



Part 3: Advanced Web Development with Django

Real Python Part 3: Advanced Web Development with Django

Jeremy Johnson

Contents

1	Preface	9
	Thank you	9
	License	10
	Conventions	11
	Course Repository	13
	Errata	14
2	Introduction	15
	Welcome to Advanced Web Programming!	15
	Why this course?	17
	What is Software Craftsmanship?	18
	What will you learn?	19
	What will you build?	20
3	Software Craftsmanship	23
	Testing Routes and Views	29
	Refactoring our Tests	43
	Testing Models	48
	Testing Forms	56
	Testing Summary	60

4	Test Driven Development	63
	The TDD Workflow	64
	Implementing TDD with Existing Code	66
	Django Ecommerce Project	67
	Offense is only as good as your worst defense	77
	Conclusion	83
	Exercises	84
5	Git Branching at a Glance	85
	Git Branching	86
	Branching Models	87
	Enough about git branching	97
	Exercises	98
6	Upgrade, Upgrade, and Upgrade some more	99
	Django 1.8	99
	The Upgrade	100
	DiscoverRunner	106
	Update Project Structure	108
	Upgrading to Python 3	110
	Python 3 Changes Things Slightly	124
	Upgrading to PostgreSQL	125
	Conclusion	130
	Exercises	131
7	Graceful Degradation and Database Transactions with Django 1.8	132
	What is Graceful Degradation?	133
	User Registration	134
	Handling Unpaid Users	139
	Improved Transaction Management	142

What is a transaction?	143
What's wrong with transaction management prior to Django 1.6?	144
What's right about transaction management in Django 1.6 and above?	146
SavePoints	152
Nested Transactions	155
Completing the Front-end	157
Conclusion	159
Exercises.	161
8 Building a Membership Site	163
It's time to make something cool	164
A Membership Site	165
The User Stories	166
9 Bootstrap 3 and Best Effort Design	168
Start with the User Story	169
Overview of what we have	170
Installing Bootstrap 3	171
Making Bootstrap your own	176
HTML5 Sections and the Semantic Web	183
More Bootstrap Customizations	189
Custom template tags	196
A Note on Structure	201
Conclusion	205
Exercises	206
10 Building the Members Page	208
User Story	209
Update Main Template	210
Template 1: Showing User Info for the Current User	212

Gravatar Support	217
Template 2: Status Updating and Reporting	220
Template 3: Displaying Status Updates	226
Exercises	229
11 REST	230
Structuring a REST API	231
REST for MEC	234
Django REST Framework (DRF)	235
Now Serving JSON	247
Using Class-Based Views	254
Authentication	257
Conclusion	262
Exercises	263
12 Django Migrations	264
The problems that Migrations Solve	265
Getting Started with Migrations	266
The Migration File	267
When Migrations Don't Work	271
Data Migrations	275
Conclusion	281
Exercises	282
13 AngularJS Primer	283
What we are covering	285
Angular for Pony Fliers (aka Django Devs)	286
Angular vs Django - re: template tags	289
Angular Data Binding Explained	290
Building User Polls	294

Conclusion	306
Exercises	307
14 Django: Integrating Django and Angular	308
The User Polls Backend Data Model	309
A REST Interface for User Polls	311
Structural Concerns	316
Building the Template	318
Loading Poll Items Dynamically	325
Refactoring - progress bars	330
Refactoring - multiple users	332
Using Angular Factories	337
Conclusion	341
Exercises	342
15 Angular Forms	343
Field Validation	344
Display Error Messages	351
Form submission with Angular	355
How I taught an old view some new JSON tricks.	361
Fixing Tests	365
Breaking Chains	367
Conclusion	373
Exercises	374
16 MongoDB Time!	375
Building the front-end first	376
MongoDB vs SQL	381
Installing Mongo	383
Configuring Django for MongoDB	384

Django Models and MongoDB	386
A Geospatial primer	387
Mongoengine Support for Geospatial Features	389
Showing a Dot on a Map	390
Connecting the Dots	395
Getting User Locations	397
Conclusion	399
Exercises	401
17 One Admin to Rule Them All	402
Basic Admin	403
All your model are belong to us	405
Editing Stuff	414
Making things Puuurdy	418
Adminstrating Your Users	420
Resetting passwords	422
Conclusion	438
Exercises	441
18 Testing, Testing, and More Testing	442
Why do we need more testing?	442
What needs to be GUI tested?	443
Ensuring Appropriate Testing Coverage	445
GUI Testing with Django	446
Our First GUI Test Case	448
The Selenium API	450
Acting on elements you have located	454
Waiting for things to happen	458
Page Objects	461

