the full thing](#), hint hint...



## On Chimera comments

If you're reading this via the Chimera online version, be aware that the platform is still under development, so it has a few missing features. For example, I don't get notified when people comment. So, if you have a question for which you want an immediate answer, email me rather than posting a comment here.

## Table P-1. Version history

- 0.1 First 4 chapters
- 0.2 Adds chapters 5 and 6, many typo corrections, and incorporates lots of other feedback. Huge thanks to Dave Pawson, Nicholas Tollervey and Jason Wirth and my editor Meghan Blanchette. Thanks also to Hansel Dunlop, Jeff Orr, Kevin De Baere, crainbf, dsisson, Galeran, Michael Allan, James O'Donnell, Marek Turnovec, SoonerBourne, julz and my mum! There are several changes to chapters 1-4, which would be worth looking at if you've been working from the previous draft.
- \* Look out for some clarifications to the pre-requisites below
  - \* In chapter 2, look out for the mention of `implicitly_wait`, the fix to the missing `if __name__ == '__main__':`, and the "TDD concepts" section at the end
  - \* In chapter 3 there's a little "useful commands & concepts" recap at the end.
  - \* Chapter 4 has a flowchart illustrating the TDD process, well worth a look before diving into chapters 5 & 6, which are quite meaty.
- 0.3 Python 3, styling and deployment.
- \* The entire book has been converted to Python 3. See the top of chapter 7 for what to do if you've been using Python 2 to date, and see the preface for updated installation instructions
  - \* Added Chapter 7, which talks about layout and styling, static files, using Bootstrap, and how it can be tested
  - \* Added Chapter 8 in which we deploy the application to a real web server. Call this "Devops" if you will. In this we cover nginx, gunicorn, upstart, virtualenvs and deployment automation using fabric. At each step we use our tests to check our setup against a "staging" site, and then use automated deployment for the production site.
- Huge thanks to Jonathan Hartley, Hynek Schlawack, Cody Farmer, William Vincent, and many others.
- 0.4 Forms and input validation
- Thanks to Emily Bache and Gary Bernhardt who convinced me to go for slightly more purist unit tests in chapters 5 onwards. Thanks to Russell Keith-Magee and Trey Hunner for their comments on appendix II, and some correlated improvements to ch. 9
- Thanks to all my other Early Release readers for your invaluable feedback and support.
- Warning: to all those that missed the previous update, the whole book has switched to Python 3. To update your codebase, my recommendation is to go back to the beginning of the book and just start again from scratch — it really won't take that long, it's much quicker the second time, and it's good revision besides. If you really want to "cheat", check out the appropriate branch (chapter\_XX) from my [github repo](#)
- 0.5 Django 1.6, better deployment, South migrations, Javascript
- \* Fully upgraded to Django 1.6. This simplifies chapter 3, 6, and 10 somewhat.
  - \* Tweaks to the deployment chapter, add a git tag.
  - \* (New) Chapter 12: Database Migrations. Currently uses South.
  - \* (New) Chapter 13: Dipping our toes into JavaScript
- Thanks to David Souther for his detailed comments on the JavaScript chapter, and to all the early release readers that have provided feedback: Tom Perkin, Sorchia Bowler, Jon Poler, Charles Quast, Siddhartha Naithani, Steve Young, Roger Camargo, Wesley Hansen, Johansen Christian Vermeer, Ian Laurain, Sean Robertson, Hari Jayaram, Bayard Randel, Konrad Korzel, Matthew Waller, Julian Harley, Barry McLendon, Simon Jakobi, Angelo Cordon, Jyrki Kajala, Manish Jain, Mahadevan Sreenivasan, Konrad Korzel, Deric Crago, Cosmo Smith, Markus Kemmerling, Andrea Costantini, Daniel Patrick and Ryan Allen.

- 0.6 Integrating a 3rd-party auth system (Persona), spiking, and mocking in Javascript and Python, server-side debugging, Outside-In TDD
- \* Add chapters 14, covering a “spike” (untested exploratory coding) and de-spike. More advanced JavaScript testing, using mocks
  - \* Chapter 15 covers mocking in Python, and customising Django authentication.
  - \* Chapter 16 does a little server-side debugging.
  - \* Chapter 17 finishes the user story with a discussion of Outside-In TDD.
- Thanks to Steve Young, Jason Selby, Greg Vaughan, Jonathan Sundqvist, Richard Bailey, Diane Soini, and many others — the mailing list is getting to be a real active community now, thanks to all!

The bottom entry is the version you’re reading now. This version history applies to the paid-for Early Release e-book version (thanks again if you’ve bought that!), not to the Chimera online version.

## 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, onto master. I’d say that I’m, at best, a journeyman TDD 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’ve 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 regular flossing 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 3 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 overridden. 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?”

I've second-guessed every rule, I've suggested every shortcut, I've demanded justifications for every seemingly pointless aspect of TDD, and I've come 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 test saves me from making a really silly logic error. And psychologically, it's made development a much less stressful process, and we produce code that we're pleased 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 websockets. 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, and 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 took a lot of justifying to convince myself of anything.

## Outline

I've split this book into three parts.

### *Part 1 (Chaps 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 2 (Chaps 7-13) - 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 3 (Chaps 14-20) - More advanced topics*

Mocking, integrating a 3rd. party authentication system, Ajax, test fixtures, Outside-in TDD and Continuous Integration (CI).

## Some pre-requisites

### Python 3 & 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, 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 on with this book on Mac, Windows or Linux. If you're on Windows, you can download Python 3 from [Python.org](https://python.org). If you're on a mac, you should already have Python 2 installed, but you'll need to install Python 3 manually. Again, have a look at [Python.org](https://python.org) If you're on Linux, I trust you to figure out how to get it installed. In the last two cases, be clear that you know how to launch Python 3 as opposed to 2.

If for whatever reason you are stuck on Python 2, you should find that all of the code examples can be made to work in Python 2.7, perhaps with a judiciously placed `__future__ import` or two.

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 I](#) before you get started.

In any case, I expect you to have access to Python, and to know how to launch it from a command-line (usually with the command `python3`), and how to edit a Python file and run it. Again, have a look at the 3 books I recommend above 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 (`pip3` or `pip-3.3`) 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. If you're not sure about those, you'll need to find a basic HTML tutorial — there are a few at [www.webplatform.org/](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 what a form is and how it might work, then you're probably OK.

## Required software installations:

Aside from Python, you'll need:

- **Firefox** the 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 brows-

ers, 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.

- **Git** the version control system. This is available for any platform, [GitHub](#) have some good installation instructions if you need them. Make sure the `git` executable is available from a command shell.
- **Pip** the Python package management tool. On Linux, you can install this as `python3-pip` under most package managers, or just Google for manual download + installation instructions. On Windows and Mac, things are a touch more complex, see boxes below.

To make sure we're using the Python3 version of pip, I'll always use `pip-3.3` as the executable in my command-line examples. Depending on your platform, it may be `pip-3`, or `pip3`, or maybe `pip-3.4` in future (NB - I haven't done any testing against Python 3.4 yet).

## Windows Notes

Windows users can sometimes feel a little neglected, since OS X and Linux make it so 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 on Windows, it will come with 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-slashes directory separators.
2. After you install Python 3, you'll need to add two directories to your system PATH: the main Python directory (eg `c:\Python33`) **and** its Scripts subfolder, `c:\Python33\Scripts`. You can do this via *Control Panel* → *System* → *Advanced* → *Environment Variables*. There are some instructions at [Python.org](#)
3. On windows, Python 3's executable is called `python.exe`, which is exactly the same as Python 2. Similarly, you can easily end up with two different versions of pip. To avoid any confusion, create a file in your home folder (usually `C:\Users\your-username`) called `.bashrc`, and add the lines:

```
alias python3='c:\\Python33\\python.exe'
alias pip-3.3='c:\\Python33\\Scripts\\pip.exe'
```

You'll need to close your Git bash window and open a new one for this to take effect. It will only work in Git bash, not in the regular Dos command prompt.

4. To install pip, just google "Python 3 Pip", and follow the instructions, *making sure to always use python3 whenever you're running the setup scripts*. At the time of writing, the simplest solution was described in [This stackoverflow post](#) and involves



downloading some user-contributed installers for **setuptools** and **pip** from the web. As of Python 3.4, this is all set to be simplified substantially. We all look forward to that day!

The test for all this is that you should be able to go to a command prompt and just run `python3` from any folder. Once you've installed `pip` and `Django` (see below), you should also be able to just run `pip-3.3` and `django-admin.py` from any folder too.

## MacOS Notes

Macs are a bit more sane than Windows, but can still be a little twitchy, particularly as regards getting `pip-3.3` installed.

My recommendation is to **use Homebrew**; it seems to be a requirement to get a decent dev. setup on a Mac. Check out **brew.sh** for installation instructions. Once installed, it'll prompt you to go through a few setup steps. You'll need to install XCode from the app store, which means signing up for a Mac developer ID.

Once that's all done, you'll be able to install Python3 and `pip-3.3` with one simple command:

```
brew install python3
```

Similarly to windows, the test for all this is that you should be able to open a terminal and just run `python3` from anywhere. Once you've installed `pip` and `Django` (see below), you should also be able to just run `pip-3.3` and `django-admin.py` from any folder too.

## 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 `i` 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**



Did these instructions not work for you? Or have you got better ones?  
Get in touch! [obeythetestinggoat@gmail.com](mailto:obeythetestinggoat@gmail.com)

## 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 1.6** (`pip-3.3 install django==1.6.1`). This is our web framework. You should make sure you have version 1.6 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.
- **Selenium** (`pip-3.3 install --upgrade selenium`), a browser automation tool which 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 know what you're doing, don't worry about using a `virtualenv`. We'll talk about them 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, 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.

Onto 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.

## *\_Constant width italic\_*

Shows text that should be replaced with user-supplied values or by values determined by context.

```
# code listings and terminal output will be listed in constant width paragraphs
$ commands to type will be in bold
Occasionally I will use the symbols:
```

[...]

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 icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

TODO: this is a note to myself that there is something unfinished, or an idea that I might want to incorporate later. These are good things to send me feedback on! They should all be gone by the time the book is finished...

# 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\*](mailto:bookquestions@oreilly.com).

You can find errata, examples, and additional information at [\*http://www.oreilly.com/catalog/<catalog page>\*](http://www.oreilly.com/catalog/<catalog page>).

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

Facebook: [\*http://facebook.com/oreilly\*](http://facebook.com/oreilly)

Twitter: [\*http://twitter.com/oreillymedia\*](http://twitter.com/oreillymedia)

YouTube: [\*http://www.youtube.com/oreillymedia\*](http://www.youtube.com/oreillymedia)



---

# Getting Django set up using a Functional Test

Test-Driven Development (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.



Figure 1-1. Goats are more agile than you think (credit: [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.

```
from selenium import webdriver  
  
browser = webdriver.Firefox()  
browser.get('http://localhost:8000')  
  
assert 'Django' in browser.title
```

*functional\_tests.py.*

## 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, is 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

That's pretty much as simple as it could get. 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, try and open *localhost:8000*, and then the Python error message. And then, you will probably have been 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 [required installations section](#) of the preface.

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 the pre-requisites in the preface 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:

```
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 1.6, using settings 'superlists.settings'
```

Development server is running at <http://127.0.0.1:8000/>  
Quit the server with CONTROL-C.

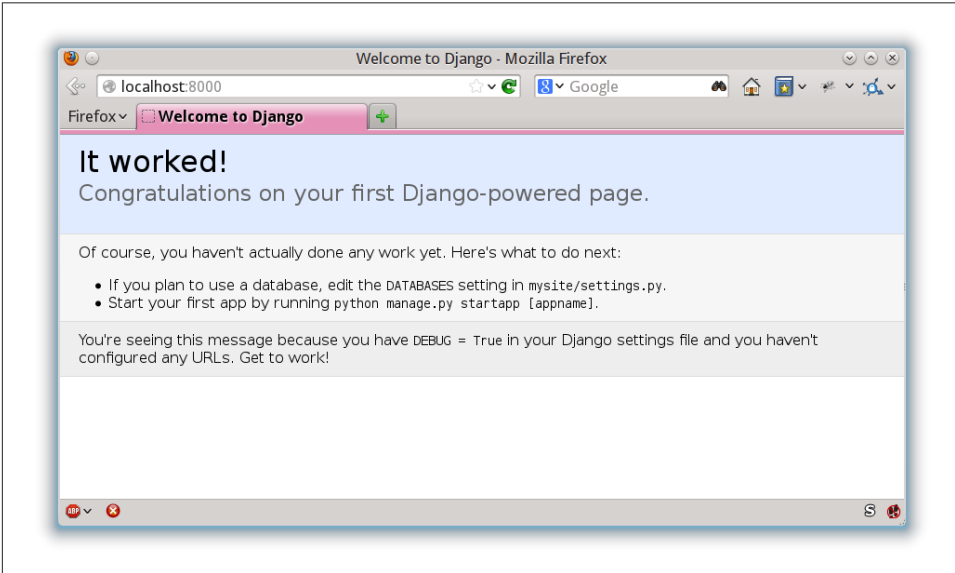
Leave that running, and open another command shell. In that, we can try running our test again (from the folder we started in):

```
$ 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/
```

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



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*.

```
$ ls
manage.py      superlists      functional_tests.py
$ git add .
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   functional_tests.py
#       new file:   manage.py
#       new file:   superlists/__init__.py
#       new file:   superlists/__pycache__/__init__.cpython-33.pyc
#       new file:   superlists/__pycache__/settings.cpython-33.pyc
#       new file:   superlists/__pycache__/urls.cpython-33.pyc
#       new file:   superlists/__pycache__/wsgi.cpython-33.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* (a special file that tells git, um, what it should ignore)

```
$ git rm -r --cached superlists/__pycache__
rm 'superlists/__pycache__/__init__.cpython-33.pyc'
rm 'superlists/__pycache__/settings.cpython-33.pyc'
rm 'superlists/__pycache__/urls.cpython-33.pyc'
rm 'superlists/__pycache__/wsgi.cpython-33.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`... Am not telling you how to do that though, I leave you to discover the secrets of git aliases on your own!)

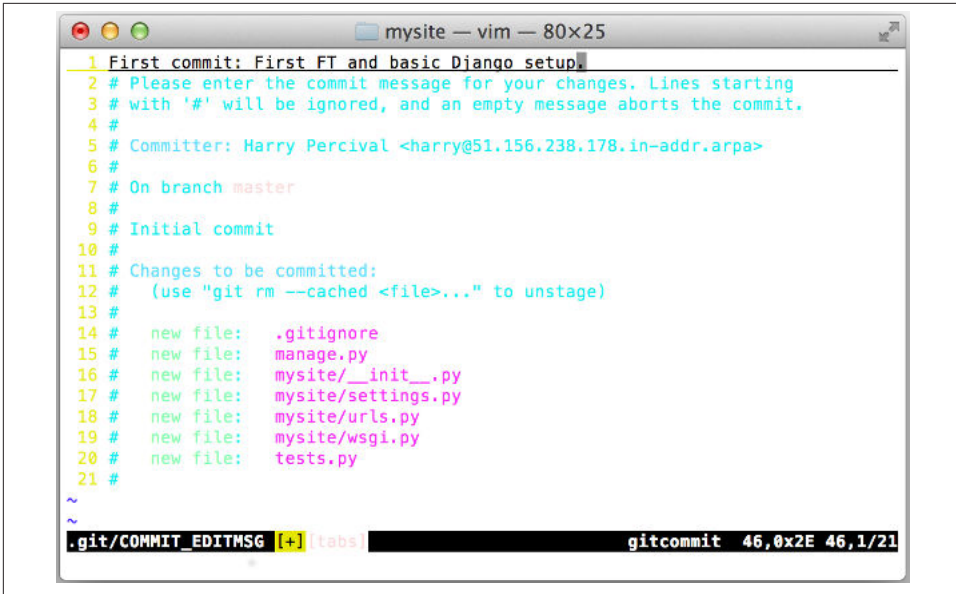
```
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   functional_tests.py
#       new file:   manage.py
#       new file:   superlists/__init__.py
#       new file:   superlists/settings.py
#       new file:   superlists/urls.py
#       new file:   superlists/wsgi.py
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       .gitignore
```

OK, we'll just add *.gitignore*, and then we're ready to do our first commit!

```
$ git add .gitignore
$ 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](#)<sup>1</sup>:

1. Did `vi` pop up and you had no idea what to do? Go and take another look at the preface, there are some brief instructions



```
1 First commit: First FT and basic Django setup.
2 # Please enter the commit message for your changes. Lines starting
3 # with '#' will be ignored, and an empty message aborts the commit.
4 #
5 # Committer: Harry Percival <harry@51.156.238.178.in-addr.arpa>
6 #
7 # On branch master
8 #
9 # Initial commit
10 #
11 # Changes to be committed:
12 #   (use "git rm --cached <file>..." to unstage)
13 #
14 #   new file:   .gitignore
15 #   new file:   manage.py
16 #   new file:   mysite/__init__.py
17 #   new file:   mysite/settings.py
18 #   new file:   mysite/urls.py
19 #   new file:   mysite/wsgi.py
20 #   new file:   tests.py
21 #
~
~
.git/COMMIT_EDITMSG [+](tabs) gitcommit 46,0x2E 46,1/21
```

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. At the time of writing, there were some called GitHub and 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.

OK that's it for the VCS lecture. So, congratulations! You've written a functional test using Selenium, and you've got Django installed and running, in a certifiable, test-first, goat-approved TDD way. Give yourself a well-deserved pat on the back before moving onto 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 lists have 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 the 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* (or FTs) <sup>1</sup>.

---

1. I should say that some people can get a bit precious about terminology, and might prefer the term *Acceptance tests*, or want to talk about *Integration tests* too. In the JavaScript world they talk about *E2E* tests, meaning "End-to-End". The main point is that these kinds of tests look at how the whole application functions, from the outside.

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.

FTs should have a human-readable story that we can follow, and we can do that by using comments that accompany the test code. When creating a new FT, we can write those comments first, to capture the key points of the User Story. Being human readable, you could even share them with non-programmers, 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:

```
from selenium import webdriver functional_tests.py.

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, 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 a topic for another book. (TODO or maybe an appendix/ chapter at the end?)

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 at least written the test correctly.

## The Python standard library's `unittest` module

But there's 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 it would clear them 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`.

- ④ 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.
- ② ③ The `setUp` and `tearDown` methods. These 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 get run even if there's an error during the test itself<sup>2</sup>. No more Firefox windows left lying around!
- ⑤ 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](#)
- ⑥ `self.fail` just fails no matter what, producing the error message given. I'm using it as a reminder to finish the test.
- ⑦ Finally, 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.
- ⑧ the `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 without 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)
-----
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)
```

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

```
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!

## Implicitly wait

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

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

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

*functional\_tests.py.*

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: whenever we ask Selenium to find something on the page, if it can't find it, it will now wait up to 3 seconds for it to appear.

## Commit

This is a nice 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 a way that we expected it to



# 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 to 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 & 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!



this chapter depends on Django 1.6. If you started with an old version of the book that used Django 1.5, upgrade now with a `pip install --upgrade django`

## 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 might re-use the same app in different projects, and you can use third-party apps developed by other people... Well, that doesn’t happen for every project, but apps can still be a good way of trying to structure your code.

Let’s start an app for our 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 admin, models, views and, of immediate interest to us, tests.

```
superlists/
├── functional_tests.py
├── lists
│   ├── admin.py
│   ├── __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, whereas 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 point of view of the user
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 re-run 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 homepage view then. Open up the new file at *lists/tests.py*, and you'll see something like this:

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

*lists/tests.py.*

```
# 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`, we'll find out what's useful about it later.

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 whatever 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:

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

*lists/tests.py.*



```
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'...
```



If you see error messages about `settings.DATABASES` being improperly configured, you are probably not working with Django 1.6, or it's possible your settings file was created by Django 1.5, and you upgraded to 1.6 half-way through. The simplest thing to do is probably to start again from scratch, and just zip through the first couple of chapters again.

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 no doubt you've guessed, the `-m` flag lets you pass in a commit message at the command-line, so you don't need to go via 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 documentation](#).

But, 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:

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

So we want to test two things:

1. Can we resolve the URL for the root of the site ("/") to a particular view function we've made?
2. 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  
  
class HomePageTest(TestCase):  
  
    def test_root_url_resolves_to_home_page_view(self):  
        found = resolve('/')  
        self.assertEqual(found.func, home_page)
```

What's going on here?

- `resolve` is the actual 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`.
- 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`:

```
from django.shortcuts import render                                lists/views.py

# 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 considerable emotion) 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/python3.3/dist-packages/django/core/urlresolvers.py",
line 453, in resolve
    return get_resolver(urlconf).resolve(path)
  File "/usr/local/lib/python3.3/dist-packages/django/core/urlresolvers.py",
line 333, in resolve
    raise Resolver404({'tried': tried, 'path': new_path})
django.core.urlresolvers.Resolver404: {'path': '', 'tried': [[<RegexURLResolver
<RegexURLPattern list> (admin:admin) ^admin/>]]}]
-----
```

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

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

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

## Reading tracebacks

A brief aside on reading tracebacks from unit tests, since it's something we do a lot of 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/python3.3/dist-packages/django/core/urlresolvers.py",
line 453, in resolve
    return get_resolver(urlconf).resolve(path)
  File "/usr/local/lib/python3.3/dist-packages/django/core/urlresolvers.py",
line 333, in resolve
    raise Resolver404({'tried': tried, 'path': new_path})
django.core.urlresolvers.Resolver404: {'path': '/', 'tried': [[<RegexURLResolver❸
<RegexURLPattern list> (admin:admin) ^admin/>]]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. 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, we'll see an example of this fourth step later in the book.

For now though, we finish up by interpreting 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:

```
from django.conf.urls import patterns, include, url
# ...

from django.contrib import admin
admin.autodiscover()

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:

```
from django.conf.urls import patterns, include, url
# ...

# from django.contrib import admin
# admin.autodiscover()

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 commented-out 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')),
```

And run the unit tests again, **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 the 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's 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. It's the quickest, 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

Onto 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() #1
        response = home_page(request) #2
        self.assertTrue(response.content.startswith(b'<html>')) #3
        self.assertIn(b'<title>To-Do lists</title>', response.content) #4
        self.assertTrue(response.content.endswith(b'</html>')) #5
```

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 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 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 — 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:

- in the terminal, run the unit tests and see how they fail
- 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:

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

*lists/views.py.*

- Tests:

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

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

```
from django.http import HttpResponse  
  
# Create your views here.  
def home_page(request):  
    return HttpResponse()
```

*lists/views.py.*

- 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 the 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*

- Run the unit tests in the terminal

- Make a minimal code change in the editor
- Repeat!

---

# 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

It's worth reminding ourselves of what the prize is. Do you remember my story about my first ever project, which started out just fine but soon accumulated a mass of technical debt, messy code, invisible dependencies and became impossible to refactor or maintain? I can't convey to you how different an experience it is to work on a project with TDD. You never have to worry about refactoring or experimenting — the tests will tell you if you make a mistake. You never have to worry about unexpected regressions. You never have to sit around robotically clicking through the same pages on your site, checking that they work. I'm constantly surprised at all the stupid little logic errors that unit tests have saved us from too.

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 (one of the great legends of 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 credit *Allie Brosh, Hyperbole and a Half*)

OK, so 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

Yes, 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 techniques 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-wavey 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 re-run 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 re-run 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*.

Also, just look at that `any` function. It's a little-known Python builtin. 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 "list comprehension generator expression", which is something I'll let you Google. Come back and tell me that's not pure joy!

Let's see how it gets on (don't forget to start up the dev server with `python3 manage.py runserver` first)

```

$ 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.

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: 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:

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

*lists/templates/home.html.*

Mmmh, syntax-highlighted... Much nicer! Now to change our view function:

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

*lists/views.py.*

```
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 in substituting in Python variables into HTML text. We're not using this feature yet, but we will do 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 builtin `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/__init__.py",
line 53, in render
    return HttpResponse(loader.render_to_string(*args, **kwargs),
  File "/usr/local/lib/python3.3/dist-packages/django/template/loader.py", line
162, in render_to_string
    t = get_template(template_name)
  File "/usr/local/lib/python3.3/dist-packages/django/template/loader.py", line
138, in get_template
    template, origin = find_template(template_name)
```

```
File "/usr/local/lib/python3.3/dist-packages/django/template/loader.py", line
131, in find_template
    raise TemplateDoesNotExist(name)
django.template.base.TemplateDoesNotExist: home.html④
```

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

Another chance to analyse a traceback

- ④ We start with the error: it can't find the template
- ① Then we double-check what test is failing: sure enough, it's our test of the view HTML
- ② Then we find the line in our tests that caused the failure: it's when we call the `home_page` function
- ③ 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`:

```
# Application definition
superlists/settings.py.

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 <sup>1</sup>. 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:

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

*lists/tests.py.*

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:

```
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)
```

*lists/tests.py.*

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!

1. 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

# On refactoring

That was an absolutely trivial example of refactoring. Yes, we probably could have skipped a few of the steps in between. But once again, this is all about learning a discipline, starting with simple examples. The way Kent Beck puts it is:

Am I recommending that you actually work this way? No. I'm recommending that you be *able* to work this way.

— Kent Beck  
*TDD by example*

We're unlikely to go wrong when it's such a simple example, but when you get into refactoring more complex and sensitive code, the step-by-step approach can make sure you never get into trouble, and you always go from working code to working code.

In fact as I was writing this my first instinct was to dive in and change the test first — make them use the `render_to_string` function straight away, delete the 3 superfluous assertions and just check 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 easily 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](#) (Google it), stick to small steps, keep refactoring and functionality changes entirely separate.



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>`:

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

*lists/templates/home.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...

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

*lists/templates/home.html.*

And now?

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

We add our placeholder text...

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

*lists/templates/home.html.*

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...

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

*lists/templates/home.html.*

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*:

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

*functional\_tests.py.*

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?

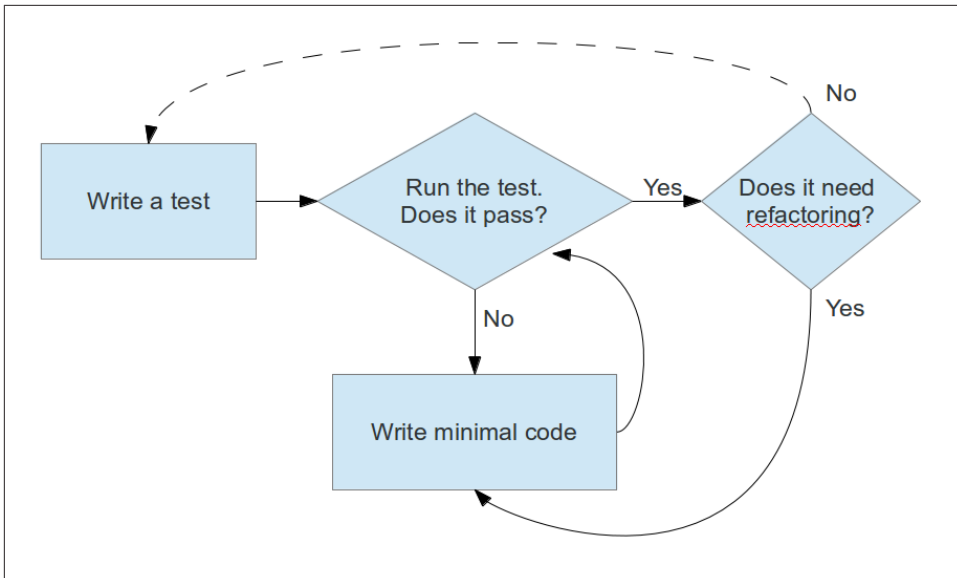


Figure 4-2. Overall TDD process

We write a test. We run the test and see it fail. We write some minimal code to get it a little further. We re-run the tests 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:



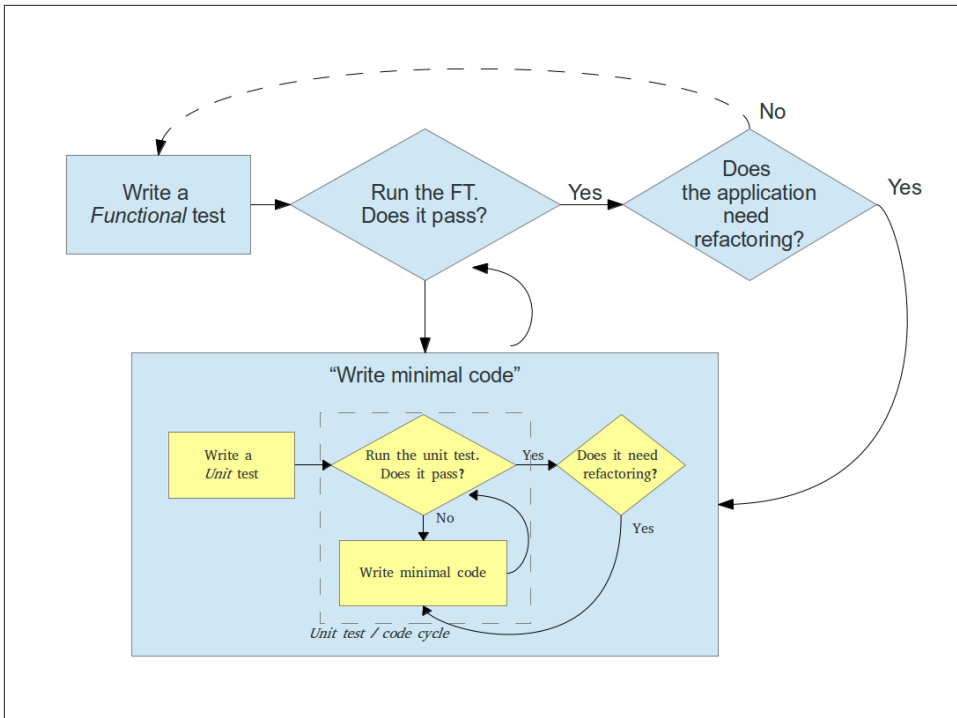


Figure 4-3. The TDD process with Functional and Unit tests

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 work-flow 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](https://github.com/hjwp/book-example/tree/chapter_03)
- Chapter 4: [https://github.com/hjwp/book-example/tree/chapter\\_04](https://github.com/hjwp/book-example/tree/chapter_04)
- Chapter 5: [https://github.com/hjwp/book-example/tree/chapter\\_05](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, 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_05.zip) [https://github.com/hjwp/book-example/archive/chapter\\_06.zip](https://github.com/hjwp/book-example/archive/chapter_06.zip)



Try not to use my repo as a crutch

You really only want to sneak a peak at the answers if 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.

Also, as regards skipping ahead, I didn’t design the chapters to stand alone, each relies on the previous ones, so skipping ahead may not work well.



---

## 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 them 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, 3 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*:

```
<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">
```

*lists/templates/home.html.*

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:

```
# 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')
```

*functional\_tests.py.*

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:

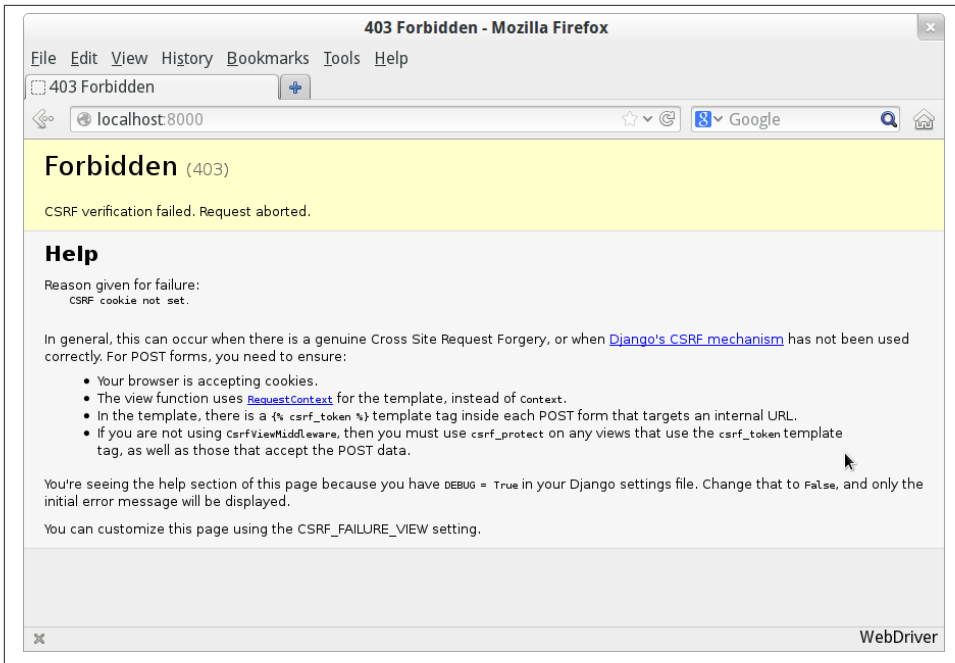


Figure 5-1. Django DEBUG page showing CSRF error

## 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 take 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 3 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. Re-running 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, i.e. `/`, 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, and then adapted it to add our POST request, then 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 3 lines at the beginning which set up the test, 1 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:

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

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

*lists/views.py.*

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.

We've already had a hint of it, 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.

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:

```
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)
```

*lists/tests.py.*



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.

How do we then use it in the actual template? The syntax is `{{ ... }}`, which displays a variable 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>
```

Now, 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 re-write our view, and tell it to pass the POST parameter to the template:

*lists/views.py.*

```
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*... in a different test! We’ve 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. Of course 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:

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

*lists/views.py.*

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
```

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

```
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,  
    )  
)
```

*functional\_tests.py.*

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`:

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

*functional\_tests.py.*

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:

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

*lists/templates/home.html.*

## 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 inputting a *second* list item gives us a “2:”

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 & 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 error with:

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

## 3 strikes and refactor

Before we go further — we’ve got a bad “code smell” in this FT. We’ve got 3 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 “3 strikes and refactor”. You can copy & paste code once, and it may be premature to try and remove the duplication it causes, but once you get 3 occurrences, it’s time to remove duplication.



A “code smell” is something about a piece of code that makes you want to re-write 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...

We start by committing what we have so far. Even though we know our site has a major flaw - it can only handle 1 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 — 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.

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

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

*functional\_tests.py.*

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

## The Django ORM & 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`, which we can call further functions on, like `.count()`, and also extract individual objects. 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 in 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:

```

from django.db import models

```

*lists/models.py.*

```

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, we make it inherit from the Django `Model` class:

```
from django.db import models lists/models.py.

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

Now the test actually gets surprisingly far:

```
self.assertEqual(first_saved_item.text, 'The first (ever) list item')
AttributeError: 'Item' object has no attribute 'text'
```

That’s a full 8 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.

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:

```
class Item(models.Model): lists/models.py.
    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](#).

In any case, the unit tests now pass, so let’s do a commit for our first ever model!

```
$ git status # see tests.py and models.py have changed
$ git diff # see actual changes to tests.py and models.py
$ git commit -am"Created model for list Items"
```

## A bold claim about the correctness of code listings in this book

A lot of people get to some point in this chapter or the next one and see results different from what I say they should be, and after a bit of double-checking, finding nothing obviously wrong with what they did, they start to think “Harry’s probably made a mistake in his code listings. The book is wrong”.

Well, if you’ll allow me a moment of hubris: I don’t reckon I have, and I don’t think it is. :-P

You see, every single code listing in this book is checked by an extensive suite of automated tests. Whenever I say “enter this code, then run this command, then you’ll see this”, I actually have a test somewhere that opens up my book, parses it for code listings, enters them into files on disk, runs the actual commands listed in the book, and checks the output character by character against the expected output in the book.

So, if you spot what you think is a real mistake (in the official, paid-for pdf of the book, not the bleeding-edge free online version), do send it in to me. If it’s a real mistake, I’ll eat my hat. And send you free a signed copy. With a correction in red marker pen and a very embarrassed note.

But first - check closely. Did you really do *exactly* what I say? Are you sure there’s not a missing comma or trailing slash or a file in the wrong place or *something*?

## 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 3 new lines (❶) to the existing test called `test_home_page_can_save_a_POST_request`

```
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.all().count(), 1) #❶  
    new_item = Item.objects.all()[0] #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)
```

*lists/tests.py.*

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:

- Code smell: POST test is too long?



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

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

Let's adjust our view:

```
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', ''),
    })
```

*lists/views.py.*

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. La la la la...

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 re-think.

But the point is that we *don't* always spot problems in advance, and sometimes the implications aren't obvious. What I'm demonstrating here is the way that TDD can help guide you to the right answer, even when you don't catch problems in advance.

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

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

*lists/views.py.*

Let's have a little look at our own to-do list. I've added a couple of the other things that are on our mind:

- Don't save blank items for every request
- Code smell: POST test is too long?
- Display multiple items in the table

- Support more than one list!

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:

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

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

*lists/tests.py.*

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:

```
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
    })
```

*lists/views.py.*

- ❶ ❷ we use a variable called `new_item_text`, which will either hold the POST
- ❸ ❶ contents, or the empty string
- ❸ ❹
- ❷ `.objects.create` is a neat shorthand for creating a new `Item`, without needing to call `.save()`.

And that gets the test passing.

## 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 homepage.

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

*lists/tests.py.*

```

response = home_page(request)

self.assertEqual(Item.objects.all().count(), 1)
new_item = Item.objects.all()[0]
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:

```

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')
```

*lists/views.py (ch05l028).*

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. Now I'm not a rigid purist when it comes to unit tests, so I don't mind breaking this rule now and again, but this really does feel like a good place to separate out our concerns:

```

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

*lists/tests.py.*

```

response = home_page(request)

self.assertEqual(Item.objects.all().count(), 1)
new_item = Item.objects.all()[0]
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 6 tests pass instead of 5:

```
Ran 6 tests in 0.010s
```

```
OK
```

## Rendering items in the template

Much better! Back to our to-do list:

- Don't save blank items for every request
- Code smell: POST test is too long?
- Display multiple items in the table
- Support more than one 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:

```

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())

```

*lists/tests.py.*

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?

```
[...]
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"

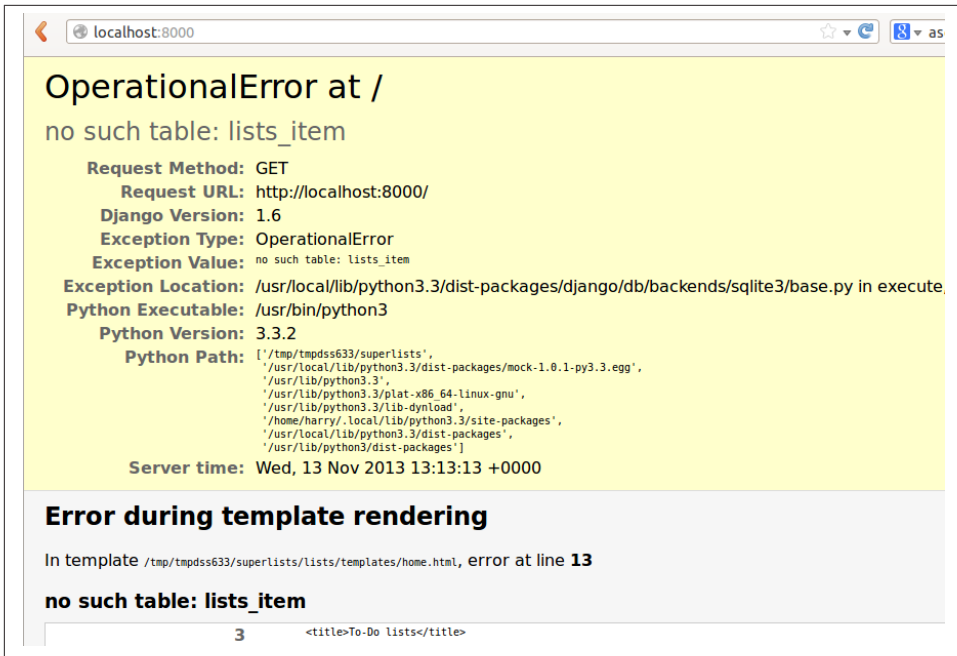


Figure 5-2. Another helpful debug message

## Creating our production database with syncdb

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:

```
[...]
# Database
# https://docs.djangoproject.com/en/1.6/ref/settings/#databases

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

*superlists/settings.py.*

We've told Django everything it needs to create the database, including what tables and columns it needs, in *models.py*. It's time for another Django Swiss army knife `manage.py` command, `syncdb`:

```
$ python3 manage.py syncdb
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
Creating table lists_item
```

You just installed Django's auth system, which means you don't have any superusers defined.

Would you like to create one now? (yes/no):

**no**

Installing custom SQL ...

Installing indexes ...

Installed 0 object(s) from 0 fixture(s)

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.<sup>1</sup>

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

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

```
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**:

---

1. if you get a different error at this point, try restarting your dev server — it may have gotten confused by the `syncdb`

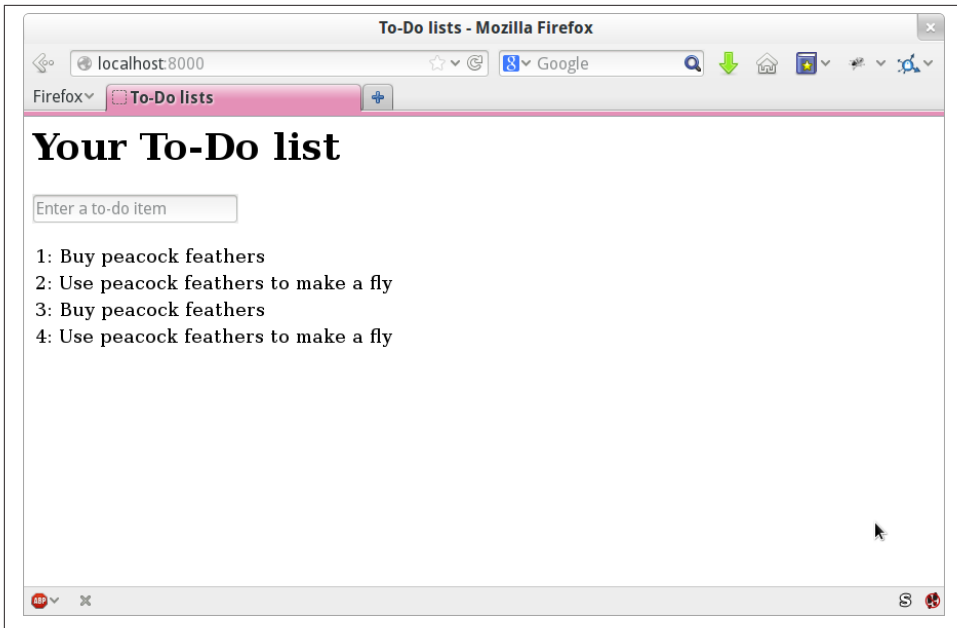


Figure 5-3. There are list items left over from the last run of the test

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 syncdb:

```
$ rm db.sqlite3
$ python3 manage.py syncdb --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*, *views.py* and *settings.py*. Of those, the first three belong together, whereas adding the database name to *settings.py* probably belongs separately. We'll also want to add the database file to our list of ignored files:



```
$ git add lists
$ git commit -m"Redirect after POST, and show all items in template"
$ git add superlists/settings.py
$ echo "db.sqlite3" >> .gitignore
$ git add .gitignore
$ git commit -m"Name database in settings.py, add it to .gitignore"
```

TODO: as of Django 1.6, no need to add superlists/settings.py cos database name is still the default. Dbl-check this.

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 3 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).

*3 strikes and refactor*

A rule of thumb for when to remove duplication from code.

*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.



---

# 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.



this chapter depends on Django 1.6. If you started with an old version of the book that used Django 1.5, upgrade now with a `pip install --upgrade django`

## 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 and we'll probably need to move off it later, it's dead useful at this stage, so let's check it out.

## A hand-rolled implementation of test database cleanup

For those a little more familiar with Django and/or testing, here's a little aside on how to do test isolation in functional tests manually.

You may not always have the luxury of a tool like `LiveServerTestCase`, for example if you decide to use a framework other than Django. I once wrote some code for an old version of a TDD tutorial, using a combination of:

- switching to a test database by overriding `settings.py`, using a file called `settings_for_fts.py` which imports everything from `settings.py`, but changes the `DATABASES` entry to have a different `NAME`
- Then run `'manage.py'` with a `settings=settings_for_fts` flag at the command line.
- in `setUp`, use `subprocess.Popen` to do a `syncdb` against the test database
- and while we're at it, `setUp` can also do a `manage.py runserver`, again using `settings_for_fts`, to start up the test server automatically.
- `tearDown` would kill the server and delete the database file after each test.

You can explore the implementation at my Github page for the Test-Driven-Django-Tutorial, on a tag called [old-ft-runner](#)

`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):

```
$ 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
│   ├── 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`:

```

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

class NewVisitorTest(LiveServerTestCase):

    def setUp(self):
        [...]

```

*functional\_tests/tests.py (ch06l001).*

Next,<sup>1</sup> instead of hard-coding the visit to localhost port 8000, `LiveServerTestCase` gives us an attribute called `live_server_url`:

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 testing book's tests. You know, the tests for the tests in the book about testing. They have tests of their own, incidentally.

```

                                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!
-----

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 "move functional_tests to functional_tests 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!).

## 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
(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!

-----
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:

```

# 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!')
functional_tests/tests.py.

```



*# 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 URLs 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 strong reaction against “Big Design Up-Front”: the traditional software engineering practice whereby, after a lengthy requirements gathering exercise, there was an equally lengthy design stage where the software was 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 ain't 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.

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

## REST

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, like

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

That will fulfil 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:

- ~~Get FTs to clean up after themselves~~
- Adjust model so that items are associated with different lists
- Add unique URLs for each list
- Add a URL for creating a new list via POST
- Add URLs for adding a new item to an existing list via POST

# Implementing the new design using TDD

How do we use TDD to implement the new design? Let's take another look at the flow-chart for the TDD process:

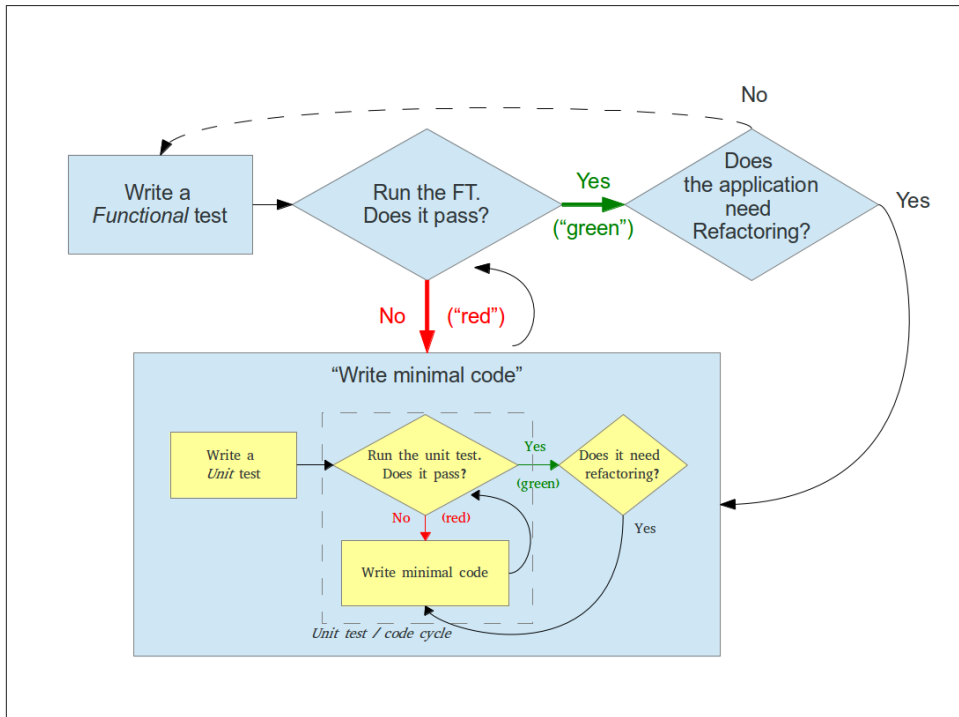


Figure 6-1. The TDD process with Functional and Unit tests

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 re-writing 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.

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:

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

*functional\_tests/tests.py.*

```

# 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 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.
self.browser.quit()
## We use a new browser session to make sure that no information
## of Edith's is coming through from cookies etc #❶
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, she goes back to sleep
```

- ❶ I'm using the convention of double-hashtags (##) 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 `pip-3.3 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 6 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 4 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:

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

*lists/tests.py.*

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/'
```

Now we can go adjust our `home_page` view in *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})
```

*lists/views.py.*

Of course that will now totally break the functional test, 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!

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 I've shown how you can test your URL resolution explicitly, and how to test view functions by actually calling them, and checking that they render your templates correctly too. Django actually provides us with a little tool that can do all three, 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 you now know how to test them.

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:

```
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
```

*lists/tests.py (ch06l009).*

- 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 know 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 integration 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
```

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.

```
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)),  
    [...])
```

*superlists/urls.py.*

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.
```

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

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

*lists/views.py.*

Now we get

```
ValueError: The view lists.views.view_list didn't return an HttpResponse  
object.
```

Let's copy the two last lines from the *home\_page* view and see if they'll do the trick:

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

*lists/views.py.*

Re-run 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']
```

Now it's time for a little tidying up. In the Red/Green/Refactor dance, we've got to green, it's time to 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:

```
$ egrep "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 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.

We can start by deleting 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
```

Now 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 new template:

```
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:

```
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 re-use 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 re-run the unit tests to check that hasn't broken anything... Good...

Now 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')
```

Re run 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*:

```
<form method="POST" action="/">
```

*lists/templates/list.html (ch06l019).*

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"
```

## A URL and view for adding list items

Where are we with our own to-do list?

- ~~Get FTs to clean up after themselves~~
- Adjust model so that items are associated with different lists
- Add unique URLs for each list
- Add a URL for creating a new list via POST
- Add URLs for adding a new item to an existing list via POST

Hmm, well, 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.

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:

```
class NewListTest(TestCase):
```

*lists/tests.py (ch06l021-1).*

```
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.

`class NewListTest(TestCase):` *lists/tests.py (ch06l021-2).*

```
def test_saving_a_POST_request(self):
    self.client.post(
        '/lists/new',
        data={'item_text': 'A new list item'})
    )
    self.assertEqual(Item.objects.all().count(), 1)
    new_item = Item.objects.all()[0]
    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/')
    )
```

Try running that:

```
self.assertEqual(Item.objects.all().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.

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`:

```
def new_list(request): lists/views.py.
    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`

```
def new_list(request): lists/views.py.
    return redirect('/lists/the-only-list-in-the-world/')

```

Which gives:

```
self.assertEqual(Item.objects.all().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`:

```
def new_list(request): lists/views.py.
    Item.objects.create(text=request.POST['item_text'])
    return redirect('/lists/the-only-list-in-the-world/')

```

And now another look at this 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:

```
def test_redirects_after_POST(self): lists/tests.py.
    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.