When Selenium Doesn't Work	470
Limitations of Django's LiveServerTestCase	473
Conclusion	477
Exercises	478
19 Deploy	479
Where to Deploy	480
What to Deploy	481
Firing up the VPS	483
Configuring the OS	484
Django Setup	493
Configuration as Code	496
Let's script it	499
Continuous Integration	502
Conclusion	504
Exercises	505
20 Conclusion	506
21 Appendix A - Solutions to Exercises	508
Software Craftsmanship	508
Test Driven Development	514
Bootstrap 3 and Best Effort Design	521
Building the Members Page	536
REST	546
Django Migrations	552
AngularJS Primer	561
Djangular: Integrating Django and Angular	564
Angular Forms	572
MongoDB Time!	578

One Admin to Rule Them All	580
Testing, Testing, and More Testing	586
Deploy	590

Chapter 1

Preface

Thank you

I commend you for taking the time out of your busy schedule to improve your craft of software development. My promise to you is that I will hold nothing back and do my best to impart the most important and useful things I have learned from my software development career in hopes that you can benefit from it and grow as a software developer yourself.

Thank you so much for purchasing this course, and a special thanks goes out to all the early [Kickstarter](#) supporters that had faith in the entire Real Python team before a single line of code had been written.

License

This e-course is copyrighted and licensed under a [Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License](#). This means that you are welcome to share this course and use it for any non-commercial purposes so long as the entire course remains intact and unaltered. That being said, if you have received this copy for free and have found it helpful, I would very much appreciate if you [purchased](#) a copy of your own.

The example Python scripts associated with this course should be considered open content. This means that anyone is welcome to use any portion of the code for any purpose.

Conventions

NOTE: Since this is the Beta release, we do not have all the conventions in place. We are working on it. Patience, please.

Formatting

1. Code blocks will be used to present example code.

```
1 print "Hello world"!
```

2. Terminal commands follow the Unix format:

```
1 $ python hello-world.py
```

(dollar signs are not part of the command)

3. *Italic text* will be used to denote a file name:

hello-world.py.

Bold text will be used to denote a new or important term:

Important term: This is an example of what an important term should look like.

NOTES, WARNINGS, and SEE ALSO boxes appear as follows:

NOTE: This is a note filled in with bacon ipsum text. Bacon ipsum dolor sit amet t-bone flank sirloin, shankle salami swine drumstick capicola doner porchetta bresaola short loin. Rump ham hock bresaola chuck flank. Prosciutto beef ribs kielbasa pork belly chicken tri-tip pork t-bone hamburger bresaola meatball. Prosciutto pork belly tri-tip pancetta spare ribs salami, porchetta strip steak rump beef filet mignon turducken tail pork chop. Shankle turducken spare ribs jerky ribeye.

WARNING: This is a warning also filled in with bacon ipsum. Bacon ipsum dolor sit amet t-bone flank sirloin, shankle salami swine drumstick capicola doner porchetta bresaola short loin. Rump ham hock bresaola chuck flank. Prosciutto beef ribs kielbasa pork belly chicken tri-tip pork t-bone hamburger

bresaola meatball. Prosciutto pork belly tri-tip pancetta spare ribs salami, porchetta strip steak rump beef filet mignon turducken tail pork chop. Shankle turducken spare ribs jerky ribeye.

SEE ALSO: This is a see also box with more tasty ipsum. Bacon ipsum dolor sit amet t-bone flank sirloin, shankle salami swine drumstick capicola doner porchetta bresaola short loin. Rump ham hock bresaola chuck flank. Prosciutto beef ribs kielbasa pork belly chicken tri-tip pork t-bone hamburger bresaola meatball. Prosciutto pork belly tri-tip pancetta spare ribs salami, porchetta strip steak rump beef filet mignon turducken tail pork chop. Shankle turducken spare ribs jerky ribeye.

Course Repository

Like the other two courses, this course has an accompanying [repository](#). Broken up by chapter, in the “_chapters” folder, you can follow along with each chapter, then check your answers in the repository. Before you start each chapter, please make sure your local code aligns with the code from the previous chapter. For example, before starting chapter 6 make sure you’re code matches the code from the “chp5” folder from the repository.