OK

We're looking good. Can we remove the old `if request.method == POST` code from `home_page`?

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

*lists/views.py.*

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 re-run the tests to make sure...

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

Finally, let's wire up our two forms to use this new URL. In *both* `home.html` and `lists.html`:

```
<form method="POST" action="/lists/new">
```

*lists/templates/home.html, lists/templates/list.html.*

And we re-run our FTs to make sure everything still works...

```
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
```

## Adjusting our models

Enough housekeeping with our URLs. It's time to bite the bullet and change our models. Let's adjust our unit tests. Just for a change, I'll present the changes in the form of a diff:

```
@@ -3,7 +3,7 @@ from django.http import HttpRequest
from django.template.loader import render_to_string
from django.test import TestCase
```

*lists/tests.py.*

```

-from lists.models import Item
+from lists.models import Item, List
from lists.views import home_page

class HomePageTest(TestCase):
@@ -60,22 +66,32 @@ 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_lists = List.objects.all()
+        self.assertEqual(saved_lists.count(), 1)
+        self.assertEqual(saved_lists[0], 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_lists[0]` 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

Time for another unit-test/code cycle. I'm just going to show the test errors for the first couple, and let you figure out for yourself what the code should be:

```
ImportError: cannot import name List

...

AttributeError: 'List' object has no attribute 'save'

...

self.assertEqual(first_saved_item.list, list_)
AttributeError: 'Item' object has no attribute 'list'
```

How do we give our Item a list attribute? Let's just try making it like the text attribute:

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

*lists/models.py.*

That gives us:

```
AssertionError: 'List object' != <List: List object>
```

Not quite — 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:

```
from django.db import models

class List(models.Model):
    pass

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

*lists/models.py.*



This change to models.py won't filter through to the “real” database at *db.sqlite3* unless you delete it, and do a `syncdb --noinput`, like we did in the last chapter. The test database `manage.py test` gets recreated with each test run, so it's fine. We'll learn more about managing database schema updates in Chapter 12.

Now what happens?

```
$ python3 manage.py test lists
[...]
ERROR: test_displays_all_items (lists.tests.ListViewTest)
django.db.utils.IntegrityError: lists_item.list_id may not be NULL
[...]
ERROR: test_redirects_after_POST (lists.tests.NewListTest)
```



```

django.db.utils.IntegrityError: lists_item.list_id may not be NULL
[...]
ERROR: test_saving_a_POST_request (lists.tests.NewListTest)
django.db.utils.IntegrityError: lists_item.list_id may not be NULL

```

Oh gawd! Well, our model tests are passing but three of our view tests are failing, because Items have to be associated with a list now. 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:

```

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_)

```

*lists/tests.py.*

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 15, 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:

```

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/')

```

*lists/views.py.*

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 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.

And, again, you don't *always* have to code like this. When things are simple, you probably *can* get away with doing several steps at once. What we're doing here is practising for the hard cases — one occasion that pops into my head is a recent one where we

decided to refactor the payment processing system at work. You can bet we were extremely careful when we worked through that, but thanks to using small steps, we got it right first time.

Anyway, just to reassure ourselves that things have worked, we can re-run 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
$ git diff
$ git commit -a
```

## 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`, and have one test that checks the correct list object to the template, and then another, more long-winded one that sanity-checks that the right things actually appear as rendered by the template:

```
lists/tests.py.

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, that `%d` may be a little confusing? Dive into Python has a [good overview](#), if you want to go look them up quickly...

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

It's time to learn how we can pass parameters from URLs to views:

```
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)),
)
```

*superlists/urls.py.*

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
```

Let's start by fixing those. We can fix that easily with a dummy parameter in *views.py*

```
def view_list(request, list_id):
```

*lists/views.py.*

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:

```
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})
```

*lists/views.py.*

Now we get errors in our 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. Hm, it looks like it hasn't been adjusted to the new world of Lists and Items. In fact, this brings to mind the fact that we actually need to treat the creation of *new* lists differently from the addition of new items to *existing* lists.

Let's adjust the test to the new world, showing that it expects this view to create a brand new list:

```
def test_redirects_after_POST(self):
    response = self.client.post(
        '/lists/new',
        data={'item_text': 'A new list item'})
    new_list = List.objects.all()[0]
    self.assertRedirects(response, '/lists/%d/' % (new_list.id,))
```

*lists/tests.py (ch06l036-1).*

That still gives us the *invalid literal* error. Let's take a look at the view itself, and change it so it redirects to a valid place:

```
return redirect('/lists/%d/' % (list_.id,))
```

*lists/views.py (ch06l036-2).*

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 uncovered 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!

## 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. We're getting pretty good at these now, so let's knock one together quickly:

```
classNewItemTest(TestCase):

def test_can_save_a_POST_request_to_an_existing_list(self):
    other_list = List.objects.create()
    correct_list = List.objects.create()

    self.client.post(
        '/lists/%d/new_item' % (correct_list.id,),
        data={'item_text': 'A new item for an existing list'})
```

*lists/tests.py.*

```

self.assertEqual(Item.objects.all().count(), 1)
new_item = Item.objects.all()[0]
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/new_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)

```

That’s a little strange. We haven’t actually specified a URL for `/lists/1/new_item` yet, so our expected failure is `404 != 302`. Why are we getting a 301? It’s because we’ve used a very greedy regular expression:

```
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 URL. In this case, `/lists/1/new_item/` would be a match for `lists/(.+)/`, with the `(.+)` capturing `1/new_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`:

```
url(r'^lists/(\d+)/$', 'lists.views.view_list', name='view_list'),
```

*superlists/urls.py.*

Now we get:

```

AssertionError: 0 != 1
[...]
AssertionError: 404 != 302 : Response didn't redirect as expected: Response
code was 404 (expected 302)

```

Now we’ve got our expected 404, let’s add a new URL for adding new items to existing lists:

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

*superlists/urls.py.*

```

url(r'^lists/(\d+)/$', 'lists.views.view_list', name='view_list'),
url(r'^lists/(\d+)/new_item$', '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.

- ~~Get FTs to clean up after themselves~~
- ~~Adjust model so that items are associated with different lists~~
- ~~Add unique URLs for each list~~
- ~~Add a URL for creating a new list via POST~~
- Add URLs for adding a new item to an existing list via POST
- Refactor away some duplication in *urls.py*

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.

```

Let's try:

```

def add_item(request):
    pass

```

*lists/views.py.*

Aha:

```

TypeError: add_item() takes 1 positional argument but 2 were given

```

```

def add_item(request, list_id):
    pass

```

*lists/views.py.*

And then:

```

ValueError: The view lists.views.add_item didn't return an HttpResponse object.

```

We can copy the `redirect` from `new_list` and the `List.objects.get` from `view_list`:

```

def add_item(request, list_id):
    list_ = List.objects.get(id=list_id)
    return redirect('/lists/%d/' % (list_.id,))

```

*lists/views.py.*

That takes us to:

```

self.assertEqual(Item.objects.all().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,))
```

That's the tests passing. Now we just need to use this URL in our *list.html* template. Open it up and adjust the form tag...

```
<form method="POST" action="/lists/templates/list.html.
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. Let's create a new unit test in `ListViewT` est:

```
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:

```
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:

```
<form method="POST" action="/lists/{{ list.id }}/new_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. 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:

- ~~Adjust model so that items are associated with different lists~~
- ~~Add unique URLs for each list~~
- ~~Add a URL for creating a new list via POST~~
- ~~Add URLs for adding a new item to an existing list via POST~~
- Refactor away some duplication in *urls.py*

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 3 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).

```
urlpatterns = patterns('',
    url(r'^$', 'lists.views.home_page', name='home'),
    url(r'^lists/', include('lists.urls')),
    # url(r'^admin/', include(admin.site.urls)),
)
```

*superlists/urls.py.*

And *lists/urls.py* we can trim down to only include the latter part of our 3 URLs, and none of the other stuff from the parent *urls.py*:



```

from django.conf.urls import patterns, url

urlpatterns = patterns('',
    url(r'^(\d+)/$', 'lists.views.view_list', name='view_list'),
    url(r'^(\d+)/new_item$', 'lists.views.add_item', name='add_item'),
    url(r'^new$', 'lists.views.new_list', name='new_list'),
)

```

And re-run 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 saw how to adapt an existing site step-by-step, going from working state to working state, in order to iterate towards our new REST-ish structure. We covered some rules of thumb like “YAGNI” and “3 strikes then refactor”

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

### *Test isolation*

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.

### *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 on step at a time, and go from working state to working state.

---

# 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 ugly at the moment.

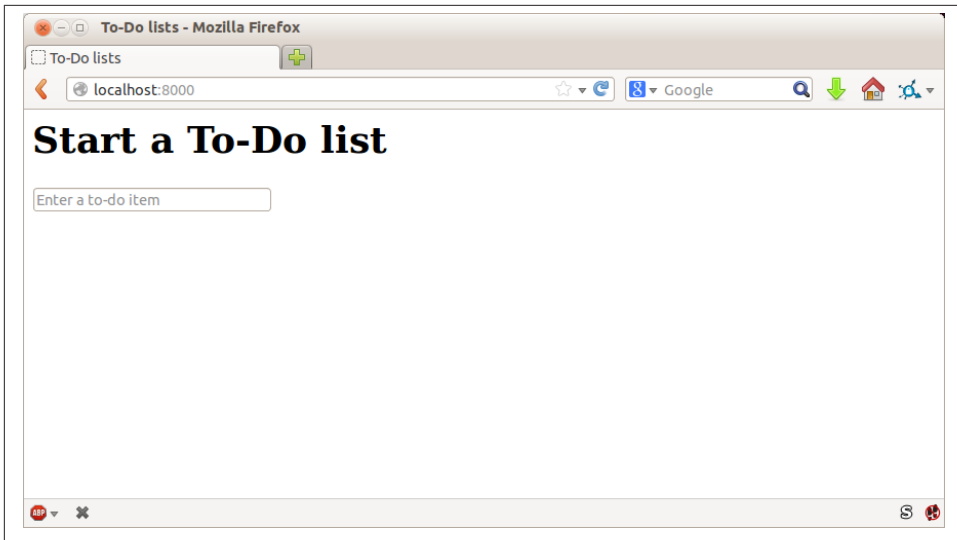


Figure 7-1. Our homepage, looking a little ugly..



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”. It’s because, in the last chapter, we added a column to an existing table: the foreign key field on the item model pointing at the list model. Syncdb doesn’t know how to modify existing tables. For now, just delete your database file (*database.sqlite*) and re-run syncdb, and everything should work again. We’ll come back to this issue in chapter 12.

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.

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 to 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:

```
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_tag_name('input')
        self.assertEqual(
            inputbox.location['x'] + inputbox.size['width'] / 2,
            512,
            delta=3
        )
```

TODO: by\_tag name can be confused if the csrf token is before the real input. change locators.

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 3 pixels.

Now, 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 103, in
test_layout_and_styling
    delta=3
AssertionError: 125.0 != 512 within 3 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 aligned-center 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_tag_name('input')  
self.assertAlmostEqual(  
    inputbox.location['x'] + inputbox.size['width'] / 2,  
    512,  
    delta=3  
)
```

That gives us another test failure:

```
File "/workspace/superlists/functional_tests/tests.py", line 113, in  
test_layout_and_styling  
    delta=3  
AssertionError: 125.0 != 512 within 3 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 un-committed 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 (let’s go shopping), 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: <sup>1</sup>

```
$ wget -O bootstrap.zip https://github.com/twbs/bootstrap/releases/download/v3.0.3/bootstrap-3.0.3
$ 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! More info in bootstrap's README



Who votes I should put an ultra-brief guide to customising bootstrap in here? Install node+npm, grunt, tweak, eg, the basic font inside the LESS, compile, and we're away? Email me via [obeythetestinggoat@gmail.com](mailto:obeythetestinggoat@gmail.com), or on the [Mailing list](#)

Our lists folder will end up looking like this:

```
lists
├── __init__.py
├── models.py
├── static
│   └── bootstrap
│       ├── css
│       │   ├── bootstrap.css
│       │   ├── bootstrap.min.css
│       │   ├── bootstrap-theme.css
│       │   └── bootstrap-theme.min.css
│       ├── fonts
│       │   ├── glyphsicons-halflings-regular.eot
│       │   ├── glyphsicons-halflings-regular.svg
│       │   ├── glyphsicons-halflings-regular.ttf
│       │   └── glyphsicons-halflings-regular.woff
│       └── js
│           ├── bootstrap.js
│           └── bootstrap.min.js
```

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.

```

├── 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 }}/new_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 %}/new_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 %}
```



```
    </table>
{% endblock %}
```

That's a refactor of the way our templates work. We re-run the FTs to make sure we haven't broken anything...

```
AssertionError: 125.0 != 512 within 3 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.*

```
<!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>
```

Finally, let's actually use some of the bootstrap magic! You'll have to read the bootstrap documentation yourself, but we can 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>
        {% block table %}
        {% endblock %}
      </div>
    </div>
  </div>
```

</div>  
</body>



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: 125.0 != 512 within 3 delta
```

Hm. no.



At this point, some Windows users have reported seeing (harmless, but distracting) error messages that say `socket.error: [Errno 10054] An existing connection was forcibly closed by the remote host`. I suspect these may have been fixed in later releases of Selenium, but please let me know if you see them too. [obeythetestinggoat@gmail.com](mailto:obeythetestinggoat@gmail.com)

## 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/1.6/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 it 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*:

```
<link href="css/bootstrap.min.css" rel="stylesheet" media="screen">
```

*lists/templates/base.html.*

To get this to work, we need to change it to

```
<link href="/static/bootstrap/css/bootstrap.min.css" rel="stylesheet" media="screen">
```

*lists/templates/base.html.*

When Django 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 then it should find it at *lists/static/bootstrap/css/bootstrap.min.css*

Hopefully it will now find the new CSS, which should get our test to pass:

```
$ python3 manage.py test functional_tests
Creating test database for alias 'default'...
..
-----
Ran 2 tests in 9.764s
```

Hooray! And, as the tests zipped past, you may have noticed our site was starting to look a little better laid-out:

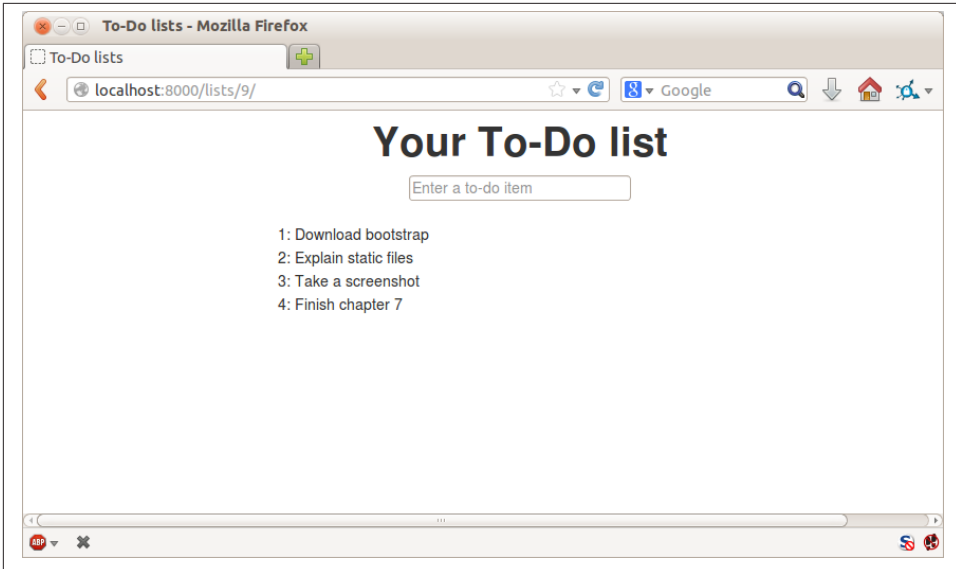


Figure 7-2. Our homepage, looking a little better...

Let's see if we can do even better. Bootstrap has a class called *jumbotron* for things that are meant to be particularly prominent on the page. Let's use that:

```
<div class="col-md-6 col-md-offset-3 jumbotron">                                lists/templates/base.html.
```

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.

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":

```
<input name="item_text" id="id_new_item" class="form-control input-lg" placeholder="Enter a to-do" lists/templates/base.html (ch07l010).
```

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:

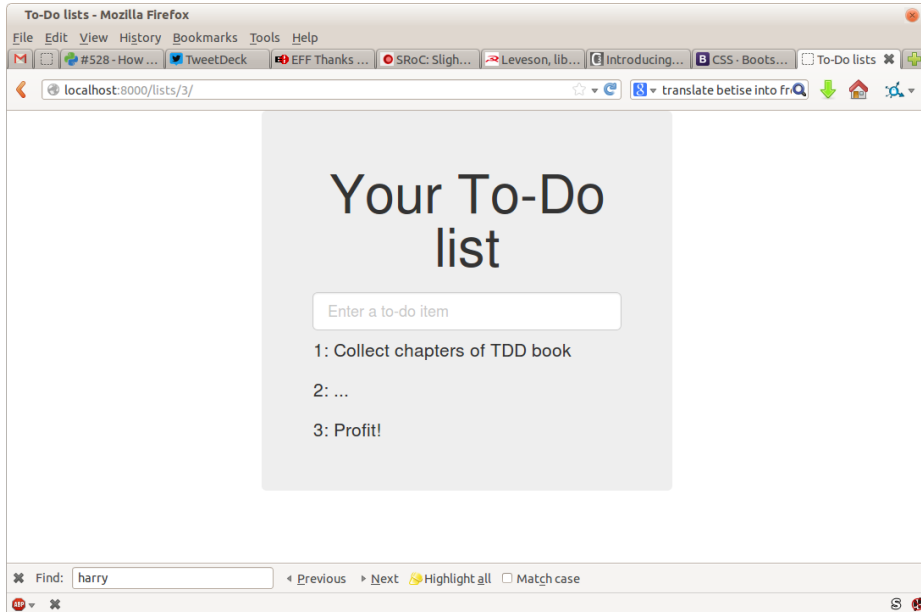
```
<head>                                                                    lists/templates/base.html.
<title>To-Do lists</title>
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<link href="/static/bootstrap/css/bootstrap.min.css" rel="stylesheet" media="screen">
<link href="/static/base.css" rel="stylesheet" media="screen">
</head>
```

And 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:

```
#id_new_item {  
    margin-top: 2ex;  
}
```

*lists/static/base.css.*

All that took me a few goes, but I'm reasonably happy with this:



If you want to go further with customising Bootstrap, you need to get into compiling LESS CSS. 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 though.

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'...
```

That's it! Definitely time for a commit:

```
$ git status # shows changes to base.html, and new folder at lists/static  
$ git add lists
```

```
$ git status # will now show all the bootstrap additions
$ git commit -m"Use Bootstrap to improve layout"
```

## What we skipped 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 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 static files are collected to, 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:

```
projects
├── superlists
│   ├── lists
│   │   └── models.py
│   ├── manage.py
│   └── superlists
└── static
    ├── base.css
    └── etc...
```

The logic is 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/1.6/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.
```

```
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.eot'
```

```
80 static files copied.
```

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, 80 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 about it later, but we're not ready to use that yet, so let's disable it for now:

```

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'
[...]

```

11 static files copied.

Much better.



are you wondering why we didn't use the functional test to test this? Unfortunately, one of the limitations of `LiveServerTestCase` is that it ignores the `STATIC_ROOT` setting, and serves static files from their app folder locations anyway. Similarly, the Django dev server (`manage.py runserver`) will serve static files from app folders when `DEBUG = True`, and not at all when `DEBUG = False`. Testing the static files setup on the production web server will be part of the next chapter.

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 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"

```

## 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.

---

# Testing deployment using a staging site

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 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 I'm going to go through and actually deploy my site to a real, live web server.

Now 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 demistify 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? It'll give me the warm and fuzzies. [obeythetestinggoat@gmail.com](mailto: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

Is all fun and game until you are need of put it in production.

— Devops Borat

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.<sup>1</sup>

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.

## Chapter overview

There’s lots of stuff in this chapter, so here’s an overview to help you keep your bearings:

1. What I’m calling a “staging” server, some people would call a “development” server, and some others would also like to distinguish “pre-production” 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.

- Adapt our FTs so they can run against a staging server.
- Spin up a server, install all the required software on it, and point our staging and live domains at it.
- Upload our code to the server using git.
- Try and get a quick & dirty version of our site running on the staging domain using the Django dev server.
- Learn how to use a virtualenv to manage our project's Python dependencies on the server.
- As we go, we'll keep running our FT, to tell us what's working and what's not.
- Move from our quick & dirty version to a production-ready configuration, using Gunicorn, upstart and domain sockets.
- 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.
- 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(LiveServerTestCase):

    @classmethod
    def setUpClass(cls): #1
        for arg in sys.argv: #2
            if 'liveserver' in arg: #3
                cls.server_url = 'http://' + arg.split('=')[1] #4
            return #5
        LiveServerTestCase.setUpClass()
        cls.server_url = cls.live_server_url

    @classmethod
    def tearDownClass(cls):
        if cls.server_url == cls.live_server_url:
            LiveServerTestCase.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, which means it only gets executed once, rather than before every test method. This is where `LiveServerTestCase` usually starts up its test server.
- ❷ ❸ We look for the *liveserver* command-line argument (which are 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`:

```
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'...
FF
```

```

=====
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 118, in
test_layout_and_styling
    delta=3
AssertionError: 0.0 != 512 within 3 delta

-----
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 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 too. 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... And 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, Rack-space, 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
- You have root access to it
- It's on the Internet,
- You can SSH into it.

I'm recommending Ubuntu as a distro, because it has Python 3.3, and it has some specific ways of configuring Nginx which I'm going to make use of below. 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.

## User accounts, SSH and privileges

In these instructions, I'm assuming that you have a non-root 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 non-root 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 `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. The indicate 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:

server command.

```
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.)

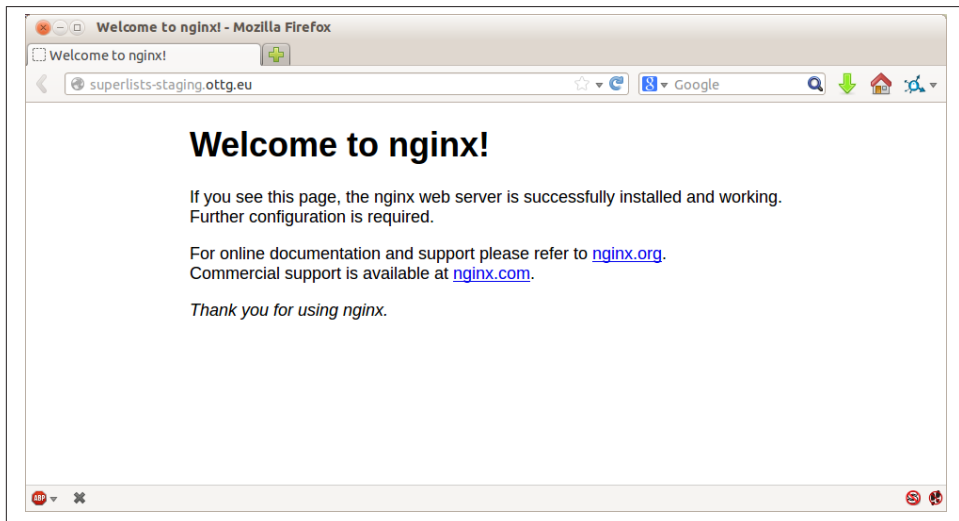


Figure 8-1. Nginx - It works!

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

server commands.

```
elspeth@server:$ sudo apt-get install git
elspeth@server:$ sudo apt-get install python3
elspeth@server:$ sudo apt-get install python3-pip
elspeth@server:$ sudo pip-3.3 install virtualenv
```

Your `pip-3.3` command may be called `pip3`, depending on what version of Ubuntu you have. Just make sure you find it, and not the Python 2 one.

## Configuring domains for staging and live

Next, 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 this:

DNS ENTRY	TYPE	PRIORITY	TTL	DESTINATION/TARGET		
*	A			81.21.76.62		
@	A			81.21.76.62		
@	MX	10		mx0.123-reg.co.uk.		
@	MX	20		mx1.123-reg.co.uk.		
dev	CNAME			harry.pythonanywhere...		
www	CNAME			harry.pythonanywhere...		
book-example	A			82.196.1.70		
book-example-staging	A			82.196.1.70		

Hostname

Type

Destination IPv4 address

Add

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...

To check this works, we can re-run our functional tests and see that their failure messages have changed slightly

```
$ python3 manage.py test functional_tests --liveserver=superlists-staging.ottg.eu
[...]
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"tag name","selector":"input"}' ; 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 do some of what you'd call "deployment", as well as provisioning, so we should be thinking about how we can automate the process, as we go.