You can download the course files directly from the repository. Press the ‘Download ZIP’ button which is located at the bottom right of the page. This allows you to download the most recent version of the code as a zip archive. **Be sure to download the updated code for each release.**

That said, it’s recommended that you work on one project throughout this course, rather than breaking it up.

NOTE: Since this is the Beta release, we do not have the structure completely updated in the repository. We will eventually change to a Git tagging model. If you have any suggestions or feedback, please contact us.

Errata

I welcome ideas, suggestions, feedback, and the occasional rant. Did you find a topic confusing? Or did you find an error in the text or code? Did I omit a topic you would love to know more about. Whatever the reason, good or bad, please send in your feedback.

You can find my contact information on the Real Python [website](#). Or submit an issue on the Real Python official support [repository](#). Thank you!

NOTE: The code found in this course has been tested on Mac OS X v. 10.8.5, Windows XP, Windows 7, Linux Mint 17, and Ubuntu 14.04 LTS.

Chapter 2

Introduction

Welcome to Advanced Web Programming!

Dear Reader:

Let me first congratulate you for taking the time to advance your skills and become a better developer. It's never easy in today's hectic world to carve out the time necessary to spend on something like improving your Python development skills. So I'm not going to waste your time, let's just jump right in.

```
1 # tinyp2p.py 1.0 (documentation at
   http://freedom-to-tinker.com/tinyp2p.html)
2 import sys, os, SimpleXMLRPCServer, xmlrpclib, re, hmac # (C) 2004,
   E.W. Felten
3 ar,pw,res = (sys.argv,lambda
   u:hmac.new(sys.argv[1],u).hexdigest(),re.search)
4 pxy,xs =
   (xmlrpclib.ServerProxy,SimpleXMLRPCServer.SimpleXMLRPCServer)
5 def ls(p=""):return filter(lambda n:(p=="")or
   res(p,n),os.listdir(os.getcwd()))
6 if ar[2]!="client": # license:
   http://creativecommons.org/licenses/by-nc-sa/2.0
7 myU,prs,srv = ("http://" + ar[3] + ":" + ar[4], ar[5:],lambda
   x:x.serve_forever())
8 def pr(x=[]): return ([y in prs] or prs.append(y) for y in x] or
   1) and prs
9 def c(n): return ((lambda f: (f.read(), f.close()))(file(n)))[0]
```



```

10 f=lambda p,n,a:(p==pw(myU))and(((n==0)and pr(a))or((n==1)and
    [ls(a)])or c(a))
11 def aug(u): return ((u==myU) and pr()) or
    pr(pxy(u).f(pw(u),0,pr([myU])))
12 pr() and [aug(s) for s in aug(pr()[0])]
13 (lambda sv:sv.register_function(f,"f") or
    srv(sv))(xs((ar[3],int(ar[4]))))
14 for url in pxy(ar[3]).f(pw(ar[3]),0,[]):
15     for fn in filter(lambda n:not n in ls(),
        (pxy(url).f(pw(url),1,ar[4]))[0]):
16         (lambda fi:fi.write(pxy(url).f(pw(url),2,fn)) or
            fi.close())(file(fn,"wc"))

```

OK, let me explain this line by line... no seriously - this is **NOT** what we are going to be doing. While this script is a cool example of the power of Python, it's painful to try to understand what's happening with the code and nearly impossible to maintain without pulling your hair out. It's a bit unfair to [Ed Felten](#), the author of this code, because he wrote this intentionally to fit the entire program in 15 lines. It's actually a pretty cool code example.

The point I'm trying to make is this:

There are many ways to write Python code. It can be a thing of beauty; it can be something very compact and powerful; it can even be a little bit scary. The inherent dynamic nature of Python, plus its extreme flexibility, makes it both awesome and dangerous at the same time. That's why this book is not only about writing cool web apps with Django and Python; it's about harnessing the power of Python and Django in a way that not only benefits from the immense power of the two - but in a way that is simple to understand, easy to maintain, and above all fun to work with

Why this course?

In today's market there's an abundance of programmers, and because of the huge push from the "everybody must learn to code" movement, that number is only going to increase.