One way of telling the difference between provisioning and 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 non-root user, in my case it would be at `/home/harry` (this is likely to be the setup on any shared hosting system, but you should always run your web apps as a non-root user, in any case). I'm going to set up my sites like this:

```
/home/harry
├── 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.



Do you need help creating a non-root user? Try: `useradd -m my-username` and then `passwd my-username`

## 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 confusions about the current working directory:

```
# 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 syncdb --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:

server commands.

```
elspeth@server:$ export SITENAME=superlists-staging.ottg.eu
elspeth@server:$ mkdir -p ~/sites/$SITENAME
elspeth@server:$ mkdir ~/sites/$SITENAME/database
elspeth@server:$ mkdir ~/sites/$SITENAME/static
elspeth@server:$ mkdir ~/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
```



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 re-define 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:

server commands.

```
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:

```
$ pip-3.3 install virtualenv
```

We'll follow the same folder structure as we're planning for the server:

```
$ virtualenv --python=python3.3 ../virtualenv
$ ls ../virtualenv/
bin  include  lib
```



this folder structure will be slightly different on Windows, eg bin=Scripts. Let me know if it's impossible to follow the instructions as a result.

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:

```
$ source ../virtualenv/bin/activate
(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 `virtualenv`s on your own PC.

That will show an `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
[...]
Successfully installed django
Cleaning up...
(virtualenv)$ python3 manage.py test lists
[...]
OK
$ ls ../virtualenv/lib/python3.3/site-packages/
django                pip                    setuptools
Django-1.6-py3.3.egg-info  pip-1.4.1-py3.3.egg-info  setuptools-0.9.8-py3.3.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==1.6
$ git add requirements.txt
$ git commit -m"Add requirements.txt for virtualenv"
```

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:

server commands.

```
elspeth@server:$ git pull
elspeth@server:$ virtualenv --python=python3.3 ../virtualenv/
elspeth@server:$ source ../virtualenv/bin/activate
(virtualenv)$ pip install -r requirements.txt
Downloading/unpacking Django==1.6 (from -r requirements.txt (line 1))
[...]
Successfully installed Django
Cleaning up...
(virtualenv)$ python3 manage.py runserver
Validating models...
0 errors found
[...]
```

That looks like it worked.

## Simple nginx configuration

Let's now go and 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 {
    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 <sup>2</sup> 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:

server command.

```
elspeth@server:$ sudo ln -s ../sites-available/$SITENAME /etc/nginx/sites-enabled/$SITENAME
```

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`.

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.



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.

We also may as well remove the default “Welcome to nginx” config, to avoid any confusion:

server command.

```
elspeth@server:$ sudo rm /etc/nginx/sites-enabled/default
elspeth@server:$ sudo reboot
```

(The reboot is there to avoid a strange issue I came across whereby nginx would keep serving the default page on the first hit. There always seems to be some voodoo in server config!)

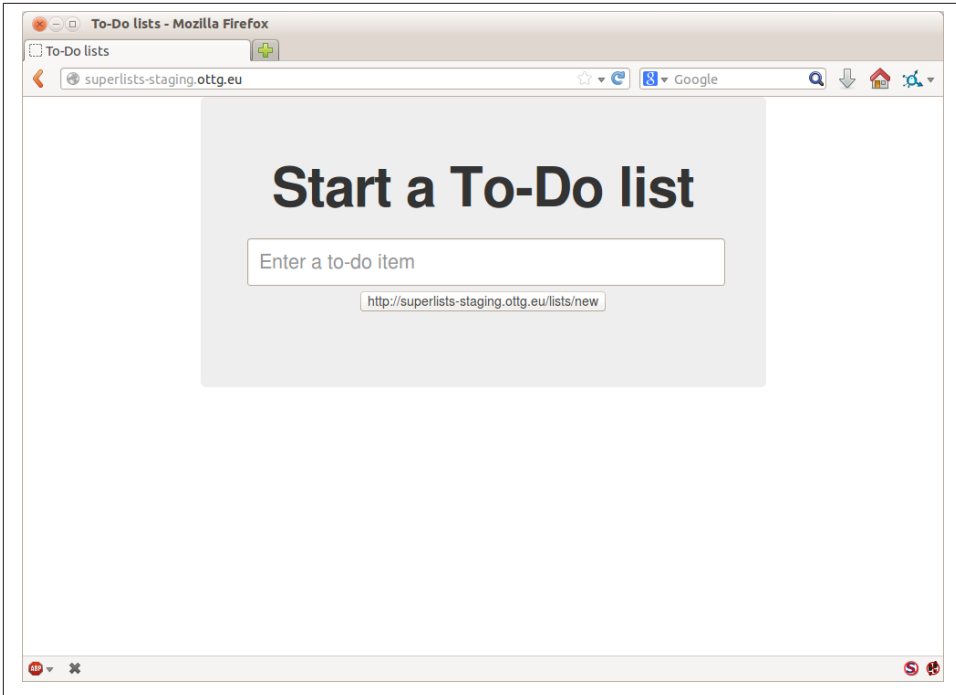
And now to test it:

server commands.

```
elspeth@server:$ sudo service nginx reload
elspeth@server:$ source ../virtualenv/bin/activate
(virtualenv)$ python3 manage.py runserver
```

A quick visual inspection confirms — the site is up!





*Figure 8-3. 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 telling us as much as the tests went through, or if you tried it manually.

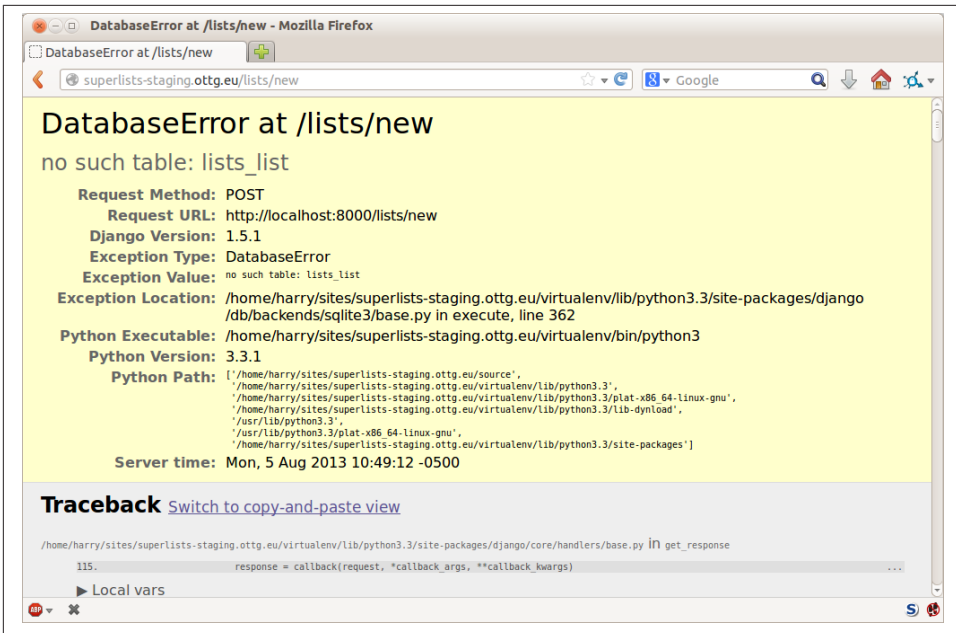


Figure 8-4. But the database isn't

Let's set up the database then.

## Creating the database with syncdb

We run syncdb using the `--noinput` argument to suppress the two little “are you sure” prompts. Press Ctrl+C to interrupt the current runserver.

server commands.

```
(virtualenv)$ python3 manage.py syncdb --noinput
Creating tables ...
[...]
(virtualenv)$ ls ../database/
db.sqlite3
(virtualenv)$ 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'...
```



if you see a “502 - Bad Gateway”, it’s probably because you forgot to restart the dev server with `manage.py runserver` after the `syncdb`

Progress! We’re at least reassured that some of the 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 — that: “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...

server command.

```
(virtualenv)$ 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`.

We can try that out, and check that all the `virtualenv` magic works too, by *deactivating* the `virtualenv` and seeing if we can *still* serve our app using the `gunicorn` executable that `pip` just put in there for us:

server commands.

```
(virtualenv)$ which gunicorn
/home/harry/sites/superlists-staging.ottg.eu/virtualenv/bin/gunicorn
(virtualenv)$ deactivate
$ ../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:

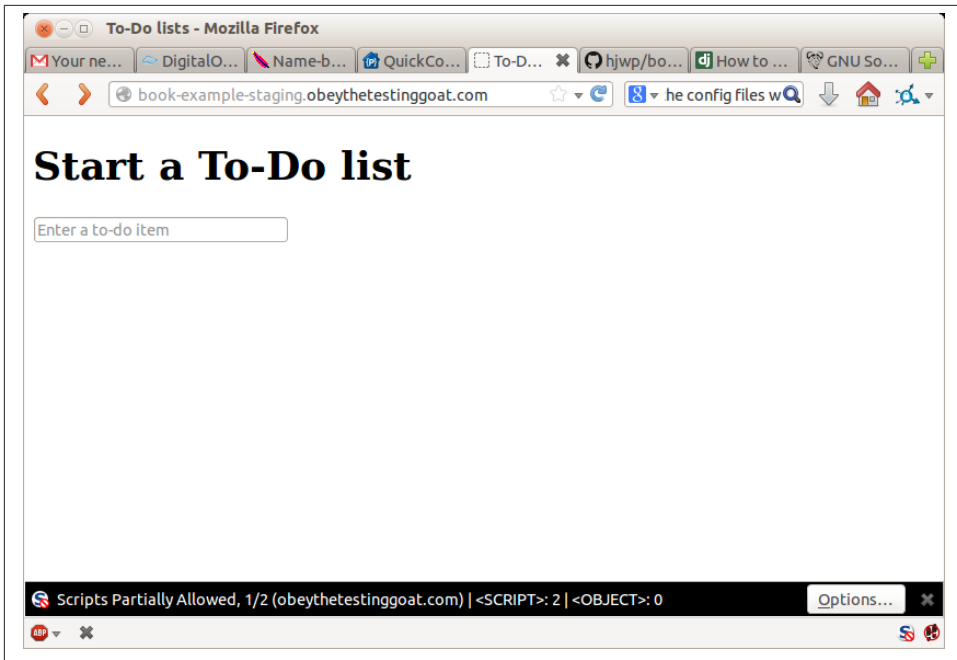


Figure 8-5. Broken CSS

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 != 497 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.

## Getting Nginx to serve static files

First we run `collectstatic` to copy all the static files to a folder where Nginx can find them:

server commands.

```
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 {
    listen 80;
    server_name superlists-staging.ottg.eu;

    location /static {
        alias /home/harry/sites/superlists-staging.ottg.eu/static;
    }

    location / {
        proxy_pass http://localhost:8000;
    }
}
```

Reload nginx and restart gunicorn...

server commands.

```
$ sudo service nginx reload
$ ../virtualenv/bin/gunicorn superlists.wsgi:application
```

And if we take another look at the site, things are looking much healthier. We can re-run 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:

```
[...]
location / {
    proxy_set_header Host $host;
```

```

    proxy_pass http://unix:/tmp/superlists-staging.ottg.eu.socket;
}
}

```

proxy\_set\_header is 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:

server commands.

```

$ sudo service nginx reload
$ ../virtualenv/bin/gunicorn --bind \
    unix:/tmp/superlists-staging.ottg.eu.socket superlists.wsgi:application

```

And again, we re-run 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:

```

# 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
stop on shutdown

respawn

chdir /home/harry/sites/superlists-staging.ottg.eu/source
exec ../virtualenv/bin/gunicorn \
    --bind unix:/tmp/superlists-staging.ottg.eu.socket \
    superlists.wsgi:application
```

Upstart is joyously simple to configure (especially if you've ever had the pleasure of writing an `init.d` script), and is fairly self-explanatory. The `start on net-device-up` makes sure Gunicorn only runs once the server has connected up to the internet. `respawn` will restart Gunicorn automatically if it crashes, and `chdir` sets the working directory

Upstart scripts live in `/etc/init`, and their names must end in `.conf`.

Now we can start gunicorn with

server commands.

```
sudo start gunicorn-superlists-staging.ottg.eu
```

And we can re-run 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
(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 re-cap on our provisioning and deployment procedures

Provisioning:

- assume we have a user account & home folder
- apt-get nginx git python-pip
- pip install virtualenv
- add nginx config for virtual host
- add upstart job for gunicorn

Deployment

- create directory structure in `~/sites`
- pull down source code into folder named source
- start virtualenv in `../virtualenv`
- pip install -r requirements.txt
- syncdb for database
- collectstatic for static files
- set `DEBUG = False` and `ALLOWED_HOSTS` in settings.py
- restart gunicorn job
- 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 re-use them later. Let's save them in a new subfolder in our repo:

```
$ mkdir deploy_tools
```

```
server {
    listen 80;
    server_name SITENAME;

    location /static {
        alias /home/harry/sites/SITENAME/static;
    }

    location / {
        proxy_set_header Host $host;
        proxy_pass http://unix:/tmp/SITENAME.socket;
    }
}
```

*deploy\_tools/nginx.template.conf.*



```

    }
}

description "Gunicorn server for SITENAME"
start on net-device-up
stop on shutdown

respawn

chdir /home/harry/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 pip-3.3 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
│   ├── unicorn-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
    └─ [...]
```

## Automating deployment with fabric

Fabric is a tool which lets you automate commands that you want to run on servers. You can install fabric system-wide — 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:

```
$ pip-2.7 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 has provided some windows binaries:

<http://www.voidspace.org.uk/python/modules.shtml#pycrypto>

(Don't forget to giggle at the mention of absurd US munitions export controls.)

So the instructions, for Windows, are: - download and install pycrypto from the url above - then pip install fabric.

Another amazing source of precompiled Python packages for Windows is maintained by Christoph Gohlke at:

<http://www.lfd.uci.edu/~gohlke/pythonlibs/>

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*

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.

```
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
SITES_FOLDER = '/home/harry/sites'

def deploy():
    _create_directory_structure_if_necessary(env.host) #❷
```

*deploy\_tools/fabfile.py.*

```

source_folder = '%s/%s/source' % (SITES_FOLDER, env.host)
_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*.

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:

```

def _create_directory_structure_if_necessary(site_name):
    for subfolder in ('database', 'static', 'virtualenv', 'source'):
        run('mkdir -p %s/%s/%s' % (SITES_FOLDER, site_name, 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 than `mkdir` 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.<sup>3</sup>

Next we want to pull down our source code:

```

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

```

3. 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

- 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.<sup>4</sup>
- 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.
- 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, the current commit that's in your local working tree needs to be up on the VCS code sharing site, so that the server can pull it down and reset to it. So 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:

```

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') #❹

```

*deploy\_tools/fabfile.py.*

4. there is a fabric “`cd`” command, but I figured it was one thing too many to add in this chapter

- 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 just `ALLOWED_HOSTS`, using a regex to match whether if the line is already there).
- 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). More info 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).



Other people 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=python3.3 %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 syncdb:

```
def _update_database(source_folder):  
    run('cd %s && ../virtualenv/bin/python3 manage.py syncdb --noinput' % (  
        source_folder,  
    ))
```

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=harry@superlists-staging.ottg.eu
```

```
[superlists-staging.ottg.eu] Executing task 'deploy'  
[superlists-staging.ottg.eu] run: mkdir -p /home/harry/sites/superlists-staging.ottg.eu  
[superlists-staging.ottg.eu] run: mkdir -p /home/harry/sites/superlists-staging.ottg.eu/database  
[superlists-staging.ottg.eu] run: mkdir -p /home/harry/sites/superlists-staging.ottg.eu/static  
[superlists-staging.ottg.eu] run: mkdir -p /home/harry/sites/superlists-staging.ottg.eu/virtualenv  
[superlists-staging.ottg.eu] run: mkdir -p /home/harry/sites/superlists-staging.ottg.eu/source  
[superlists-staging.ottg.eu] run: cd /home/harry/sites/superlists-staging.ottg.eu/source && git fe  
[localhost] local: git log -n 1 --format=%H  
[superlists-staging.ottg.eu] run: cd /home/harry/sites/superlists-staging.ottg.eu/source && git re  
[superlists-staging.ottg.eu] out: HEAD is now at 85a6c87 Add a fabfile for automated deploys  
[superlists-staging.ottg.eu] out:  
  
[superlists-staging.ottg.eu] run: sed -i.bak -r -e 's/DEBUG = True/DEBUG = False/g' "$(echo /home/  
[superlists-staging.ottg.eu] run: echo 'ALLOWED_HOSTS = ["superlists-staging.ottg.eu"]' >> "$(echo  
[superlists-staging.ottg.eu] run: echo 'SECRET_KEY = '\\\\'4p2u8fi6)bltep(6nd_3tt9r41skhr%ttyjatf4+  
[superlists-staging.ottg.eu] run: echo 'from .secret_key import SECRET_KEY' >> "$(echo /home/harry  
  
[superlists-staging.ottg.eu] run: /home/harry/sites/superlists-staging.ottg.eu/source/../virtualen  
[superlists-staging.ottg.eu] out: Requirement already satisfied (use --upgrade to upgrade): Django  
[superlists-staging.ottg.eu] out: Requirement already satisfied (use --upgrade to upgrade): gunico  
[superlists-staging.ottg.eu] out: Cleaning up...  
[superlists-staging.ottg.eu] out:  
  
[superlists-staging.ottg.eu] run: cd /home/harry/sites/superlists-staging.ottg.eu/source && ../vir  
[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/harry/sites/superlists-staging.ottg.eu/source && ../vir  
[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.
```





```

[superlists.ottg.eu] out:   Downloading Django-1.6.tar.gz (8.0MB):
[...]
[superlists.ottg.eu] out:   Downloading Django-1.6.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 '__pycache__' found u
[superlists.ottg.eu] out:     warning: no previously-included files matching '*.py[co]' found unde
[superlists.ottg.eu] out:   Downloading/unpacking gunicorn==17.5 (from -r /home/harry/sites/superlists
[superlists.ottg.eu] out:   Downloading gunicorn-17.5.tar.gz (367kB): 100% 367kB
[...]
[superlists.ottg.eu] out:   Downloading gunicorn-17.5.tar.gz (367kB): 367kB downloaded
[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 from 664 to 775
[superlists.ottg.eu] out:
[superlists.ottg.eu] out:     warning: no previously-included files matching '__pycache__' found u
[superlists.ottg.eu] out:     warning: no previously-included files matching '*.py[co]' found unde
[superlists.ottg.eu] out:     changing mode of /home/harry/sites/superlists.ottg.eu/virtualenv/bin
[superlists.ottg.eu] out:   Running setup.py install for gunicorn
[superlists.ottg.eu] out:
[superlists.ottg.eu] out:     Installing gunicorn_paster script to /home/harry/sites/superlists.ot
[superlists.ottg.eu] out:     Installing gunicorn script to /home/harry/sites/superlists.ottg.eu/v
[superlists.ottg.eu] out:     Installing gunicorn_django script to /home/harry/sites/superlists.ot
[superlists.ottg.eu] out:   Successfully installed Django gunicorn
[superlists.ottg.eu] out:   Cleaning up...
[superlists.ottg.eu] out:

[superlists.ottg.eu] run: cd /home/harry/sites/superlists.ottg.eu/source && ../virtualenv/bin/pyth
[superlists.ottg.eu] out:   Copying '/home/harry/sites/superlists.ottg.eu/source/lists/static/base.c
[superlists.ottg.eu] out:   Copying '/home/harry/sites/superlists.ottg.eu/source/lists/static/bootst
[...]
[superlists.ottg.eu] out:   Copying '/home/harry/sites/superlists.ottg.eu/source/lists/static/bootst
[superlists.ottg.eu] out:
[superlists.ottg.eu] out:   11 static files copied.
[superlists.ottg.eu] out:

[superlists.ottg.eu] run: cd /home/harry/sites/superlists.ottg.eu/source && ../virtualenv/bin/pyth
[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 the `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 `syncdb` seems to have worked too.

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?

server command.

```
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 virtual-host config file.

We can now activate that file:

server command.

```
elspeth@server:$ sudo ln -s ../sites-available/superlists.ottg.eu \
/etc/nginx/sites-enabled/superlists.ottg.eu
```

Now we write the upstart script:

server command.

```
elspeth@server:$ sed "s/SITENAME/superlists.ottg.eu/g" deploy_tools/gunicorn-upstart.template.conf
sudo tee /etc/init/gunicorn-superlists.ottg.eu.conf
```

And now we start both services:

server commands.

```
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
$ 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...

## Recap:

Lots of this, particularly on the provisioning side, was very specific to the setup I happened to use. When you deploy sites, you might use Apache instead of nginx, uWSGI instead of Gunicorn, Supervisor instead of Upstart, and so on. If you use a PaaS, some of these problems will be solved for you, others won't. But I really wanted to take you through a practical example, so we could see some of the concerns involved in deployment.

There are some elements that will be common to almost all situations though:

- You need to choose a place for your static files
- You'll need specific config for your database
- You need to run some kind of webserver, set it to listen on some port

On the deployment side, you should find that much of what we've done is transferable to any situation:

- During a deploy, you need a way to *update your source code*. We're using `git pull`.
- You need a way to update your *static files* (`collectstatic`)
- You need to update your *database* (`syncdb` for now, we'll look at South and schema migrations later)
- You need to manage your dependencies, and make sure any packages you need are available on the server. We use a *virtualenv* to isolate our various sites from each other.

- You'll probably need to tweak some items in *settings.py* when switching to production.
- You'll want to *test* that these things work, by doing your deployment to a staging site first
- You should be able to run your functional test suite against the *staging site*.
- You'll want to *automate* all of the steps involved in a deploy, to give yourself confidence that when you deploy to live, things will go just as smoothly as when you deployed to staging.

## Further reading:

I'm no grizzled expert on deployment. I've tried to set you off on a reasonably sane path, but there's lots of things you could do differently, and lots, lots more to learn besides. Here are some articles I used for inspiration:

- Solid Python Deployments for Everybody
- Git-based fabric deployments are awesome

For some ideas on how you might go about automating the provisioning step, and an alternative to Fabric, go check out [Appendix III](#).

## Todos

There's no such thing as the One True Way in deployment, and as I say I'm no kind of expert. I hope that this chapter will change and improve over the coming months, especially thanks to the feedback of beloved readers!

Here's a few things I'm thinking of changing or adding. Let me know what you think, and what else should be on the list! [obeythetestinggoat@gmail.com](mailto:obeythetestinggoat@gmail.com)

Chapter Objectives:

- As simple as possible
- But no simpler
- Illustrate some (the main?) challenges of deployment
- Show where TDD fits in
- Try and make it reminiscent of the environment you'd get in a PaaS

Possible changes:

- setup logging... but how best? and where to put log files?

- `setuid` in upstart script
- using `/home/username` to make it “like shared hosting” — is that totally outdated? Should I just put everything in `/var/www`?

---

# 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.



I had originally intended to spend just one chapter talking about forms and validation. It's ended up being three. I'd really like your feedback — is it too much? What, if anything, would you drop?

## 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't get our site to help!

Let's see how that might work as an FT:

```
def test_cannot_add_empty_list_items(self):  
    functional_tests/tests.py (ch09l001).  
    # 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

First off, 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`:

```
from unittest import skip
[...]
```

*functional\_tests/tests.py (ch09l001-1).*

```
@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 re-run 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.

It's definitely true that 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, ideas for how to refactor code will occur to you a few days after you've written a piece of code — perhaps weeks or months later, as you're writing some new functionality, when you spot some old code with fresh eyes. But if you're half-way through something else, should you stop to refactor the old code?

The answer is: 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 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, and wait until we're back to a fully passing test suite before refactoring.

## Splitting functional tests out into many files



this chapter depends on Django 1.6. If you started with an old version of the book that used Django 1.5, upgrade now with a `pip install --upgrade django`

We start putting each test into its own class, still in the same file:

```
class FunctionalTest(LiveServerTestCase): functional_tests/tests.py (ch09l002).  
  
    @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 re-run 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 4 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 re-use it in our new FT.

```
from django.test import LiveServerTestCase
from selenium import webdriver
import sys

class FunctionalTest(LiveServerTestCase):

    @classmethod
    def setUpClass(cls):
        [...]
    def tearDownClass(cls):
        [...]
```

*functional\_tests/base.py (ch09l003).*

```
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 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 (ch09l004).
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 those a lot more often 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 source. That applies in this case because I know for sure all the test 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 (ch09l005).
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 (ch09l006).
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 re-running `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 (ch09l007).
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 (ch09l008).
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 called `.has-error` to mark our error text. We'll see that Bootstrap has some useful styling for those
- ❷ As predicted, we are re-using 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 absolutely vital to preventing 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:

```

## 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 test, 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 a dunderinit<sup>1</sup> — it needs to be there or else the tests folder isn’t a valid Python module...

Now we turn `test_all` 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:

```
from django.test import TestCase                                     lists/tests/test_models.py (ch09l009).

from lists.models import Item, List
```

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

Whereas `test_views.py` just loses one class:

```
--- a/lists/tests/test_views.py                                     lists/tests/test_views.py (ch09l010).
+++ b/lists/tests/test_views.py
@@ -103,34 +103,3 @@ 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):
- [...]
```

And we re-run the tests to check everything is still there:

```
$ python3 manage.py test lists
[...]
Ran 10 tests in 0.040s

OK
```

1. “dunder” is shorthand for double-underscore, so “dunderinit” means `__init__.py`

Great!