In fact, [enrollment](#) in Computer Science degree programs for U.S. Universities rose by 29% in 2013. With everybody jumping in on the game, it may seem like the market will soon become saturated with would-be employees all vying for the same number of limited jobs. However, most high-tech companies tell a different story - the story of way too many resumes and not enough talent. *The truth is: Anybody can write code that looks like the sample from above, but far fewer developers can write truly maintainable, well structured, easy to understand, reliable code.*

That is exactly why the major theme of this course is **Software Craftsmanship**.

What is Software Craftsmanship?

Software Craftsmanship is the idea that writing software is about more than hammering on the keyboard until your program does what it's supposed to do. It's about focusing on the overall cost of ownership of the product - e.g., not only the costs of developing the software, but the cost of fixing all the defects and maintaining the software for years to come.

This is precisely the reason that Django was developed. Billed as “the framework for perfectionists with deadlines”, if you understand its intricacies, it's enjoyable to write beautiful, well-factored code, and to write it “with a deadline”.

What will you learn?

Quality and maintainability

We will dive into this idea of well-factored code and quality right from the start.

It's important to place an emphasis on quality and maintainability because those are the hallmarks of modern day software development, and they don't just happen by chance. They come about through the process of deliberately using a number of proven techniques and practices, all of which you will learn.

Advanced Django

Although Software Craftsmanship should underlie all that we do, it is of course important to understand the inner workings of the Django Framework so you can use it efficiently and effectively. So, we will cover specific Django topics in nearly every chapter in this course. We will cover all the usual suspects - like views, models and, templates as well as more advanced topics such as database transactions, class-based views, mixins, the Django Rest Framework, forms, permissions, custom fields, and so on.

This course is here to teach you all you need to know to develop great web applications that could actually be used in a production environment, not just simple tutorials that gloss over all the difficult stuff.

Full-stack Python

In today's world of HTML5 and mobile browsers, JavaScript is everywhere. Python and Django just are not enough. To develop a website from end to end, you must know the JavaScript frameworks, REST-based APIs, Bootstrap and responsive design, NoSQL databases ... and all the other technologies that make it possible to compete at the global level. That's why this course does not just cover Django exclusively. Instead, it covers all the pieces that you need to know to develop modern web applications from the client to the server - developing, testing, tooling, profiling, and deploying.

By the end of this course you will have developed a **REAL** web based application ready to be used to run an online business. Don't worry, we are not going to solely focus on buzzwords and hype. We're going to have fun along the way and build something awesome.

Throughout the entire course we focus on a single application, so you can see how applications evolve and understand what goes into making a **REAL** application.

What will you build?

We will start with the Django sample application introduced in the second course of this series, [Web Development with Python](#). Although it's highly recommend to complete course two before starting this one, there was only one chapter dedicated to that specific Django application, and all the code is in the associated Github [repository](#) for this course.

To give you an idea of where we'll build, the current application looks like this:

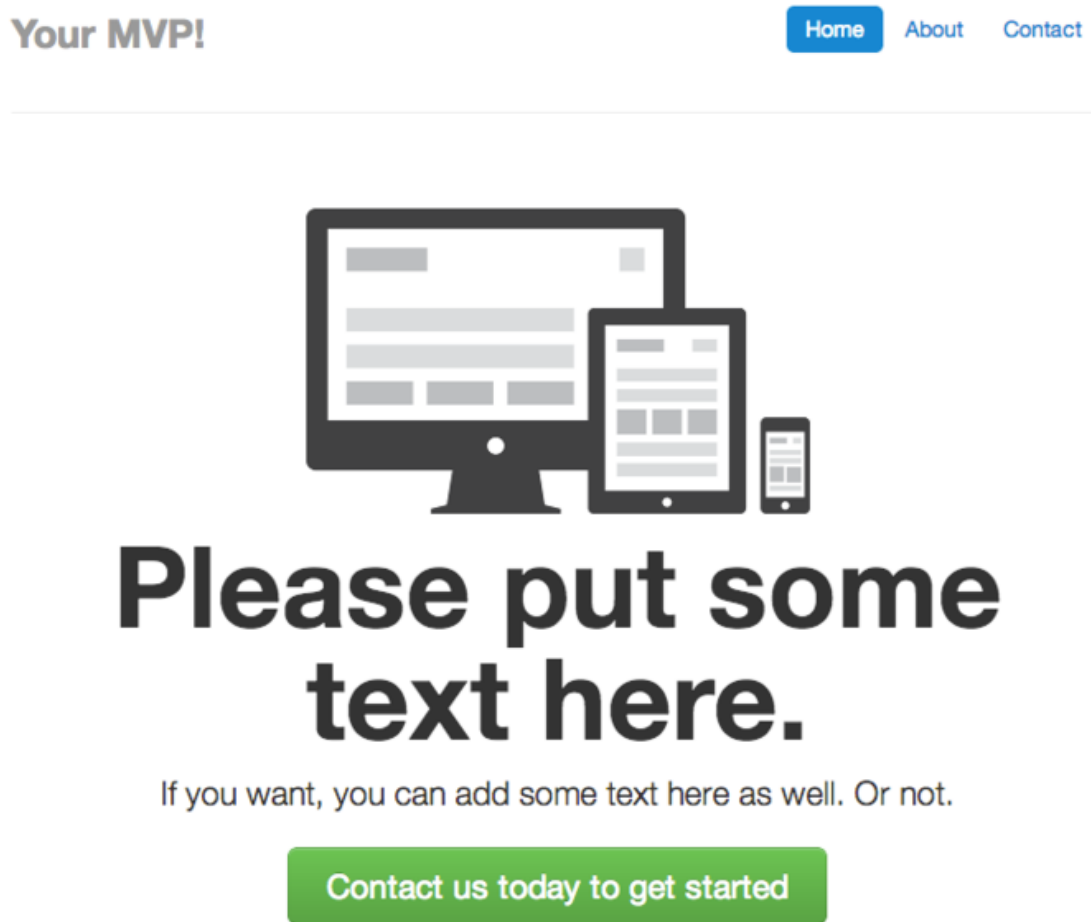


Figure 2.1: Initial look

It's a nice start, but admittedly not something that you could take to Y Combinator to get funded. However, by chapter 10 that same application will look like this:

And by the end of the course... well, you're just going to have to work your way through the course and find out.