```
$ git add lists/tests
$ git commit -m "Split out unit tests into two files"
```



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:

```
from django.core.exceptions import ValidationError
class ListAndItemModelsTest(TestCase):
    [...]

    def test_cannot_save_empty_list_items(self):
        list1 = List.objects.create()
        item = Item(list=list1, 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 set-up, clean-up, 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 full_clean should have raised an exception')
except ValidationError:
    pass
```

But the `with` formulation is neater. Now, we can try running the test, and see if fail:

```
item.save()
AssertionError: ValidationError not raised
```

## Overriding the save method on a model to ensure validation

And now we discover one of Django's dirty little secrets. *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 tedious 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.

Django does have a method to manually run full validation however, called `full_clean`. You can hack it in to see it work if you like:

```
with self.assertRaises(ValidationError):  
    item.save()  
    item.full_clean()
```

*lists/tests/test\_models.py.*

Which would get the tests to pass. Let's revert it and make a real implementation by overriding the model's save method:

```
class Item(models.Model):  
    text = models.TextField()  
    list = models.ForeignKey(List)  
  
    def save(self, *args, **kwargs):  
        self.full_clean()  
        super().save(*args, **kwargs)
```

*lists/models.py (ch09l013).*



It's good practice to use `*args`, `**kwargs` when overriding Django model methods like `save`, because they're called from all sorts of strange places, and you want to make sure those arguments get passed to the superclass `save`, so that all the Django magic still works.

That works:

```
$ python3 manage.py test lists  
Creating test database for alias 'default'...  
.....  
-----  
Ran 11 tests in 0.037s  
  
OK  
Destroying test database for alias 'default'...
```

TODO: putting `full_clean` into the model save method may not be a good idea in general. investigate putting it in the view instead.

## Handling model validation errors in the view:

Next we want to surface those validation errors from the model into a useful form for the user. This is the job of the view and template. We start by adjusting our 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 (ch09l014).  
  
class NewListTest(TestCase):  
    ...  
  
    def test_validation_errors_sent_back_to_home_page_template(self):  
        response = self.client.post('/lists/new', data={'item_text': ''})  
        self.assertEqual(Item.objects.all().count(), 0)  
        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 hard-coded 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 hard-coded 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.

As it is, the test fails out with an error — our view tries to save an item with blank text, but the model validation raises an exception:

```
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 by just the try/except and nothing else. The tests should tell us what to code next...

```
lists/views.py (ch09l015).  
  
from django.core.exceptions import ValidationError  
...  
  
def new_list(request):  
    list_ = List.objects.create()  
    try:  
        Item.objects.create(text=request.POST['item_text'], list=list_)  
    except ValidationError:  
        pass  
    return redirect('/lists/%d/' % (list_.id,))
```

As we're looking at the view code, we make a note that there's a hard-coded URL in there. Let's add that to our scratchpad:



- remove hard-coded URLs from *views.py*

Back to the test, which wants us to use a template:

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

We try that naively:

```
except ValidationError:
    return render(request, 'home.html')
```

*lists/views.py (ch09l016).*

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:

```
except ValidationError:
    error_text = "You can't have an empty list item"
    return render(request, 'home.html', {"error": error_text})
```

*lists/views.py (ch09l017).*

And adjusting the template HTML itself — it's not actually in *home.html*, it's in the parent template. But while we're looking around for it, we notice another hard-coded URL:

```
{% block form_action %}/lists/new{% endblock %}
```

We'll make a note to fix that, and then go up to *base.html*:

- remove hard-coded URLs from *views.py*
- remove hard-coded URLs from form in *home.html*

Here's what we actually want to change now:

```
<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
  {% csrf_token %}
  {% if error %}
    <div class="form-group has-error">
      <span class="help-block">{{ error }}</span>
    </div>
  {% endif %}
</form>
```

*lists/templates/base.html (ch09l018).*

Take a look at the [Bootstrap docs](#) for more info on form controls.

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 in to 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 (ch09l019).*

```
from django.utils.html import escape
[...]

expected_error = escape("You can't have an empty list item")
self.assertContains(response, expected_error)
```

That passes! 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 24, in test_cannot_add_empty_list_items
[...]
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"id","selector":"id_list_table"}' ; Stacktrace:
```

Not quite, but they did get a little further. Checking the line 24, 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 raises an exception. That's currently producing a server error instead of a nice exception, so let's fix that.

But first, a little commit:

```
$ git commit -am"Adjust new list view to render validation errors"
```

## Django pattern: processing POST request 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:

```
{% block form_action %}/lists/{{ list.id }}/{% endblock %}
```

*lists/templates/list.html (ch09l020).*

Incidentally, that's another item in our to-do list:

- remove hard-coded URLs from *views.py*
- remove hard-coded URLs from form in *home.html*
- remove hard-coded URLs from form in *list.html*

This will immediately break our original functional test:

```
$ python3 manage.py test functional_tests.test_simple_list_creation
AssertionError: '2: Use peacock feathers to make a fly' not found in ['1: Buy
peacock feathers']
```

Now let's change the tests for saving POST requests to existing lists. We move them both into *ListViewTest*, and make them point at the base list URL:

```
lists/tests/test_views.py (ch09l021).
```

```
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.all().count(), 1)
        new_item = Item.objects.all()[0]
        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. 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, and delete the `add_item` view:

```

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_})

```

*lists/views.py (ch09l022).*

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.
[...]
django.core.exceptions.ViewDoesNotExist: Could not import lists.views.add_item.
View does not exist in module lists.views.

```

It's because we've deleted the view, but it's still being referred to in `urls.py`. We remove it from there:

```

urlpatterns = patterns('',
    url(r'^(\d+)/$', 'lists.views.view_list', name='view_list'),
    url(r'^new$', 'lists.views.new_list', name='new_list'),
)

```

*lists/urls.py (ch09l023).*

And that gets us to the OK. Let's try a full FT run, to make sure our refactor is complete:

```

$ python3 manage.py test functional_tests
[...]

```

```
Ran 3 tests in 15.276s
```

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

We're back to the 1 failure in our new functional test. We should commit there.

```
$ git commit -am"Refactor list view to handle new item POSTs"
```



Am I breaking 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 it's OK for the FT for the current story you're working to be failing.

Next we write a new unit test for the validation of items posted to the *existing* lists view. It's very similar to the one for the home page, just a couple of tweaks:

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

    def test_validation_errors_end_up_on_lists_page(self):
        listey = List.objects.create()

        response = self.client.post(
            '/lists/%d/' % (listey.id,),
            data={'item_text': ''}
        )
        self.assertEqual(Item.objects.all().count(), 0)
        self.assertTemplateUsed(response, 'list.html')
        expected_error = escape("You can't have an empty list item")
        self.assertContains(response, expected_error)
```

*lists/tests/test\_views.py (ch09l024).*

Which should fail, because our view currently doesn't catch validation errors from the save.

```
django.core.exceptions.ValidationError: {'text': ['This field cannot be blank.']}
```

Here's an implementation:

```
def view_list(request, list_id):
    list_ = List.objects.get(id=list_id)
    error = None

    if request.method == 'POST':
        try:
            Item.objects.create(text=request.POST['item_text'], list=list_)
            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})
```

*lists/views.py (ch09l025).*

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.

Let's wait a bit before we do a refactor 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:

- remove hard-coded URLs from *views.py*
- remove hard-coded URLs from form in *home.html*
- remove hard-coded URLs from form in *list.html*
- remove duplication of validation logic in views.



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...

And that gets us to the end of the test!

OK

Fantastic. We're back to a working state, so we can take a look at some of the items on our scratchpad. But I'd say it is *definitely* time for a tea break first.

## Refactor: Removing hard-coded 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!

```
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 hard-coded URL in *home.html* with a Django template tag which refers to the URL's “name”:

```
{% block form_action %}{% url 'new_list' %}{% endblock %}
```

*lists/templates/home.html (ch09l026-1).*

We check that doesn't break the unit tests:

```
$ python3 manage.py test lists
OK
```

And we check the functional tests too:

```
$ python3 manage.py test functional_tests
OK
```

Fabulous, that's one refactor done:

- remove hard-coded URLs from *views.py*
- ~~remove hard-coded URLs from form in *home.html*~~
- remove hard-coded URLs from form in *list.html*
- remove duplication of validation logic in views.

Let's do the other template while we're at it. This one is more interesting, because we pass it a parameter:

```
{% block form_action %}{% url 'view_list' list.id %}{% endblock %}
```

*lists/templates/list.html (ch09l026-2).*

Check out the [Django docs on reverse url resolution](#) for more info.

We run the tests again, and check they all pass (or get to the expected self.fail):

```
$ python3 manage.py test lists
OK
$ python3 manage.py test functional_tests
OK
```

That's worthy of a commit:

```
$ 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:

```
def new_list(request):
    [...]
    return redirect('view_list', list_.id)
```

*lists/views.py (ch09l026-3).*

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 view (which we'll see later in the book): 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 (ch09l026-4).*

```
def test_get_absolute_url(self):
    list1 = List.objects.create()
    self.assertEqual(list1.get_absolute_url(), '/lists/%d/' % (list1.id,))
```

Which gives

```
AttributeError: 'List' object has no attribute 'get_absolute_url'
```

And the implementation use one of Django's shortcut functions, so again we obey DRY and avoid using a hard-coded string:

```
from django.shortcuts import resolve_url
```

*lists/models.py (ch09l026-5).*

```
class List(models.Model):

    def get_absolute_url(self):
        return resolve_url('view_list', 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 automatically!

```
def new_list(request):
    [...]
    return redirect(list_)
```

*lists/views.py (ch09l026-6).*

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`:

```
def view_list(request, list_id):
    [...]

    try:
        Item.objects.create(text=request.POST['item_text'], list=list_)
        return redirect(list_)
    except ValidationError:
        error = "You can't have an empty list item"
```

*lists/views.py (ch09l026-7).*

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 todos:

- ~~remove hard-coded URLs from *views.py*~~
- ~~remove hard-coded URLs from form in *home.html*~~
- ~~remove hard-coded URLs from form in *list.html*~~



- remove duplication of validation logic in views.

And 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 test, 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*, *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 10

# 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 improve our view unit tests. Currently, many of them test several things at once. We'll move them towards best practice, where each has a single assertion.

## 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 powers in Django:

- They can process user input and validate it for errors.
- They can be rendered 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.

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 forms 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 explanatory coding. You could just as easily use a `manage.py shell` session, although you'd need to keep reloading your code for each change.

For now it will just fail with an import error.

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:

```
widget=forms.fields.TextInput(attrs={
    'placeholder': 'Enter a to-do item',
    'class': 'form-control input-lg',
}),
```

*lists/forms.py.*



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-floppy-forms](#) 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 re-write 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 re-use 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`:

```
from django import forms

from lists.models import Item

class ItemForm(forms.models.ModelForm):
```

*lists/forms.py.*

```
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:

```
def test_form_validation_for_blank_items(self):
    form = ItemForm(data={'text': ''})
    form.save()
```

*lists/tests/test\_forms.py (ch09l046).*

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`:

```
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"]  
    )
```

*lists/tests/test\_forms.py (ch09l047).*

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 which 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 does involve hacking the form's `init` though:

```
from django import forms  
  
from lists.models import Item  
  
class ItemForm(forms.models.ModelForm):  
  
    class Meta:  
        [...]  
  
    def __init__(self, *args, **kwargs):  
        super().__init__(*args, **kwargs)  
        empty_error = "You can't have an empty list item"  
        self.fields['text'].error_messages['required'] = empty_error
```

*lists/forms.py (ch10l010).*



Django 1.6 has a simpler way of overriding field error messages. I haven't had time to implement it yet, but you should feel free to look it up and use it!

You know what would be even better than messing about with all these error strings? Having a constant:

*lists/forms.py (ch10l011).*

```
EMPTY_LIST_ERROR = "You can't have an empty list item"
```

```
class ItemForm(forms.models.ModelForm):  
  
    class Meta:  
        [...]  
  
    def __init__(self, *args, **kwargs):  
        super().__init__(*args, **kwargs)  
        self.fields['text'].error_messages['required'] = EMPTY_LIST_ERROR  
  
    [...]
```

Re-run the tests to see they pass.... OK. Now we change the test:

```
from lists.forms import EMPTY_LIST_ERROR, ItemForm lists/tests/test_forms.py (ch09l050).  
[...]  
  
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. 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` with a set of tests that use the Django Test Client. We leave the old test in at first, to check that our new tests are equivalent:

TODO: say we're getting rid of `test_root_url_resolves_to_home_page_view` too

*lists/tests/test\_views.py (ch10l013).*

```
from lists.forms import ItemForm
[...]
```

```
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 form

That gives us:

```
KeyError: 'form'
```

So we use the form in our home page view:

*lists/views.py (ch10l014).*

```
[...]
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 (ch10l015).*

```
<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: '<!DOCTYPE html>\n<html lang="en">\n    <head>\n'
[...]
```

That error message is impossible to read though. Let's clarify its message a little:

```
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:

```
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 old test. The `assertTemplateUsed` and `response.context` checks from the new test are sufficient for testing a basic view with a GET request.

## A big find & 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 & 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*
```

That's a good call for a refactor. Let's make a new helper method in *base.py*:

```
class FunctionalTest(LiveServerTestCase):
    [...]
    def get_item_input_box(self):
        return self.browser.find_element_by_id('id_text')
```

*functional\_tests/base.py (ch09l057).*

And then we use it throughout - I had to make 3 changes in *test\_simple\_list\_creation.py*, 2 in *test\_layout\_and\_styling.py* and 4 in *test\_list\_item\_validation.py*, eg:

```
# She is invited to enter a to-do item straight away
inputbox = self.get_item_input_box()
```

Or

```
# 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 re-do 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.objects.create(text=request.POST['item_text'],
list=list_)
lists/views.py:        Item.objects.create(text=request.POST['item_text'],
list=list_)
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 re-run the unit tests to check everything still works:

```
$ python3 manage.py test lists
Creating test database for alias 'default'...
.....
-----
Ran 16 tests in 0.126s

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

And the functional tests too:

```
$ python3 manage.py test functional_tests
[...]
=====
ERROR: test_cannot_add_empty_list_items
-----
(functional_tests.test_list_item_validation.ItemValidationTest)
  File "/workspace/superlists/functional_tests/base.py", line 30, 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:
[...]
```

So close! Let's look at where this is happening — we check the line number from the validation FT, and see it's happening after we've submitted a blank list item. We see the error text on the next page, but the form has disappeared!

Now, looking in *views.py*, we see that we're not passing the form to the *home.html* template inside the *new\_list* view:

```
except ValidationError:
    error_text = "You can't have an empty list item"
    return render(request, 'home.html', {"error": error_text})
```

This is a job our form should be doing! 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 adjust the unit tests for the *new\_list* view. Instead of manually checking for a hard-coded error string, we check for the `EMPTY_LIST_ERROR` from *forms.py*, and we can also check that a form of the right class was passed to the template.

*lists/tests/test\_views.py (ch09l066).*

```

from lists.forms import ItemForm, EMPTY_LIST_ERROR
[...]

class NewListTest(TestCase):
    [...]

    def test_validation_errors_sent_back_to_home_page_template(self):
        response = self.client.post('/lists/new', data={'text': ''})
        self.assertEqual(Item.objects.all().count(), 0)
        self.assertTemplateUsed(response, 'home.html')
        self.assertContains(response, escape(EMPTY_LIST_ERROR))
        self.assertIsInstance(response.context['form'], ItemForm)

```

Sure enough, the form isn't being passed to the template:

```
KeyError: 'form'
```

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 bad case, we pass the form down to the template, instead of our hard-coded error string.

Incidentally, did you notice that we've also fixed a small bug? In the previous code, we were saving a superfluous List object, even for invalid inputs. They would have been left lying around our database. We should add an extra assert in our unit tests once this refactor is done.

- remove duplication of validation logic in views.
- add test that we don't save superfluous lists

At this point the tests will fail, because we're not yet using the form to display errors in the template:

*lists/templates/base.html (ch10l026).*

```

<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 re-work `view_list` as well, before we can get back to a working state.

## Using the form in the final view

This view handles both GET and POST requests. Let's start with checking the form is used in GET requests. Let's add a new test for that:

```

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"')

```

*lists/tests/test\_views.py.*

That gives:

```

KeyError: 'form'

```

Here's a minimal implementation:

```

def view_list(request, list_id):
    [...]
    form = ItemForm()
    return render(request, 'list.html', {'list': list_, "form": form, "error": error})

```

*lists/views.py (ch10l023).*

## A helper method for several short tests

Onto invalid forms. We'll split our current single test for the invalid case (`test_validation_errors_end_up_on_lists_page`) into several separate ones:

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

    def post_invalid_input(self):
        list_ = List.objects.create()
        return self.client.post(
            '/lists/%d/' % (list_.id,),
            data={'text': ''}
        )

    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'], ItemForm)
        self.assertContains(response, escape(EMPTY_LIST_ERROR))
```

Look at that — by making a little helper function, `post_invalid_input`, we can make three separate tests without duplicating lots of lines of code.

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.

```
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:

```
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 19 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 fairly major change to our small app — that input field, its name and ID, is absolutely critical to making everything work. We've touched 7 or 8 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.

- ~~remove duplication of validation logic in views.~~
- add test that we don't save superfluous lists

Definitely time for a commit.

```

$ git diff
$ git commit -am"use form in all views, back to working state"

```

Before we forget, let's add our check that invalid new list forms don't create a pointless list object:

```

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')
    self.assertContains(response, escape(EMPTY_LIST_ERROR))

```

*lists/tests/test\_views.py (ch09l069).*

That should pass. Commit, strike final item off list!

```
$ git commit -am"extra test for not saving List on invalid input"
```

- remove duplication of validation logic in views.
- add test that we don't save superfluous lists

## 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()`:

```
def test_form_save_handles_saving_to_a_list(self): lists/tests/test_forms.py (ch10l027).
    form = ItemForm(data={'text': 'do me'})
    new_item = form.save()
```

Django isn't happy, because an item needs to belong to a list:

```
django.core.exceptions.ValidationError: {'list': ['This field cannot be null.']}
```

Our solution is to tell the form's save method what list it should save to:

```
from lists.models import Item, List lists/tests/test_forms.py.
[...]

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.all()[0])
    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:

```
def save(self, for_list): lists/forms.py (ch10l029).
    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 by far the neatest.

```
Ran 20 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})
```

Re-run the test to check everything still passes:

```
Ran 20 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})
```

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.

## The request.POST or None trick

There’s one final, optional refactor I want to show you, which I got from my colleague Hansel Dunlop (and who in turn got it from Danny Greenfield, I believe):

*lists/views.py.*

```
def view_list(request, list_id):
    list_ = List.objects.get(id=list_id)
    form = ItemForm(data=request.POST or None)
    if form.is_valid():
        form.save(for_list=list_)
```

```
        return redirect(list_)
    return render(request, 'list.html', {'list': list_, "form": form})
```

It relies on the fact that the `or` operator, in Python, evaluates to the second argument if the first is falsy. Constructing a form with `data=None` is equivalent to creating an “unbound” form, for which `is_valid()` will always be `False`. On the other hand, if there is something in the POST dict, then we *will* pass it to the form, and `is_valid()` will do its normal validation work.

A quick peek at the [Django docs on bound and unbound forms](#) may be worth it at this point.

So that trick saves us at least two lines of code and one level of indentation, but some people might argue it’s less readable. It’s up to you whether you adopt it!

TODO: this trick is actually rather frowned on it turns out. Basically it will go wrong in some edge cases where `request.POST` is the empty dict, and thus falsy, but it could still be valid user input. There’s more info on [Pydanny’s blog](#). Thanks to Mark Lavin for pointing it out to me, especially for highlighting the fact that unticked checkboxes don’t appear in POST dictionaries, but they may well be valid inputs to a form.

So, it’s probably best if you *don’t* follow my advice and use this trick. I’ll probably remove this section altogether before the book is finished...

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.

### *Single assertion per test*

One assertion per test is the ultimate goal, but it’s OK if you don’t start out that way. Helper functions can keep them from getting too bloated.



# More advanced Forms

Now let's look at some more advanced forms usage. We've helped our users to avoid blank list items, let's help them avoid duplicate items.



If any chapter is going to get the chop, this is a likely one. So, please do let me know what you think: is it valuable? Which are your favourite and least favourite bits? What would you try and preserve from it, if it were to disappear?

## Another FT for duplicate items

We add a second test method to `ItemValidationTest`:

```
functional_tests/test_list_item_validation.py (ch11l001).
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

We add another test to our model unit tests to check that duplicate items in the same list raise an error

```
def test_cannot_save_duplicate_items(self): lists/tests/test_models.py (ch09l028).
    list1 = List.objects.create()
    Item.objects.create(list=list1, text='bla')
    with self.assertRaises(ValidationError):
        Item.objects.create(list=list1, text='bla')
```

And, while it occurs to us, we add another test to make sure we don't overdo it on our integrity constraints:

```
def test_CAN_save_same_item_to_different_lists(self): lists/tests/test_models.py (ch09l029).
    list1 = List.objects.create()
    list2 = List.objects.create()
    Item.objects.create(list=list1, text='bla')
    Item.objects.create(list=list2, text='bla') # 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:

```
class Item(models.Model): lists/models.py (ch09l030).
    text = models.TextField(unique=True)
    list = models.ForeignKey(List)
```

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 60, in
test_CAN_save_same_item_to_different_lists
    Item.objects.create(list=list2, text='bla') # 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 screwy 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 to do 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:

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

    class Meta:
        unique_together = ('list', 'text')

    def save(self, *args, **kwargs):
        [...]
```

*lists/models.py (ch09l031).*

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:

TODO: Some users have reported they don't see this unexpected fail. Test on various platforms, amend if necessary.

```
=====
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
[...]
```

That's a bit of a puzzler. A bit of print-based debugging:

```
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')
```

*lists/tests/test\_models.py.*

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:

```
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]
    )
```

*lists/tests/test\_models.py (ch09l032).*

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 tests:



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.

```
def test_string_representation(self):  
    list1 = List.objects.create()  
    item1 = Item.objects.create(list=list1, text='some text')  
    self.assertEqual(str(item1), item1.text)
```

*lists/tests/test\_models.py (ch09l033).*

That gives us:

```
AssertionError: 'Item object' != 'some text'
```

As well as the other two failures. Let's start fixing them all now:

```
class Item(models.Model):  
    [...]  
  
    def __str__(self):  
        return self.text
```

*lists/models.py (ch09l034).*



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 2 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:

```
class Meta:  
    ordering = ('id',)  
    unique_together = ('list', 'text')
```

*lists/models.py (ch09l035).*

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 in our test:

TODO: investigate new Django test helper, “assertQuerySetEqual”?

```
self.assertEqual(
    list(Item.objects.all()),
    [item1, item2, item3]
)
```

*lists/tests/test\_models.py (ch09l036).*

That works, we get a fully passing test suite:

OK

Time for a commit!

```
$ git diff
$ git commit -am "Implement duplicate item validation at model layer"
```

The next task is to handle the validation error in the view. Before we do that, a quick aside, for the curious. Do you remember I mentioned earlier that some data integrity errors *are* picked up on save? Try temporarily disabling our `.full_clean` in the model save:

```
def save(self, *args, **kwargs):
    #self.full_clean()
    super().save(*args, **kwargs)
```

*lists/models.py.*

That gives

```
ERROR: test_cannot_save_duplicate_items
(lists.tests.test_models.ListAndItemModelsTest)
    return Database.Cursor.execute(self, query, params)
django.db.utils.IntegrityError: columns list_id, text are not unique
```

[... and a bunch of other failures due to validation not working any more]

Note that it's a different error to the one we want, an `IntegrityError` instead of a `ValidationError`.

## Handling validation at the views layer

Let's put our `full_clean` back, and 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 <sup>1</sup>. A quick unit test at the view level ought to clear this up:

```
lists/tests/test_views.py (ch11l014).
def test_invalid_input_renders_form_with_errors(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'}
    )

    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.core.exceptions.ValidationError: {'__all__': ['Item with this List and
Text already exists.']}
```

Here's one possible solution:

```
lists/views.py (ch11l015).
if form.is_valid():
    try:
        form.save(for_list=list_)
        return redirect(list_)
    except ValidationError:
        form.errors.update({'text': "You've already got this in your list"})
    return render(request, 'list.html', {'list': list_, "form": form})
```

OK, we know that's an ugly hack, we need to get the form to do this work, we did for new lists, but — it will probably work.

```
Ran 25 tests in 0.104s
```

```
OK
```

```
Ran 4 tests in 19.048s
```

```
OK
```

What matters is that it gets us to *Green*. We'll definitely do the *Refactor* part very soon. First, a commit.

1. It's showing a server error, code 500. Gotta get with the jargon!

```
$ git diff
$ git commit -am"duplicate item validation hacked in at views level"
```

## 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 up our tests for the previous form, tweaking them slightly:

```

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:

```

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:

```
class ExistingListItemForm(ItemForm):
    def __init__(self, for_list, *args, **kwargs):
        super().__init__(*args, **kwargs)
```

*lists/forms.py (ch09l072).*

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:

```
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)
```

*lists/forms.py.*

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 up to the form.

And we're there! A quick commit

```
$ git diff
$ git commit -a
```

## Using the existing lists item form in the list view

Now let's see if we can put this form to work in our view.

Let's start by using our constant, now that we've defined it. Tidy tidy!

*lists/tests/test\_views.py (ch11l049).*

```
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)

Ran 28 tests in 0.130s
```

OK

Next — let's find places where we should be using our new form class. There's two places in our view tests where we check the form, let's adjust them both:

*lists/tests/test\_views.py (ch11l050).*

```
def test_displays_item_form(self):
    list_ = List.objects.create()
    response = self.client.get('/lists/%d/' % (list_.id,))
    self.assertIsInstance(response.context['form'], ExistingListItemForm)

def test_invalid_input_renders_form_with_errors(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 (ch11l051).*

```
from lists.forms import ExistingListItemForm, ItemForm
[...]
def view_list(request, list_id):
    list_ = List.objects.get(id=list_id)
    form = ExistingListItemForm(for_list=list_, data=request.POST or None)
    if form.is_valid():
        form.save()
        return redirect(list_)
    return render(request, 'list.html', {'list': list_, "form": form})
```

And, oops, 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 (ch11l053).*

```
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 (ch11l054).*

```

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! Unit tests pass!

```

$ python3 manage.py test lists
[...]
Ran 29 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 re-run *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 show all view test + 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"') #5
    def test_displays_only_items_for_that_list(self):
        self.assertContains(response, 'itemey 1') #6
        self.assertContains(response, 'itemey 2') #7
        self.assertNotContains(response, 'other list item 1') #8
    def test_can_save_a_POST_request_to_an_existing_list(self):
        self.assertEqual(Item.objects.all().count(), 1) #9
        self.assertEqual(new_item.text, 'A new item for an existing list') #10
    def test_POST_redirects_to_list_view(self):
        self.assertRedirects(response, '/lists/%d/' % (correct_list.id,)) #11
    def test_invalid_input_means_nothing_saved_to_db(self):
        self.assertEqual(Item.objects.all().count(), 0) #12
    def test_invalid_input_renders_list_template(self):
        self.assertTemplateUsed(response, 'list.html') #13
    def test_invalid_input_renders_form_with_errors(self):
        self.assertIsInstance(response.context['form'], ExistingListItemForm) #14
        self.assertContains(response, escape(EMPTY_LIST_ERROR)) #15

```

- 1 Use the Django Test Client
- 2 Check the template used. Then, check each item in the template context:
- 3 Check any objects are the right ones, or Querysets have the correct items.
- 4 Check any forms are of the correct class
- 6 7 Test any template logic: any for or if should get a minimal test
- 8
- 9 10 For views that handle POST requests, make sure you test both the valid case and
- 11 12 the invalid case.
- 13
- 5 14 Sanity-check that your form is rendered, and its errors are displayed
- 15

Why these points? Skip ahead to [Appendix II](#), 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: pushing our changes up to the server.

---

# Database migrations

We’ve made a change to our database — we’ve added some constraints to some of the columns in the list table. In order to apply these to our live site, we’ll need to alter the tables in the existing database. This is a *database migration*.

## South vs Django migrations

The current established tool for database migrations with Django is called **South**.



At the time of writing, Andrew Godwin, South’s talented creator, had just finished a **project** to integrate South into the Django core, in the shape of a new feature called **Migrations**. I plan to upgrade to this as soon as it gets released in Django 1.7... The basic concepts and steps involved aren’t going to change much though.

We start by installing South:

```
$ ../virtualenv/bin/pip install south
[...]
```

Successfully installed south

We add it to `INSTALLED_APPS`:

```
INSTALLED_APPS = (
    [...]
    'lists',
    'south',
)
```

*superlists/settings.py (ch12l001).*

And to *requirements.txt*:

*requirements.txt.*



```
Django==1.6
unicorn==18.0
South==0.8.3
```

We can commit that to version control:

```
$ git commit -am "Add South"
```

## Creating an initial migration to match the current live state

For migrations to work, they need to know what we're migrating from and to. The place this really matters is on the live server, so we want to be able to migrate from the database state as it currently is on live, to the state that's in the latest version of the code.

Here's where version control comes in useful. You remember in chapter 8 we tagged the current release? That makes it easy for us to get an old version of our code that matches what's deployed.

### If you didn't tag the release back in chapter 8

If you were working from a previous version of the book, you might not have tagged the release. You know, because I just only decided to add that bit. Instead, you can use `git log` to look back for a commit that we made during chapter 8, or maybe right at the beginning of chapter 9.

```
$ git log --oneline
[...]
18480bd Create base FT class and a class for each test.
87e99b5 Moved functional tests into a folder.
7fa00f1 New ft for item validation.
8ca488b Add a fabfile for automated deploys # <--- this looks like it!
d28e6ea Notes and template config files for provisioning
6b0d814 Add unicorn to virtualenv requirements
6a6c91e Add requirements.txt for virtualenv
```

Note down that commit number and then use it to retrospectively add the tag:

```
$ git tag LIVE 8ca488b # substitute in your own commit number!
```

Then you should be able to follow on with the instructions.

To revert our models to the state they were in at that point:

```
$ git checkout LIVE -- lists/models.py
```

Now we can create our initial migration. This tells south what the “starting” state of the database should be:

```
$ ../virtualenv/bin/python3 manage.py schemamigration lists --initial
Creating migrations directory at '/workspace/superlists/lists/migrations'...
[...]
```

```
+ Added model lists.List
+ Added model lists.Item
Created 0001_initial.py. You can now apply this migration with: ./manage.py
migrate lists
```

The migration is stored in a directory called *migrations* inside the lists app:

```
$ tree lists/migrations/
lists/migrations/
├── 0001_initial.py
├── __init__.py
└── __pycache__
```

We go back to the latest code version:

```
$ git checkout HEAD -- lists/models.py
```

And we add the initial migration to version control:

```
$ git add lists/migrations/
$ git status # should show 2 new files
$ git commit -m"initial migration to match live"
```

Next we create the “real” migration that we want to apply. This time we use `--auto`:

```
$ ../virtualenv/bin/python3 manage.py schemamigration --auto lists
+ Added unique constraint for ['list', 'text'] on lists.Item
Created 0002_auto__add_unique_item_list_text.py. You can now apply this
migration with: ./manage.py migrate lists
```

Sure enough, it spots the new constraint. Let’s add that to VCS too:

```
$ git add lists/migrations/0002_auto__add_unique_item_list_text.py
$ git commit -m"Add new migration for list item uniqueness constraint"
```

## Migrations: like a VCS for your database

The way migrations work is that they store a series of pictures of what your database looks like as your code evolves. In order to apply a migration, the migrations tool also needs to have a view of what the database *currently* looks like.

But, because we only started using migrations half-way through our development (and this is quite a common occurrence), the migrations tool doesn’t know where it currently is. In order to tell it, we do what’s called a “fake” migration, to tell it that the current database state is at the 0001 migration that we stored earlier, and that the one we want to go to is 0002.

If that hasn’t melted your brain enough, how about this: so South needs to store information about what it thinks the current state of the database is, right? And where do you think it stores it? You can see this coming, can’t you? It stores it in the database. South has its own set of tables. Please don’t ask me whether south stores information in those tables about the tables themselves.

Let's test this out locally and try and get used to it. First, we go and make a database that has the old state, ie one that looks like live:

```
$ git checkout LIVE
$ rm ../database/db.sqlite3
$ python3 manage.py syncdb --noinput
$ git checkout master
```

Now, if you try doing a migration, you'll see that South explodes violently:

```
$ ../virtualenv/bin/python3 manage.py syncdb --migrate
Syncing...
Creating tables ...
Creating table south_migrationhistory
[...]
Migrating...
Running migrations for lists:
- Migrating forwards to 0002_auto__add_unique_item_list_text.
  > lists:0001_initial
FATAL ERROR - The following SQL query failed: CREATE TABLE "lists_list" ("id"
integer NOT NULL PRIMARY KEY)
[...]
! NOTE: The error which caused the migration to fail is further up.
Error in migration: lists:0001_initial
[...]
django.db.utils.OperationalError: table "lists_list" already exists
```

It's because it's confused about the current state of the database. It thinks it needs to create the lists table, but it's already there. Here's how we tell it that the database actually matches migration 0001, ie the place where live is:

```
$ ../virtualenv/bin/python3 manage.py migrate lists --fake 0001
Running migrations for lists:
- Migrating forwards to 0001_initial.
  > lists:0001_initial
    (faked)
```

And now we can test applying the real migration we want to do to live:

```
$ ../virtualenv/bin/python3 manage.py migrate lists
Running migrations for lists:
- Migrating forwards to 0002_auto__add_unique_item_list_text.
  > lists:0002_auto__add_unique_item_list_text
- Loading initial data for lists.
Installed 0 object(s) from 0 fixture(s)
```

Brilliant! Are you confused? I am, slightly, and I'm the one writing this. Here's a recap:

- We need to apply a database migration to the live database when we deploy, to add the uniqueness constraint.

- We're going to use South migrations for this. We've created two migrations, one (0001) which takes us from nothing to the old state, and one which takes us from there to the state we want (0002).
- To test this, we've created a database in the same state as live by checking out our old model code and doing a syncdb.
- In order to apply a migration, South needs to know what the current state of the database is.
- We tell it by applying a “fake” version of migration 0001.
- Then we're in a position to apply the real migration, 0002.

So how are we actually going to do this on our live servers? By replicating those last two steps. We're using a fabfile for our deployments, so let's adjust it now:

```

def _update_database(source_folder):
    run('cd %s && ../virtualenv/bin/python3 manage.py syncdb' % (source_folder,))
    # one-off fake database migration. remove me before next deploy
    run('cd %s && ../virtualenv/bin/python3 manage.py migrate lists --fake 0001' % (
        source_folder,
    ))
    run('cd %s && ../virtualenv/bin/python3 manage.py migrate' % (source_folder,))

```

Still nervous? Me too, but that's why we have a staging environment. Here goes nothing! We start by pushing up our latest changes so that we can pull them down on the server:

```
$ git push
```

And deploy!

```

$ cd deploy_tools
$ fab deploy --host=superlists-staging.ottg.eu
[superlists-staging.ottg.eu] Executing task 'deploy'
[superlists-staging.ottg.eu] run: mkdir -p
/home/harry/sites/superlists-staging.ottg.eu

[...]

[superlists-staging.ottg.eu] run: cd
/home/harry/sites/superlists-staging.ottg.eu/source &&
../virtualenv/bin/python3 manage.py syncdb
[superlists-staging.ottg.eu] out: Syncing...
[superlists-staging.ottg.eu] out: Creating tables ...
[superlists-staging.ottg.eu] out: Creating table south_migrationhistory
[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:
[superlists-staging.ottg.eu] out: Synced:
[superlists-staging.ottg.eu] out: > django.contrib.auth
[superlists-staging.ottg.eu] out: > django.contrib.contenttypes

```

```

[superlists-staging.ottg.eu] out: > django.contrib.sessions
[superlists-staging.ottg.eu] out: > django.contrib.sites
[superlists-staging.ottg.eu] out: > django.contrib.messages
[superlists-staging.ottg.eu] out: > django.contrib.staticfiles
[superlists-staging.ottg.eu] out: > functional_tests
[superlists-staging.ottg.eu] out: > south
[superlists-staging.ottg.eu] out:
[superlists-staging.ottg.eu] out: Not synced (use migrations):
[superlists-staging.ottg.eu] out: - lists
[superlists-staging.ottg.eu] out: (use ./manage.py migrate to migrate these)
[superlists-staging.ottg.eu] out:

[superlists-staging.ottg.eu] run: cd
/home/harry/sites/superlists-staging.ottg.eu/source &&
../virtualenv/bin/python3 manage.py migrate lists --fake 0001
[superlists-staging.ottg.eu] out: - Soft matched migration 0001 to 0001_initial.
[superlists-staging.ottg.eu] out: Running migrations for lists:
[superlists-staging.ottg.eu] out: - Migrating forwards to 0001_initial.
[superlists-staging.ottg.eu] out: > lists:0001_initial
[superlists-staging.ottg.eu] out: (faked)
[superlists-staging.ottg.eu] out:
[superlists-staging.ottg.eu] run: cd
/home/harry/sites/superlists-staging.ottg.eu/source &&
../virtualenv/bin/python3 manage.py migrate
[superlists-staging.ottg.eu] out: Running migrations for lists:
[superlists-staging.ottg.eu] out: - Migrating forwards to
0002_auto__add_unique_item_list_text.
[superlists-staging.ottg.eu] out: > lists:0002_auto__add_unique_item_list_text
[superlists-staging.ottg.eu] out: - Loading initial data for lists.
[superlists-staging.ottg.eu] out: Installed 0 object(s) from 0 fixture(s)
[superlists-staging.ottg.eu] out:

```

Looks good. We then go in and restart our web server:

server commands.

```
user@server:$ sudo restart gunicorn-superlists-staging.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:

```

$ cd deploy_tools
$ fab deploy --host=superlists.ottg.eu
[superlists.ottg.eu] Executing task 'deploy'

```

[...]

You'll need to restart the live unicorn job too.

## Wrap-up: remove fake migration and git tag

Before we forget, let's remove that fake migration from the fabfile. We don't want to run that next time we deploy, because south is now in sync on the server. In fact we can simplify it down to a single command, the `syncdb --migrate`:

```
def _update_database(source_folder):  
    run('cd %s && ../virtualenv/bin/python3 manage.py syncdb --migrate --noinput' % (  
        source_folder,  
    ))  
    deploy_tools/fabfile.py (ch12l005).
```

We commit that:

```
$ git status  
$ git commit -am "deploy script now does syncdb --migrate"
```

And finally we tag our latest release:

```
$ git tag -f LIVE # needs the -f because we are replacing the old tag  
$ export TAG=`date +%F/%H%M`  
$ git tag $TAG  
$ git push -f origin LIVE $TAG
```



We went through quite a bit of pain setting up that fake migrations. If we'd started using South at the time of our first deployment, none of this would have been necessary. I hope that going through this more complex procedure has given you more of an insight into how South works but... In your real projects, start using South from the very first deploy!

## On testing database migrations

We've now tested out our migration locally, and we've run it once on the staging site. We've tested that our application still works after the migration, both locally and on staging, using our functional test suite. We're comfortable that we can modify our database schema. Is there anything else we need to do?

You might worry that the most dangerous thing about a migration isn't so much that we can adjust our database schema, but more that we might lose data during the change. Shouldn't we somehow test that the existing data in the database is still there after we migrate?

The answer to that is: you should if you're *particularly* nervous. Hopefully you've now got enough building blocks from this book to see how you might be able to write some automated tests that would do just that.

## Don't test third party code

One of the rules of thumb in testing is “don't test third party code”. If you're using some kind of external library, you can't afford to spend your time writing tests for their code as well as your own — you just have to decide whether you trust them or not. South is an incredibly popular tool, it's been around for ages, and we can be pretty confident that it's going to do what it says it does.

## Do test migrations for speed

One thing you should be testing is how long your migrations are going to take. Database migrations typically involve down-time, 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.

And on that stern note, time to move on to the next chapter! Hopefully it'll have something fun in it to cheer us up. Oh, wait --

# 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
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()) #2

```

- 1 2** `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 (ch13l002).
def get_error_element(self):
    return self.browser.find_element_by_css_selector('.has-error')

```

And we then make 5 replacements in *test\_list\_item\_validation*, like this one for example:

```

functional_tests/test_list_item_validation.py (ch13l003).
# 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. (I trust you to do this without needing me to show you how any more!)

## 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 **py.test** to be very impressive. But there is a clear default option, and it's just fine.<sup>1</sup>

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

1. Admittedly once you start looking for Python BDD tools, things are a little more confusing.

end there either: as I had settled on one of them Mocha <sup>2</sup> you find out that you 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.



I was dead keen to use Mocha, because of the Nyan cat test runner. If someone can show me a simple mocha setup that lets the user test, in the browser and from the command-line as well, using the “tdd” ui and the “assert” assertion library... Well, I'll be eternally grateful.

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>
```

2. purely because it features the **NyanCat**

```
</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.



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 this:

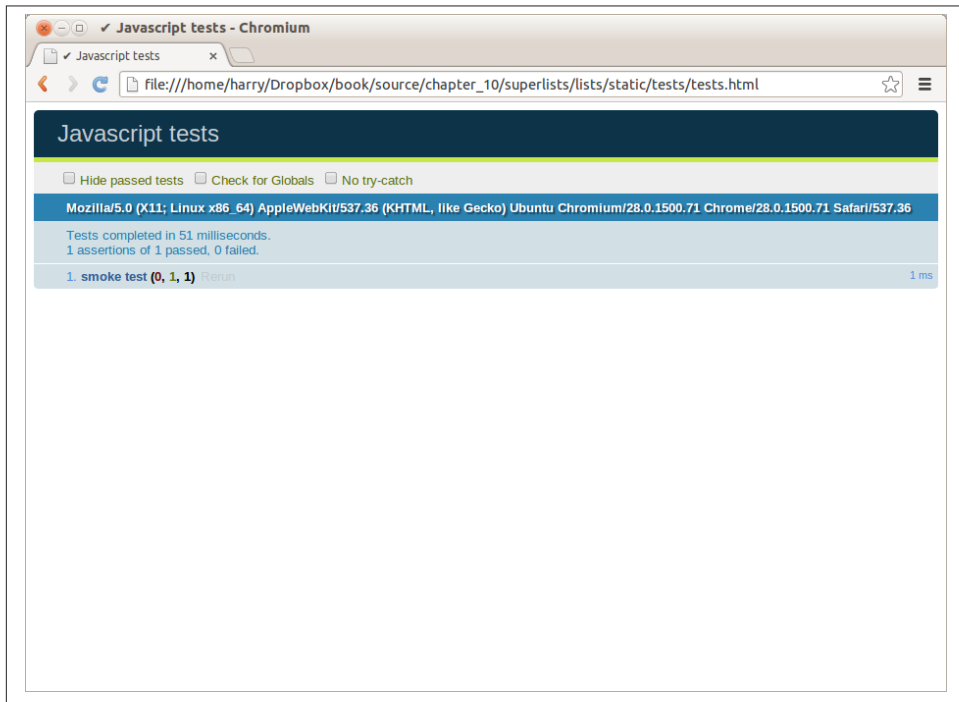


Figure 13-1. Basic Qunit screen

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 Here we use `.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 browser.

```
2 assertions of 2 passed, 0 failed.  
1. smoke test (0, 2, 2)
```

Time to see how fixtures work. If we just dupe up this test:

```
<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>
```

*lists/static/tests/tests.html.*

Slightly unexpectedly, we find one of them fails:

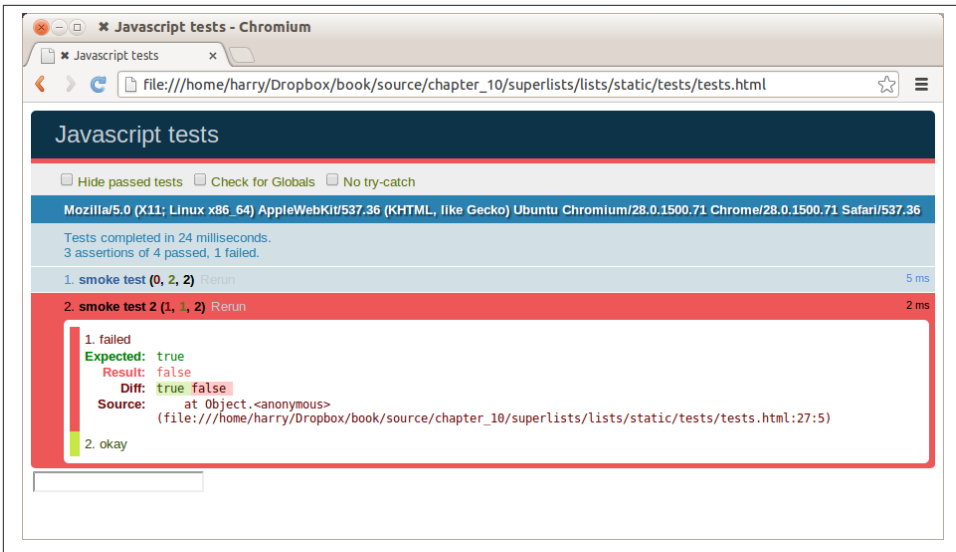


Figure 13-2. One of the two tests is failing

What's happening here is that the first test hides the error div, so when the second test runs, it starts out invisible.



Qunit test 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. Thankfully, and you can probably see this coming, but the `qunit-fixture` div is exactly what we're looking for. Move the form in there:

```

<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>

```

*lists/static/tests/tests.html.*

And that gets us back to 2 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

Switch back to just one test:

```
<script>
/*global $, test, equal */

test("errors should be hidden on keypress", function () {
    $('input').trigger('keypress'); // ❶
    equal($('.has-error').is(':visible'), false);
});

</script>
```

*lists/static/tests/tests.html.*

- ❶ The jQuery `.trigger` method is mainly used for testing. It says “fire off a JavaScript 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. Just check out [Quirksmode.org](http://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*

```
<script src="qunit.js"></script>
<script src="../list.js"></script>
<script>
```

*lists/static/tests/tests.html.*

Here’s the minimal code to get that test to pass:

```
$('.has-error').hide();
```

*lists/static/list.js.*

It has an obvious problem. We’d better add another test:

```

test("errors should be hidden on keypress", function () { lists/static/tests/tests.html.
    $('input').trigger('keypress');
    equal($('.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 () {
    $('.has-error').hide();
});

```

That gets our unit tests to pass!

Grand, so let's pull in our script, and jquery, on all our pages:

```

lists/templates/base.html (ch13l014).
</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_
[...]

Ran 1 test in 3.023s

OK

```



Hooray! That's a commit!

## 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:

```
$(document).ready(function () {  
    $('input').on('keypress', function () {  
        $('.has-error').hide();  
    });  
});
```

*lists/static/list.js.*

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:

```
jQuery(document).ready(function ($) {  
    $('input').on('keypress', function () {  
        $('.has-error').hide();  
    });  
});
```

*lists/static/list.js.*

Read more in the [jQuery .ready\(\) docs](#).

I leave it to you to re-run the javascript tests, and then maybe a full FT run, just to reassure yourself that nothing is broken. And then, it's onto user authentication!

### 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*.

TODO: take the opportunity to use `{% static %}` tag in templates?



---

## User authentication, integrating 3rd party plugins, and Mocking with JavaScript

So 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 federated authentication system instead.

(If you *insist* on storing your own passwords, Django’s default auth module 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, and 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](mailto: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? 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. This 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 3rd 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 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 chapters 10 & 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 3 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. Feel free to add this code to your own site too, and then you can have a play with it too!

Let's start with the front-end. I was able to cut & paste code from the Persona site and Dan's slides with minimal modification:

```
lists/templates/base.html (ch14l001).
<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 a 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 (ch14l002).*

```
<body>
  <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="container">
    [...]
```

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 re-use 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 into 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's more details on the [Mozilla site](#), including all the clever public key crypto that keeps Google from knowing what site you want to log into, but also stops replay attacks and so on. Smart.

Before starting on a spike, it's a good idea to start a new branch:

```
$ git checkout -b persona-auth-spike
```

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's definitely “spike” code, from things like that commented-out line as evidence of an early experiment that failed. We'll definitely put something tidier into production.

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 re-use 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`. 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:

```

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 hard-coded 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 (ch14l006).
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 (ch14l007).
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:

```
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'),
)
```

And finally, 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',
'south',
'accounts',
)

AUTH_USER_MODEL = 'accounts.ListUser'
AUTHENTICATION_BACKENDS = (
    'accounts.authentication.PersonaAuthenticationBackend',
)

MIDDLEWARE_CLASSES = (
    [...]

```

Why not spin up a dev server with `runserver` and see how it all looks?

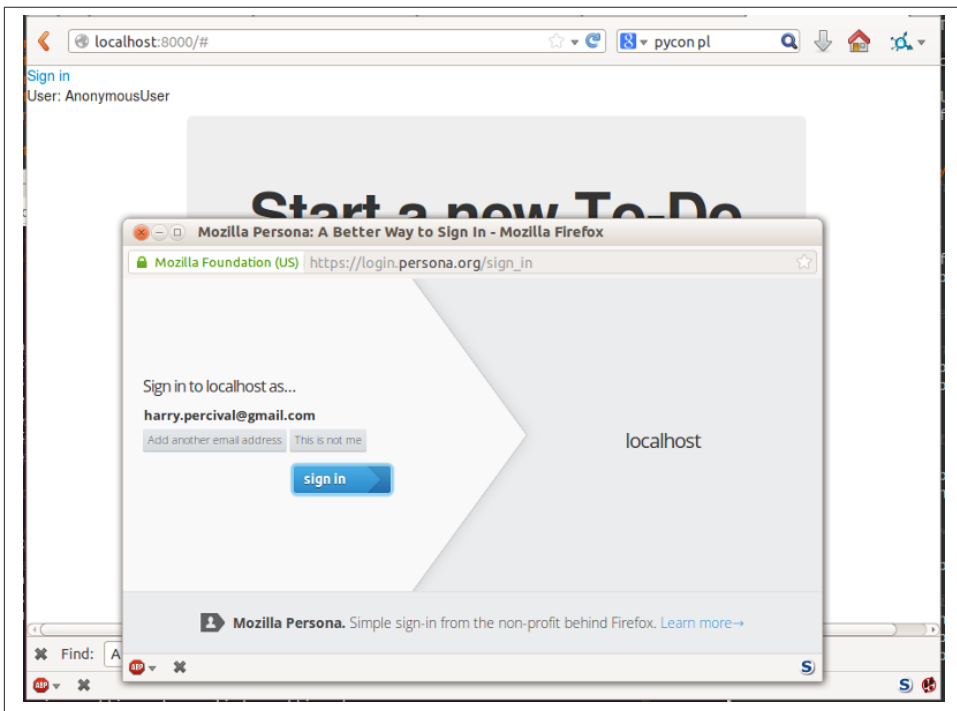


Figure 14-1. It works! It works! mwahahahaha.

NB - you will need to run a syncdb to get the accounts tables all set up.

That's pretty much it! Along the way, I had to fight pretty hard, including debugging ajax requests by hand in the Firefox console, catching infinite page-refresh loops, stum-

bling over several missing attributes on my custom user model (because I didn't read the docs properly), and finally discovering that we have to upgrade to the dev version of Django to actually get it to work in the admin site.<sup>1</sup>

## 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 exceptions to the terminal by default, but you can make it do so with a variable called `LOGGING` in `settings.py`:

*superlists/settings.py (ch14l011).*

```
LOGGING = {
    'version': 1,
    'handlers': {
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
        },
    },
    'loggers': {
        'django': {
            'handlers': ['console'],
        },
    },
}
```

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. More info in the [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 re-writing your prototype code using TDD. We now have enough information to “do it properly”. So what's the first step? An FT of course!

1. <http://stackoverflow.com/questions/16983547/django-problems-with-id-in-custom-usermodel/18458659#18458659>

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.

## A common Selenium technique: waiting for

Here's the basic outline:

```
from .base import FunctionalTest functional_tests/test_login.py.

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')
        self.browser.find_element_by_id(
            'authentication_email'
        ).send_keys(TEST_EMAIL)
        self.browser.find_element_by_tag_name('button').click()

        # We get redirected to the Yahoo page
        self.wait_for_element_with_id('username')
        self.browser.find_element_by_id(
            'username'
        ).send_keys(TEST_EMAIL)
        self.browser.find_element_by_id(
            'passwd'
        ).send_keys(TEST_PASSWORD)
        self.browser.find_element_by_id('.save').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')
        navbar = self.browser.find_element_by_css_selector('.navbar')
        self.assertIn(TEST_EMAIL, navbar.text)
```

Where did I get TEST\_EMAIL and TEST\_PASSWORD from? I just set up a free web-mail account especially for this... You could do the same, or check out <http://www.mockmyid.com> and <http://www.personatestuser.org>.

TODO: demo of how to run thru process manually with firefox debug toolbar to find correct locators in case not same as the ones I came across

TODO: wider discussion of benefits of mocked service vs real thing

The FT needs two helper functions, both of which do something that's very common in Selenium testing: they wait for something to happen. Here's the first:

```
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')
```

*functional\_tests/test\_login.py (ch14l014).*

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:

```
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)
    )
```

*functional\_tests/test\_login.py (ch14l015).*

This is what Selenium calls an "explicit wait". If you remember, we already defined an "implicit wait" in the `FunctionalTest.setUp`. We set that to just 3 seconds though, 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!

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
[...]
.
```

```
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 re-introduce 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 # save our new FT.
$ git commit -m "FT for login with Persona"
```

Now we re-run 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 (ch14l017).
self.browser.find_element_by_id('id_login').click()
[...]
self.wait_for_element_with_id('id_logout')
```

OK — so let's add a do-nothing log in 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!

First off, a bit of housekeeping. We create a site-wide static files directory inside *superlists/superlists*, and we move all the bootstrap css, qunit code, and base.css into it

```
$ 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

TODO: move base.html into site-wide folder too?

That means adjusting our existing JavaScript unit tests:

```

                                                                    lists/static/tests/tests.html (ch14l020).
<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 can re-run them to check that worked:

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'),
)
```





You might want to re-introduce the `LOGGING` setting from earlier at this point. There's no need for an explicit test for this, and our current test suite will let us know in the unlikely event that it breaks anything.

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(); };  
}
```

## Why Mock?

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* 3rd party function in our unit tests, because we don't need our unit tests popping up `persona` windows all over the shop. So instead, we are going to do what's called "mocking it out". 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. Ah well, needs must; thankfully, it's quite a straightforward one. Still, you may find you need to read this next section a few times before it all makes total sense.

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.

## 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”<sup>2</sup> function. The code we’ll end up with in *base.html* will look like this:

lists/templates/base.html.

```
<script src="https://login.persona.org/include.js"></script>
<script src="/static/accounts/accounts.js"></script>
<script src="/static/list.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`!

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 though. 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>

<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
```

2. 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

```

        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
- ❻ 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`
- 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`.

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.



I've called this object a "mock", but it's probably more correctly called a "spy". 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.

Does that make sense? Let's play around with the test and see if we can get the hang of it.

Our first error is this:

```
1. Died on test #1
@file:///workspace/superlists/accounts/static/tests/tests.html:34:
Superlists is not defined
```

That's the equivalent of an `ImportError` in Python. Let's start work on *accounts/static/accounts.js*:

```
/*global $ */
window.Superlists = null;
```

We start with the usual on-document-ready boilerplate, and then address our immediate problem: Superlists is not defined. Now, just as in Python we might do `Superlists = None`, 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:34:
Superlists is null
```

OK, next baby step or two:

```
window.Superlists = {
  Accounts: {}
};
```

Gives <sup>3</sup>

```
Superlists.Accounts.initialize is not a function
```

So let's make it a function:

```
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
```

3. 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, jQuery's `$.extend` is another possibility. But, there's already a lot of content in this chapter, and I thought this was probably one too many things to talk about!

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:

```
var initialize = function (navigator) {  
    console.log(navigator);  
};  
  
window.Superlists = {  
    Accounts: {  
        initialize: initialize  
    }  
};
```

*accounts/static/accounts.js (ch14|028).*

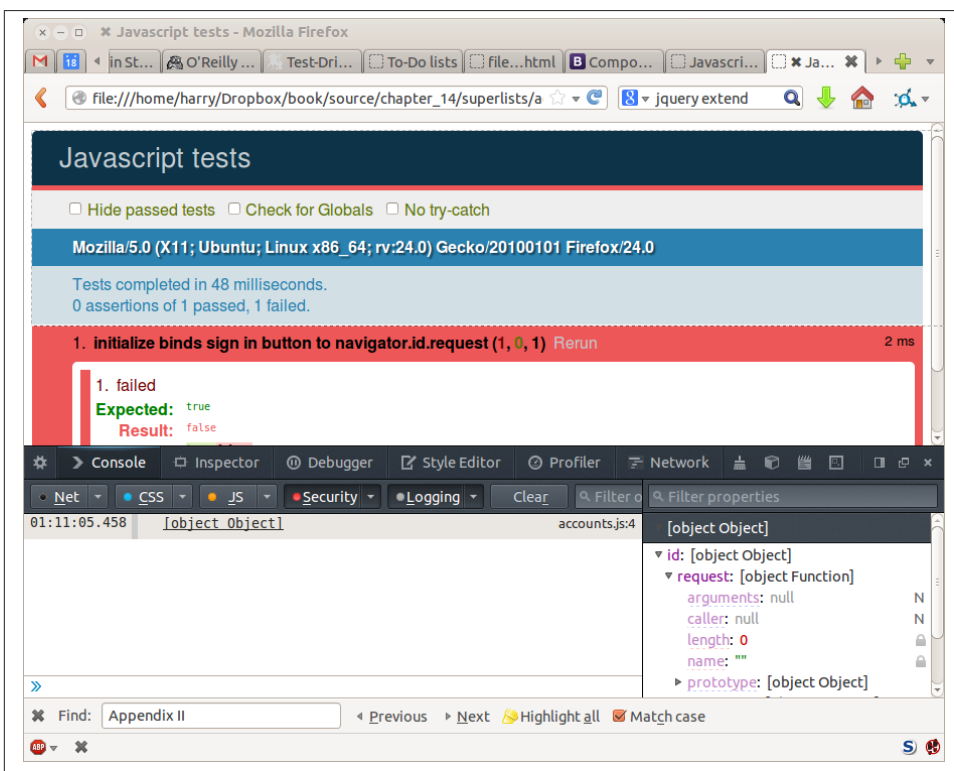


Figure 14-2. Debugging in the JavaScript console

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. 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`.

So let's now just pile in and get the test to pass:

```
var initialize = function (navigator) {  
  navigator.id.request();  
};
```

*accounts/static/accounts.js (ch14l029).*

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:

```
var initialize = function (navigator) {  
  navigator.id.request();  
  navigator.id.doSomethingElse();  
};
```

*accounts/static/accounts.js.*

We get:

```
1. Died on test #1  
@file:///workspace/superlists/accounts/static/tests/tests.html:34:  
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:

```
var mockNavigator = {  
  id: {  
    request: mockRequestFunction,  
    doSomethingElse: function () { console.log("called me!"); }  
  }  
};
```

*accounts/static/tests/tests.html.*

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:

```
var mockNavigator = {  
  id: {  
    request: mockRequestFunction  
  }  
};
```

*accounts/static/tests/tests.html (ch14l032).*

```
};

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_login` is clicked:

```
var initialize = function (navigator) {
  $('#id_login').on('click', function () {
    navigator.id.request();
  });
};
```

*accounts/static/accounts.js.*

That passes. A good start! Let’s try pulling it into our template:

```
<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>
  $(document).ready( function () {
    Superlists.Accounts.initialize(navigator);
  });
</script>
</body>
```

*lists/templates/base.html.*

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`:

```
+++ b/superlists/settings.py
@@ -130,6 +130,7 @@ INSTALLED_APPS = (
    'lists',
    'south',
```

*superlists/settings.py.*

```
+ '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

We've also got two hard-coded URLs in there, which it 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 spy 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>
```

And now we can write a test that uses sinon’s so-called “spy” object:

```
accounts/static/tests/tests.html (ch14l038).
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.spy() //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.spy()` 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...

```
var initialize = function (navigator, user, token, urls) {  
    $('#id_login').on('click', function () {  
        navigator.id.request();  
    });  
  
    navigator.id.watch();  
});
```

*accounts/static/accounts.js (ch14l039).*

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:35:  
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:

```
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 () {}  
        }  
    };  
});
```

*accounts/static/tests/tests.html.*

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:

```
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.spy()  
        }  
    };  
  
    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 spy 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:71:  
watchCallArgs is undefined
```

Because we're not currently passing any arguments to watch. Step-by-step, we can do

```
navigator.id.watch({});
```

*accounts/static/accounts.js (ch14l043).*

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:

```
navigator.id.watch({  
    loggedInUser: user  
});
```

*accounts/static/accounts.js (ch14l044).*

## 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 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, which you can find out more about here: <http://api.jquery.com/jquery.post/>

Let's add this “module” after the first test, before the test for “initialize calls navigator.id.watch”:

```
accounts/static/tests/tests.html (ch14l045).
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.spy()
      }
    };
    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 unit test `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 spy 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 (ch14l046).
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 (ch14l047).
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 spy 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, 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 `$.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 (ch14l052).

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 (ch14l053).

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 (ch14l054).

},
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 <sup>4</sup>, 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:

```

                                accounts/static/tests/tests.html (ch14l055).
test("onlogin post failure should do navigator.id.logout ", function () {
  mockNavigator.id.logout = sinon.spy(); //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 spy 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 (ch14l056).
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 (ch14l057).
navigator.id.watch({
  loggedInUser: user,
  onlogin: function (assertion) {
    $.post(
      urls.login,

```

4. you can't mock out `window.location.reload`, so instead you have to define an (untested) function called eg `Superlists.Accounts.refreshPage`, and then put a spy on *that* to check that it gets set as the `ajax.done` callback



```

        { 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.logout);
deferred.always(function () { window.location.reload(); });

```

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:

```

<script>
$(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>

```

*lists/templates/base.html.*

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 57, in
test_login_with_persona
    self.wait_for_element_with_id('id_logout')
  File "/workspace/superlists/functional_tests/test_login.py", line 26, 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 3rd party API. Whilst it is possible to “roll your own” mocks in JavaScript, a mocking framework like Sinon.js provides a lot of helpful shortcuts which will make it easier to write (and more importantly, read) your tests.

### *Unit testing Ajax*

Sinon.js is a great help here. Manually mocking Ajax methods is a real pain.



---

# Server-side authentication and mocking in Python

Let's crack on with the server side of our new auth system.



as with all new chapters, I'd really appreciate feedback. How is the pace? What do you think of the “left as an exercise for the reader” bit at the end?

## Mocking in Python

Here's the spiked version of our view:

```
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('/')
```

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.

The popular *mock* package was added to the standard library as part of Python 3. It provides a magical object called a Mock, which is a bit like the sinon spy 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:

(I trust you to set up a tests folder with a dunderinit. Don't forget to delete the default *tests.py*, as well.)

```

from django.test import TestCase
from unittest.mock import patch

```

*accounts/tests/test\_views.py.*

```

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

```

- ❶ The decorator called patch is a bit like the sinon spy 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`:

```
from django.contrib.auth import authenticate
```

*accounts/views.py.*

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:

```
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)),
)
```

*superlists/urls.py.*

```
from django.conf.urls import patterns, url

urlpatterns = patterns('',
    url(r'^login$', 'accounts.views.login', name='login'),
)
```

*accounts/urls.py.*

Will a minimal view do anything?

```
from django.contrib.auth import authenticate

def login():
    pass
```

*accounts/views.py.*

Yep:

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

And so:

```
def login(request):
    pass
```

*accounts/views.py (ch15l008).*

Then

```
ValueError: The view accounts.views.login didn't return an HttpResponse object.
```

*accounts/views.py (ch15l009).*

```
from django.contrib.auth import authenticate
from django.http import HttpResponseRedirect
```

```
def login(request):
    return HttpResponseRedirect()
```

And we're back to:

```
AssertionError: Expected 'authenticate' to be called once. Called 0 times.
```

We try:

```
def login(request):
    authenticate()
    return HttpResponseRedirect()
```

*accounts/views.py.*

And sure enough, we get:

```
AssertionError: Expected call: authenticate(assertion='assert this')
Actual call: authenticate()
```

And then we can fix that too:

```
def login(request):
    authenticate(assertion=request.POST['assertion'])
    return HttpResponseRedirect()
```

*accounts/views.py.*

OK so far. One Python function mocked and tested.

But our authenticate view also needs to call the Django auth.login function if authenticate returns a user, and 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. We'll need to mock out the auth\_login view as well:

```
from django.contrib.auth import get_user_model
from django.http import HttpRequest
from django.test import TestCase
from unittest.mock import patch
```

*accounts/tests/test\_views.py (ch15l012).*

```
User = get_user_model() #❶
```

```
from accounts.views import login
```

```
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')

```

```

@patch('accounts.views.auth_login')
@patch('accounts.views.authenticate')
def test_calls_auth_login_if_authenticate_returns_a_user(
    self, mock_authenticate, mock_auth_login
):
    request = HttpRequest()
    request.POST['assertion'] = 'asserted'
    mock_user = mock_authenticate.return_value
    login(request)
    mock_auth_login.assert_called_once_with(request, mock_user)

```

```

@patch('accounts.views.auth_login')
@patch('accounts.views.authenticate')
def test_does_not_call_auth_login_if_authenticate_returns_None(
    self, mock_authenticate, mock_auth_login
):
    request = HttpRequest()
    request.POST['assertion'] = 'asserted'
    mock_authenticate.return_value = None
    login(request)
    self.assertFalse(mock_auth_login.called)

```

- 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)

Notice that, for these tests, we go back to importing the view function directly, and calling it with an `HttpRequest` we build manually. The Django Test Client does a bit too much magic, and for these highly mocky tests, we need more control — we need to check that `auth_login` was passed the same request object that we called the view with, for example, and that's not possible if you use the Django client.

That gives us:

```

$ python3 manage.py test accounts
[...]
AttributeError: <module 'accounts.views' from
'/workspace/superlists/accounts/views.py'> does not have the attribute
'auth_login'
[...]
AttributeError: <module 'accounts.views' from
'/workspace/superlists/accounts/views.py'> does not have the attribute
'auth_login'

```



```
[...]
AssertionError: '' != 'OK'
+ OK
```

Adding the import takes us down to two failures:

```
from django.contrib.auth import authenticate
from django.contrib.auth import login as auth_login
from django.http import HttpResponseRedirect
[...]
```

*accounts/views.py.*

And we go through another couple of TDD cycles, until:

```
def login(request):
    user = authenticate(assertion=request.POST['assertion'])
    if user:
        auth_login(request, user)
    return HttpResponseRedirect('OK')
```

*accounts/views.py.*

...

OK

## De-spiking our custom authentication back-end: mocking out an internet request

Our custom authentication back-end is next! Here's how it looked in the spike:

*accounts/authentication.py.*

```
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 4 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 3 steps (make sure the Goat sees you doing each one individually!)

```
def authenticate(self, assertion):
    requests.post(
        PERSONA_VERIFY_URL,
        data={'assertion': assertion, 'audience': DOMAIN}
    )
```

*accounts/authentication.py.*

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:

```
@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)
```

*accounts/tests/test\_authentication.py (ch15l020).*

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 “3 strikes” rule:

```
@patch('accounts.authentication.requests.post') #❶
class AuthenticateTest(TestCase):
```

*accounts/tests/test\_authentication.py (ch15l021).*

```

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 for test for the positive case where authenticate should return a user object. We expect this to fail.

```

accounts/tests/test_authentication.py (ch15l022).
from unittest.mock import Mock, patch
from django.contrib.auth import get_user_model
User = get_user_model()
from django.test import TestCase
[...]

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 (ch15l023).

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.all()[0]
```

That gets our new test passing, but other the other tests fail:

```
ERROR: test_returns_none_if_response_errors
IndexError: list index out of range
[...]
ERROR: test_returns_none_if_status_not_okay
IndexError: list index out of range
[...]
ERROR: test_sends_assertion_to_mozilla_with_domain
IndexError: list index out of range
```

The positive case passes, we now need to build some of our guards for cases where authentication should fail — if the response errors, or if the status is not okay. Let's start with this:

```
accounts/authentication.py (ch15l024).

def authenticate(self, assertion):
    response = requests.post(
        PERSONA_VERIFY_URL,
        data={'assertion': assertion, 'audience': DOMAIN}
    )
    if response.json()['status'] == 'okay':
        return User.objects.all()[0]
```

That actually fixes all three, slightly surprisingly:

OK

But we know we've got a couple of "cheats". Let's add the to our scratchpad for now.

- authenticate is just returning the first user it finds
- why is the test for response.ok passing?

## 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<sup>1</sup>. So, when we do our

```
if response.json()['status'] == 'okay':
```

What we actually end up doing is 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:

```
accounts/tests/test_authentication.py (ch15l025).
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 (ch15l026).
def authenticate(self, assertion):
    response = requests.post(
        PERSONA_VERIFY_URL,
        data={'assertion': assertion, 'audience': DOMAIN}
    )
    if response.ok and response.json()['status'] == 'okay':
        return User.objects.all()[0]
```

...

OK

Good.

- `authenticate` is just returning the first user it finds
- why is the test for `response.ok` passing?

1. 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 (ch15l027).
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.all()[0]
    self.assertEqual(found_user, new_user)
    self.assertEqual(found_user.email, 'a@b.com')
```

That fails as follows:

```
IndexError: list index out of range
```

So we add a `try/except`, returning an “empty” user at first:

```
accounts/authentication.py (ch15l028).
if response.ok and response.json()['status'] == 'okay':
    try:
        return User.objects.all()[0]
    except:
        return User.objects.create()
```

And that fails because our new user doesn't have the right email:

```
self.assertEqual(found_user.email, 'a@b.com')
AssertionError: '' != 'a@b.com'
```

And so we fix it by getting the email from the response json:

```
accounts/authentication.py (ch15l029).
if response.ok and response.json()['status'] == 'okay':
    email = response.json()['email']
    try:
        return User.objects.all()[0]
    except:
        return User.objects.create(email=email)
```

That gets us to passing tests:

```
$ python3 manage.py test accounts
[...]
Ran 9 tests in 0.019s
OK
```

## Tests the `get_user` method by mocking the Django ORM

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.

The simplest way to test this would be, as in the previous example, but creating actual objects in the database, and by letting the method use the ORM to find them, or not.

But, since we're learning about mocks, I thought I'd show how to mock out the Django ORM. Although I have no qualms about using the database in my unit tests, some people really don't like it: the objection is that “true” unit tests should have no external dependencies at all, and should never touch the database.



You can find out more about the “purist” approach to unit testing in the appendix entitled “The Database is Hot Lava”.

So, by way of an educational exercise, here's how to avoid touching the database by mocking out the Django ORM. We'll also learn about how to use mocks to test exception-handling, which will be useful.

```
accounts/tests/test_authentication.py (ch15l030).  
  
class GetUserTest(TestCase):  
  
    @patch('accounts.authentication.User.objects.get') #❶  
    def test_gets_user_from ORM_using_email(self, mock_User_get):  
        backend = PersonaAuthenticationBackend()  
        found_user = backend.get_user('a@b.com')  
        self.assertEqual(found_user, mock_User_get.return_value) #❷  
        mock_User_get.assert_called_once_with(email='a@b.com') #❸
```

- ❶ We patch out the User ORM class, so that we can check on any calls our code will make to it.
- ❷ We check that `get_user` gives us the return value from our mocked `User.objects.get`.
- ❸ We also check that `User.objects.get` was called correctly, passing in the email as an argument.

Here's our first failure:

```
AttributeError: 'PersonaAuthenticationBackend' object has no attribute  
'get_user'
```

Let's create a placeholder one then:

```
accounts/authentication.py (ch15l031).  
  
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

```
def get_user(self, email):
    pass
```

*accounts/authentication.py (ch15l032).*

Next:

```
self.assertEqual(found_user, mock_User_get.return_value)
AssertionError: None != <MagicMock name='get()' id='140631293381136'>
```

And (step-by-step):

```
def get_user(self, email):
    return User.objects.get()
```

*accounts/authentication.py (ch15l033).*

That gets us past the first assertion, onto the mock check:

```
AssertionError: Expected call: get(email='a@b.com')
Actual call: get()
```

And so we call get with the email as an argument:

```
def get_user(self, email):
    return User.objects.get(email=email)
```

*accounts/authentication.py (ch15l034).*

That gets us to passing tests: ...

OK

## Testing exception handling

The other thing we need to check is that our `get_user` function should return `None` if the user doesn't exist (this wasn't well documented at the time of writing, but that is the interface we have to comply with. See [the source](#)).

Ordinarily, if you do an ORM lookup and the user doesn't exist, it will raise an `Exception`. Since we're mocking out the ORM in these tests, we have to simulate that exception. This is a good chance to learn about how to test exception handling with mocks. We do it by setting a special Mock attribute called `side_effect`:

```
class GetUserTest(TestCase):
```

*accounts/tests/test\_authentication.py (ch15l035).*

```
    @patch('accounts.authentication.User.objects.get')
    def test_gets_user_from ORM_using_email(self, mock_User_get):
        [...]
```

```

@patch('accounts.authentication.User.objects.get')
def test_returns_none_if_user_does_not_exist(self, mock_User_get):
    def raise_no_user_error(*_, **__): #❶
        raise User.DoesNotExist()
    mock_User_get.side_effect = raise_no_user_error #❷
    backend = PersonaAuthenticationBackend()

    self.assertIsNone(backend.get_user('a@b.com'))

```

- ❶ We define a function whose only job is to raise an Exception. If you've not seen it before, I'm using the convention where variables named with underscores signify variables we don't care about — it's the equivalent of (\*args, \*\*kwargs), but we're just going to ignore what those args and kwargs are.
- ❷ We then assign that function as the "side\_effect" of our mocked User.objects.get function. When our code tries to call it, it will invoke our exception-raising function instead.

You can actually trace the effect of our mocking in the traceback we get:

```

ERROR: test_returns_none_if_user_does_not_exist
[...]
File "/workspace/superlists/accounts/tests/test_authentication.py", line 70,
in test_returns_none_if_user_does_not_exist
    self.assertIsNone(backend.get_user('a@b.com'))❶
File "/workspace/superlists/accounts/authentication.py", line 25, in get_user
    return User.objects.get(email=email)❷
File "/usr/lib/python3.3/unittest/mock.py", line 846, in __call__
    return _mock_self._mock_call(*args, **kwargs)❸
File "/usr/lib/python3.3/unittest/mock.py", line 911, in _mock_call
    ret_val = effect(*args, **kwargs)
File "/workspace/superlists/accounts/tests/test_authentication.py", line 66,
in raise_no_user_error❹
    raise User.DoesNotExist()
django.contrib.auth.models.DoesNotExist

```

- ❶ We call get\_user
- ❷ get\_user calls User.objects.get
- ❸ Because User.objects.get is mocked out, the call is diverted into the mock library's code
- ❹ Mock calls our pre-prepared exception-raising function.

As you can see, mocks are powerful, but they can be pretty mind-bending! That's why I always say you should avoid using them if you can. In any case, here's our implementation:

```

def get_user(self, email):
    try:

```

*accounts/authentication.py (ch15l036).*

```

        return User.objects.get(email=email)
    except User.DoesNotExist:
        pass

```

And we *almost* have a working authentication backend.

```

$ python3 manage.py test accounts
[...]
Ran 11 tests in 0.020s
OK

```

We still have one item on our scratchpad

- authenticate is just returning the first user it finds
- ~~why is the test for response.ok passing?~~

Before we can fix that, we'll need to 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!

```

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

```

*accounts/tests/test\_models.py.*

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?

```

from django.db import models

class User(models.Model):
    email = models.EmailField()

```

*accounts/models.py.*

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 (ch15l039).*

```
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 <sup>2</sup> question?

```
AttributeError: type object 'User' has no attribute 'USERNAME_FIELD'
```

So:

*accounts/models.py.*

```
class User(models.Model):
    email = models.EmailField()
    REQUIRED_FIELDS = ()
    USERNAME_FIELD = 'email'
```

What now?

```
accounts.user: The USERNAME_FIELD must be unique. Add unique=True to the field
parameters.
```

I'll do one better! Let's make the email field into the primary key, and thus implicitly remove the auto-generated id column:

*accounts/models.py (ch15l042).*

```
email = models.EmailField(primary_key=True)
```

## Tests as documentation

That gets our tests running. We'll write a test for this feature anyway, just as a form of documentation:

*accounts/tests/test\_models.py (ch15l043).*

```
def test_email_is_primary_key(self):
    user = User()
    self.assertFalse(hasattr(user, 'id'))
```

2. 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.

And just to double-check, we can temporarily switch back the `EmailField` to using `unique=True` instead of `primary_key=True`, just to see the test fail



People sometimes say that your tests are a form of documentation for your code — they express what your requirements are of a particular class. 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.

## A slight disappointment

Meanwhile, we have a weird unexpected failure:

```
$ python3 manage.py test accounts
[...]
ERROR: test_returns_OK_when_user_found
(accounts.tests.test_views.LoginViewTest)
  File "/workspace/superlists/accounts/tests/test_views.py", line 29, in
test_returns_OK_when_user_found
    response = self.client.post('/accounts/login', {'assertion': 'a'})
[...]
  File "/workspace/superlists/accounts/views.py", line 8, in login
    auth_login(request, user)
[...]
    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.

```
from django.db import models
from django.utils import timezone

class User(models.Model):
    email = models.EmailField(primary_key=True)
    last_login = models.DateTimeField(default=timezone.now)
    REQUIRED_FIELDS = ()
    USERNAME_FIELD = 'email'
```

*accounts/models.py.*

Right! Anyway! That works!

```
$ python3 manage.py test accounts
[...]
Ran 13 tests in 0.021s
OK
```

## Fixing our cheat

Remember our little “cheat”? Now that we have a custom user model that only know about email, we can apply a little tweak to our authentication unit test, which we couldn’t before:

```
accounts/tests/test_authentication.py (ch15l045).
def test_finds_existing_user_with_email(self, mock_post):
    mock_post.return_value.json.return_value = {'status': 'okay', 'email': 'a@b.com'}
    User.objects.create(email='someone@else.com')
    actual_user = User.objects.create(email='a@b.com')
    found_user = self.backend.authenticate('an assertion')
    self.assertEqual(found_user, actual_user)
```

We create the user with the username `someone@else.com`, to make sure our implementation doesn’t just grab any old user. Now the test fails:

```
FAIL: test_finds_existing_user_with_email
[...]
AssertionError: <User: User object> != <User: User object>
```

And so we can improve on our “cheat” implementation:

```
accounts/authentication.py (ch15l046).
def authenticate(self, assertion):
    [...]
    email = response.json()['email']
    try:
        return User.objects.get(email=email)
    except:
        return User.objects.create(email=email)
```

And we’re back to passing tests.

## 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>
    $(document).ready( function () {
        var user = "{{ user.email }}" || null;
        var token = "{{ csrf_token }}";
        var urls = {
            login: "% url '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 inter-related, 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:

```

# 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(TEST_EMAIL, navbar.text)

```

And, after 4 repetitions of very similar code, a helper function suggested itself:

*functional\_tests/test\_login.py.*

```

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(TEST_EMAIL, 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(TEST_EMAIL, navbar.text)

```

And I extended the FT like this:

```

[...]  

# 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:

```

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 %s. Page text was %s' % (
            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 39, 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
testinggoat@yahoo.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:

```
urlpatterns = patterns('',  
    url(r'^login$', 'accounts.views.login', name='login'),  
    url(r'^logout$', 'django.contrib.auth.views.logout', {'next_page': '/'}, name='logout'),  
)
```

*accounts/urls.py.*

And in `base.html`, we just make the logout into a normal URL link:

```
<a class="btn navbar-btn navbar-right" id="id_logout" href="{% url 'logout' %}">Log out</a>
```

*lists/templates/base.html.*

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 48 tests in 78.124s  
  
OK
```

## 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 error*

Be aware that mocking things out can cause counter-intuitive behaviour in `if` statements. Mocks are truthy, and they can also mask errors, because they have all attributes and methods.

### *Mocking the Django ORM*

If you want to avoid “touching” the database in your tests, you can use Mock to simulate the Django ORM. Sometimes, this is more trouble than it’s worth. See the “Hot Lava” appendix for more discussion.

*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, if you can.



---

# Test fixtures 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, it would be good 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 that 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 (ch16l002-2).
from selenium.webdriver.support.ui import WebDriverWait
[...]

class FunctionalTest(LiveServerTestCase):
    [...]

    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 (ch16l003).
def test_login_with_persona(self):
    [...]

    # She can see that she is logged in
    self.wait_to_be_logged_in(email=TEST_EMAIL)

    # 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(email=TEST_EMAIL)

    # Terrified of this new feature, she reflexively clicks "logout"
    self.browser.find_element_by_id('id_logout').click()
    self.wait_to_be_logged_out(email=TEST_EMAIL)

    # The "logged out" status also persists after a refresh
    self.browser.refresh()
    self.wait_to_be_logged_out(email=TEST_EMAIL)
```

Just to check we haven't broken anything, we re-run 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 (ch16l004).
def test_logged_in_users_lists_are_saved_as_my_lists(self):
    email = 'edith@email.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, and then we’ll have to figure out a way of managing the database on the test server.

```
$ fab deploy --host=superlists-staging.ottg.eu
[...]
```

And restart gunicorn...

```
elspeth@server: sudo restart gunicorn-superlists-staging.ottg.eu
```

## Staging finds an unexpected bug (that’s what it’s for!)

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 "/workspace/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 "/workspace/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.

I had considered just going back and fixing this in the previous chapter, and pretending it never happened, but I think leaving it in teaches a better lesson: first off, I'm not that smart, and second: this is exactly 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:

/etc/init/gunicorn-superlists-staging.ottg.eu.conf.

```
[...]
exec ../virtualenv/bin/gunicorn \
    --bind unix:/tmp/SITENAME.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)

```
def authenticate(self, assertion):
    logging.warning('entering authenticate function')
    response = requests.post(
        PERSONA_VERIFY_URL,
        data={'assertion': assertion, 'audience': settings.DOMAIN})
```

*accounts/authentication.py.*



```

    )
    logging.warning('got response from persona')
    logging.warning(response.content.decode())
    [...]

```

We restart gunicorn again, and then either re-run 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:b'{"status":"failure","reason":"audience mismatch: domain mismatch"}'

```

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 hardcoded as "localhost" in *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:

```

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:

```

# This setting is changed by the deploy script
DOMAIN = "localhost"

ALLOWED_HOSTS = [DOMAIN]

```

*superlists/settings.py.*

We feed that change back through the tests:

```

@@ -1,9 +1,9 @@
from unittest.mock import Mock, patch
+from django.conf import settings
from django.test import TestCase

from accounts.authentication import (
- PERSONA_VERIFY_URL, DOMAIN,
+ PERSONA_VERIFY_URL,
  PersonaAuthenticationBackend, User
)

@@ -28,7 +28,7 @@ class AuthenticateTest(TestCase):
    self.backend.authenticate('an assertion')

```

*accounts/test\_authentication.py.*

```

        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/authenticate.py.

@@ -1,8 +1,8 @@
import requests
from django.contrib.auth import get_user_model
+from django.conf import settings

PERSONA_VERIFY_URL = 'https://verifier.login.persona.org/verify'
-DOMAIN = 'localhost'
User = get_user_model()

@@ -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 not response.ok:
            return

```

Re-running the tests just to be sure:

```

$ python3 manage.py test accounts
[...]
Ran 18 tests in 0.053s
OK

```

Next we update our fabfile to make it adjust the domain in settings.py:

```

deploy_tools/fabfile.py.

def _update_settings(source_folder, site_name):
    settings_path = path.join(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 re-deploy, 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 re-run 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 `create_pre_authenticated_session` function only acts on the local database. Let's find out how to 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'
```

Ah, 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:

```
+++ 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
qns1ckvp2aga7tm6xuivyb0ob1akzzwl
```

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:

```
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='/',
        ))
    )

[...]
```

First let's see how we know whether or not we're working against the staging server. `self.against_staging` gets populated in *base.py*:

```

from .server_tools import reset_database                                     functional_tests/base.py.

class FunctionalTest(LiveServerTestCase):

    @classmethod
    def setUpClass(cls):
        for arg in sys.argv:
            if 'liveserver' in arg:
                cls.server_host = arg.split('=')[1] #❶
                cls.server_url = 'http://' + cls.server_host
                cls.against_staging = True #❷
            return
        LiveServerTestCase.setUpClass()
        cls.against_staging = False
        cls.server_url = cls.live_server_url

    @classmethod
    def tearDownClass(cls):
        if not self.against_staging:
            LiveServerTestCase.tearDownClass()

    def setUp(self):
        if self.against_staging:
            reset_database(self.server_host) #❸
        self.browser = webdriver.Firefox()
        self.browser.implicitly_wait(3)
```

- ❶ ❷ 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
- ❸ 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

In Python 2, you can call fabric functions directly from Python code. Because we're working with Python 3, we have to do an extra hop and call the `fab` command, 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), #❶
            '--host={}'.format(host),
            '--hide=everything,status', #❷
        ],
        cwd=THIS_FOLDER
    ).decode().strip() #❸

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 ❸ 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 Because of all the hopping around via fabric and subprocesses, we're forced to be quite careful about extracting the session key from the output of the command as it gets run on the server.



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 setup 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
```

I'm quite proud of that one. Anyway, let's see if it works...

```
$ python3 manage.py test functional_tests.MyListsTest \
--liveserver=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 re-run all the tests to make sure...

```
$ python3 manage.py test functional_tests --liveserver=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! But before we can deploy our actual live site, we'd better actually give the users what they wanted — the ability to save their lists.



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.



We're into dangerous territory, now that we have code that can directly affect the 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 used different keypairs for authentication, with different passphrases.



TODO: save logging code

TODO: bake in logging.warning if persona fails, using unit test.

## Fixtures, locally and on the server

### *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 log-in 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. Dealing with test fixtures is an important part of testing

### *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

### *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.

# 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, but now I’ll make it explicit, and talk about some of the common issues involved.

## The FT for “My Lists”

We know our `create_pre_authenticated_session` code works now, so we can just write our FT to look for a “My Lists” page:

```
def test_logged_in_users_lists_are_saved_as_my_lists(self):
    # Edith is a logged-in user
    self.create_pre_authenticated_session()

    # 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:

```

## Outside-in TDD

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 TDD - we design the system from the outside, and build up our code in layers. I’ll point out how 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.

## The outside layer: presentation & 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 *home.html*, in our navigation bar. Here’s the minimal code change:

```

                                                                    lists/templates/home.html (ch17002-1).
{% 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" href="{% url 'logout' %}">Log out</a>
[...]
```

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/home.html (ch17002-2).
<li><a href="{% url 'my_lists' user.email %}">My lists</a></li>
```

## Moving down one layer to view functions (the controller)

That will cause a template error, so we can move in one step, from the presentation layer down to the controller layer, Django's view functions.

As always, we start with a test:

```
lists/test_views.py (ch17l003).

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: False is not true : Template 'my_lists.html' was not a template
used to render the response. Actual template(s) used: <Unknown Template>
```

And we fix it, still at the views level, in *urls.py*, *views.py*, and by creating *my\_lists.html*:

```
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'),
)
```

Here's a minimal view:

```
lists/views.py (ch17l005).

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":'
```

## 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 can start to write the template code we'd like to use to get the my lists page to work the way we want it to. It forces us

to think about the API we want our code to have, from the point of view of the things that use it, rather than trying to work bottom-up.

## A quick re-structure 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 (ch17l007).

<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 %}

  {% block extra_content %}
  {% endblock %}

</div>
```

We haven't seen this feature of the Django template language yet: *list.html* and *home.html* now need to explicitly pull down the `list_form` block content using `{{ block.super }}`

```
lists/templates/home.html.

{% extends 'base.html' %}

{% block list_form %}{{ block.super }}{% endblock %}

{% block header_text %}Start a new To-Do list{% endblock %}

{% block form_action %}{% url 'new_list' %}{% endblock %}

lists/templates/home.html.

{% extends 'base.html' %}

{% block list_form %}{{ block.super }}{% endblock %}

{% block header_text %}Your To-Do list{% endblock %}

{% block form_action %}{% url 'view_list' list.id %}{% endblock %}

{% block table %}
```

```
<table id="id_list_table">
[...]
```

## Designing our API using the template

Meanwhile, *my\_lists.html* can just work in the new `extra_content` block:

```
{% extends 'base.html' %}
                                                                    lists/templates/my_lists.html.

{% block header_text %}My Lists{% endblock %}

{% block extra_content %}
    <h2>{{ owner.email }}'s lists</h2>
    <ul>
        {% for list in owner.list_set.all %}
            <li><a href="{{ list.get_absolute_url }}">{{ list.name }}</a></li>
        {% endfor %}
    </ul>
{% endblock %}
```

We’ve made several design decisions in this template which are going to filter their way down through the code:

- We want a variable called `owner` to represent the user in our template.
- 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)
- 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.

We can re-run 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)
```

This is a good 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

```
from django.contrib.auth import get_user_model
User = get_user_model()
[...]

def test_passes_owner_to_template(self):
    user = User.objects.create(email='a@b.com')
    response = self.client.get('/lists/users/a@b.com/')
    self.assertEqual(response.context['owner'], user)
```

*lists/tests/test\_views.py (ch17l011).*

Gives

```
KeyError: 'owner'
```

so

```
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})
```

*lists/views.py.*

We'll then get an error which will require adding a user to our other unit test

```
def test_my_lists_url_renders_my_lists_template(self):
    User.objects.create(email='a@b.com')
    [...]
```

*lists/tests/test\_views.py (ch17l013).*

And we get to an OK

```
OK
```

## The next “requirement” from the views layer

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:

```
from django.http import HttpRequest
[...]
from lists.views import new_list
[...]

class NewListTest(TestCase):
    [...]

    def test_list_owner_is_saved(self):
        request = HttpRequest()
        request.user = User.objects.create(email='a@b.com')
```

*lists/tests.py (ch17l014).*

```

request.POST['text'] = 'new list item'
new_list(request)
list_ = List.objects.all()[0]
self.assertEqual(list_.owner, request.user)

```

1

That fails as follows:

AttributeError: 'List' object has no attribute 'owner'

To fix this, we can try writing code like this:

*lists/views.py.*

```

def new_list(request):
    form = ItemForm(data=request.POST)
    if form.is_valid():
        list_ = List.objects.create()
        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 until we go down to the next layer and adjust the model.

## A more purist approach involving mocks

Is this “pure” outside-in TDD? No. A purist approach to outside-in TDD would want you to use mocks at this point, and have unit tests that are more isolated from one level to another. Something like this:

*lists/tests/test\_views.py.*

```

from unittest.mock import Mock, patch
[...]

@patch('lists.views.ItemForm.save', Mock()) #❶
@patch('lists.views.List.objects.create') #❷
def test_list_owner_is_saved_mocky(self, mock_List_create):
    request = HttpRequest()
    request.user = Mock()
    request.POST['text'] = 'new list item'
    mock_list = mock_List_create.return_value
    new_list(request)
    self.assertEqual(mock_list.owner, request.user) #❸

```

1. I've chosen to use the raw view function, and to manually construct an HttpRequest, rather than using the Django Test Client, because our custom authentication function module, since it relies on Persona, would need a mock to get the test to work. I think the non-mocky way is simpler, but, if you're curious, why not try and write it differently?



- ❷ We mock out the `List.objects.create` function to be able to get access to the list that's going to be created by the view.
- ❸ Then we can assert about the owner we assign to it
- ❶ This is needed because otherwise the `form.save()` will complain that it's not been passed a real `List` object.

Try it! You should find that it will pass, if you've added the `list_.owner = bit` to the view. Try removing the owner assignment, and you'll see it fail:

```
AssertionError: <MagicMock name='create().owner' id='140176904220432'> != <Mock id='140176904185168'>
```

Actually, *strictly* speaking, you'd need another check that the `list.owner` gets assigned *before* the save function is called, making the test even more complicated:

```

                                                                    lists/tests/test_views.py (ch17l016).
mock_list = mock_List_create.return_value
def check_owner_assigned_before_save():
    self.assertEqual(mock_list.owner, request.user)
mock_list.save.side_effect = check_owner_assigned_before_save

new_list(request)
```

So, yes, it's a more purist approach, but it does leave you with much mockier, and less readable tests. That's why I prefer a more pragmatic approach. I think “purist” Outside-In TDD, sometimes called “London-Style TDD”, isn't worth it when you're dealing with the Django ORM a lot — it works better if you have code that has no external dependencies or “boundaries”. There's more discussion of this in the “Hot Lava” chapter.

## Moving down again: to the model layer

Next we move down to the model layer, to get the `owner.list_set.all` API working:

```

                                                                    lists/tests/test_models.py (ch17l018).
from django.contrib.auth import get_user_model
User = get_user_model()
[...]

def test_list_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())
```

TODO: separate tests out into two test classes, one broadly for lists, the other broadly for items.

The naive implementation would be this:

```

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:

```
def test_list_owner_is_optional(self):  
    List.objects.create() # should not raise
```

*lists/tests/test\_models.py (ch17l020).*

The correct implementation is this:

```
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 resolve_url('view_list', self.id)
```

*lists/models.py.*

Now running the tests gives a 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 a schema migration

```
$ *python3 manage.py schemamigration lists --auto
```

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.
```

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. When they're not logged in, Django represents them using a class called AnonymousUser:

```
from django.contrib.auth.models import AnonymousUser  
[...]  
  
if form.is_valid():  
    list_ = List.objects.create()  
    if not isinstance(request.user, AnonymousUser):  
        list_.owner = request.user  
        list_.save()  
    form.save(for_list=list_)  
[...]
```

*lists/tests.py.*

And that gets us passing!

```
$ python3 manage.py test lists
Creating test database for alias 'default'...
.....
-----
Ran 35 tests in 0.237s
```

OK

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:

```
def test_list_name_is_first_item_text(self): lists/tests/test_models.py (ch17l023).
    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')

@property lists/models.py (ch17l024).
def name(self):
    return self.item_set.all()[0].text
```

And that, believe it or not, actually gets us a passing test, and a working “My Lists” page!

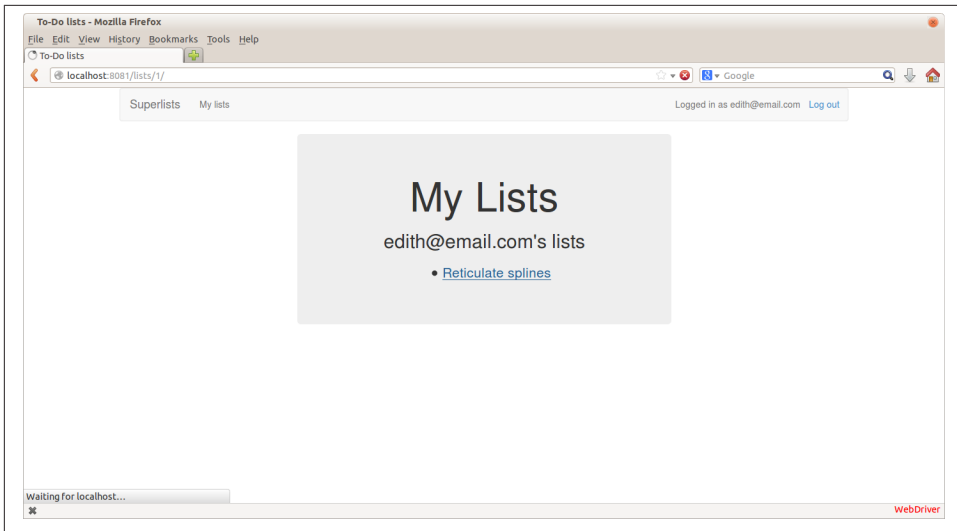


Figure 17-1. The “My Lists” page, in all its glory

```
$ python3 manage.py test functional_tests
Creating test database for alias 'default'...
.....
```

```
-----
Ran 7 tests in 93.819s
```

OK

I’ll tell you what though, those FTs are taking an annoyingly long time to run though. I wonder if there’s something we can do about that?

TODO: outside-in TDD wrap-up



---

## 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 pyVirtualDisplay

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/webdriver.py", line 58, in
self.binary, timeout),
File "/usr/local/lib/python3.3/site-packages/selenium/webdriver/firefox/extension_connection.py",
self.binary.launch_browser(self.profile)
File "/usr/local/lib/python3.3/site-packages/selenium/webdriver/firefox/firefox_binary.py", line 4
self._wait_until_connectable()
File "/usr/local/lib/python3.3/site-packages/selenium/webdriver/firefox/firefox_binary.py", line 9
```

```
self._get_firefox_output())
selenium.common.exceptions.WebDriverException: Message: 'The browser appears to have exited before
```

The fix is to use *pyVirtualDisplay*, which will magically start up a virtual display using an Xvfb framebuffer:

```
from selenium import webdriver
from pyvirtualdisplay import Display

display = Display()
display.start()

browser = webdriver.Firefox()

try:
    browser.get('http://localhost:8000')
    assert 'Django' in browser.title

finally:
    browser.quit()
    display.stop()
```

Running that gives us our expected failure

```
$ 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')
```

That should work better.<sup>1</sup>

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.



You'll need to remember to hit “Reload Web App” whenever you make changes to the code, to update the site.

## 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 17 if you're curious, there's some example code in there.



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, [\*obeythetesting@goat@gmail.com\*](mailto:obeythetesting@goat@gmail.com)





---

# Django Class-Based Views

This appendix follows on from Chapter 9, 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 chapter, 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

It's worth making a distinction at this point, 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 major benefit 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 re-use 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've found that using class-based views becomes much more complicated, and you end up with code that's harder to read than a classic view function.

Still, because we're forced to use a lot 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 tests 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:

```
from django.views.generic import FormView
[...]

class HomePageView(FormView):
    template_name = 'home.html'
    form_class = ItemForm
```

*lists/views.py (ch21l001).*

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`:

```
url(r'^$', HomePageView.as_view(), name='home'),
```

*superlists/urls.py (ch21l002).*

And the tests all check out! That was easy..

```
$ python3 manage.py test lists
Creating test database for alias 'default'...
.....
-----
Ran 28 tests in 0.119s

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

$ python3 manage.py test functional_tests
Creating test database for alias 'default'...
....
-----
Ran 4 tests in 15.160s

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

So far so good. We've replaced a 1-line view function with a 2-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. 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.*

```
class NewListView(CreateView):
    form_class = ItemForm

def new_list(request):
    form = ItemForm(data=request.POST)
    if form.is_valid():
        list = List.objects.create()
        Item.objects.create(text=request.POST['text'], list=list)
        return redirect(list)
    else:
        return render(request, 'home.html', {"form": form})
```

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.*

```
url(r'^new$', NewListView.as_view(), name='new_list'),
```

Now running the tests gives 3 errors:

```
$ python3 manage.py test lists
Creating test database for alias 'default'...
.....EEE
=====
ERROR: test_redirects_after_POST (lists.tests.test_views.NewListTest)
-----
Traceback (most recent call last):
  File "/home/harry/Dropbox/book/source/appendix_II/superlists/lists/tests/test_views.py", line 33, in test_redirects_after_POST
    data={'text': 'A new list item'}
    [...]
  File "/usr/local/lib/python3.3/dist-packages/django/views/generic/edit.py", line 165, in post
    return self.form_valid(form)
  File "/usr/local/lib/python3.3/dist-packages/django/views/generic/edit.py", line 127, in form_valid
    self.object = form.save()
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_sent_back_to_home_page_template (lists.tests.test_views.NewListTest)
-----
[...]
django.template.base.TemplateDoesNotExist: No template names provided
-----

Ran 22 tests in 0.114s

FAILED (errors=3)
Destroying test database for alias 'default'...
```

TODO: talk through decoding traceback.

Let's start with the third — maybe we can just add the template?

```
class NewListView(CreateView):                                lists/views.py.
    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():`:

```
class NewListView(CreateView):                                lists/views.py.
    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 28 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!

```
class NewListView(CreateView, HomePageView):                  lists/views.py.

    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.



This is not good object-oriented practice. Inheritance implies an “is-a” relationship, and I don’t think it’s appropriate to say that our view for creating new lists “is-a” home page view. Don’t 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 re-wrote some of chapters 9-11 as a result.

Anyway, after much hacking and swearing, this is the solution I eventually got to work. First I had to add a `get_absolute_url` on the `Item` class:

```
class Item(models.Model):
    [...]

    def get_absolute_url(self):
        return self.list.get_absolute_url()
```

*lists/models.py.*

Then I was able to get the view working using an override of the `get_form` method:

```
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)
```

*lists/views.py.*



I did also manage to get it working using `get_form_kwargs`, but I decided it was uglier. Any other suggestions are very much welcomed!

## Compare old and new

Let's see the old version for comparison?

```
def view_list(request, list_id):
    list_ = List.objects.get(id=list_id)
    form = ExistingListItemForm(for_list=list_, data=request.POST or None)
    if form.is_valid():
        form.save()
        return redirect(list_)
    return render(request, 'list.html', {'list': list_, "form": form})
```

Well, it's the same number of lines of code, 7. 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. But I guess some of it is in the eye of the beholder.

## Best practices for unit testing CBVs?

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 Integration tests.

They told me whether I was getting things right or wrong, but they didn't offer many 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_as_cbv(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 test with single assertions helps

One thing I definitely did conclude from this chapter was that having many short unit tests for views was much more helpful than having few tests with a narrative series of assertions.

One monolithic test like this:

```
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)
```

Was 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.





---

# Provisioning with Ansible



I'm not sure how or whether I'll include this in the final version of the book. It's currently very light, not really finished. Comment welcomed.

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:

```
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](#).

TODO: comments on switching everything over from fabric?

TODO: mention (demo?) Vagrant as a way of spinning up servers + vms