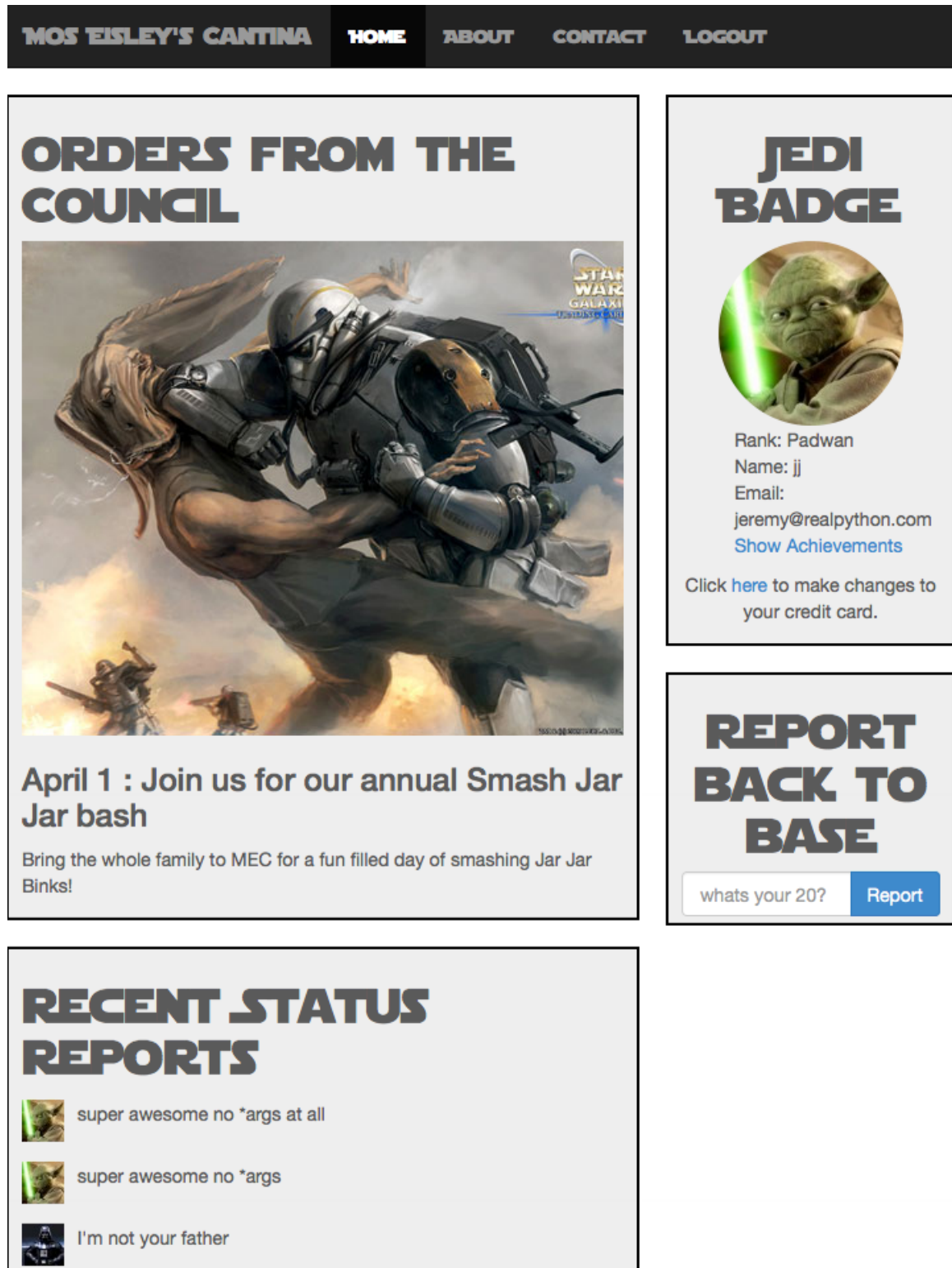


Figure 2.2: Chapter 10

Now let's build something **REAL!**

Chapter 3

Software Craftsmanship

Somewhere along the path to software development mastery every engineer comes to the realization that there is more to writing code than just producing the required functionality.

Novice engineers will generally stop as soon as the requirements are met, but that leaves a lot to be desired. Software development is a craft, and like any craft there are many facets to learn in order to master it. True craftsman insist on quality throughout all aspects of a product development process; it is never good enough to just deliver requirements. McDonald's delivers requirements, but a charcoal-grilled, juicy burger cooked to perfection on a Sunday afternoon and shared with friends... that's craftsmanship.

In this course we will talk about craftsmanship from the get go, because it's not something you can add in at the end. It's a way of thinking about software development that values:

- Not only working software, but *well-crafted software*;
- Moving beyond responding to change and focusing on *steadily adding value*; and,
- Creating *maintainable code* that is *well-tested* and *simple to understand*.

This chapter is about the value of software craftsmanship, and writing code that is elegant, easily maintainable and dare I say... beautiful. And while this may seem high-browed, flutey-tooty stuff, the path to achieving well-crafted software can be broken down into a number of simple techniques and practices.

We will start off with the most important of these practices and the first that any engineer looking to improve his or her craft should learn: **Automated Unit Testing**.

Automated unit testing

Automated unit testing is the foundation upon which good Software Craftsmanship practices are built. Since this is a course on Django development, and the third course in the [Real Python](#) series, we will start by taking a look at the Django MVP app from the last chapter of the previous course, [Web Development with Python](#) by Michael Herman.

One of the [first](#) things you should do when you start work on an existing project is to run the tests (if there are any of course). Further, since we have taken ownership of this project it's now our responsibility to ensure that the code is free from bugs, inconsistencies, or any other problems that could cause issues down the road. This is achieved primarily through unit testings. By adding unit tests to this application we will start to see how testing should be done and, more importantly, how it can make you a better software developer.

Before we get our hands dirty, let's go over the testing tools we have at our disposal with Django.

Behind the Scenes: Running unit tests with Django

There are several [classes](#) we can inherit from to make our testing easier in Django.

Provided test case classes

1. `unittest.TestCase()` - The default Python test case, which:
 - Has no association with Django, so there's nothing extra to load; because of this, it produces the fastest test case execution times.
 - Provides basic assertion functionality.
2. `django.test.SimpleTestCase()` - A very thin wrapper around `'unittest.TestCase()'`, which:
 - Asserts that a particular exception was raised.
 - Asserts on Form Fields - i.e., to check field validations.
 - Provides various HTML asserts, like `assert.contains` - which checks for an HTML fragment.
 - Loads Django settings and allows for loading custom Django settings.
3. `django.test.TestCase()` or `django.test.TransactionTestCase()` - which differ only by how they deal with the database layer.

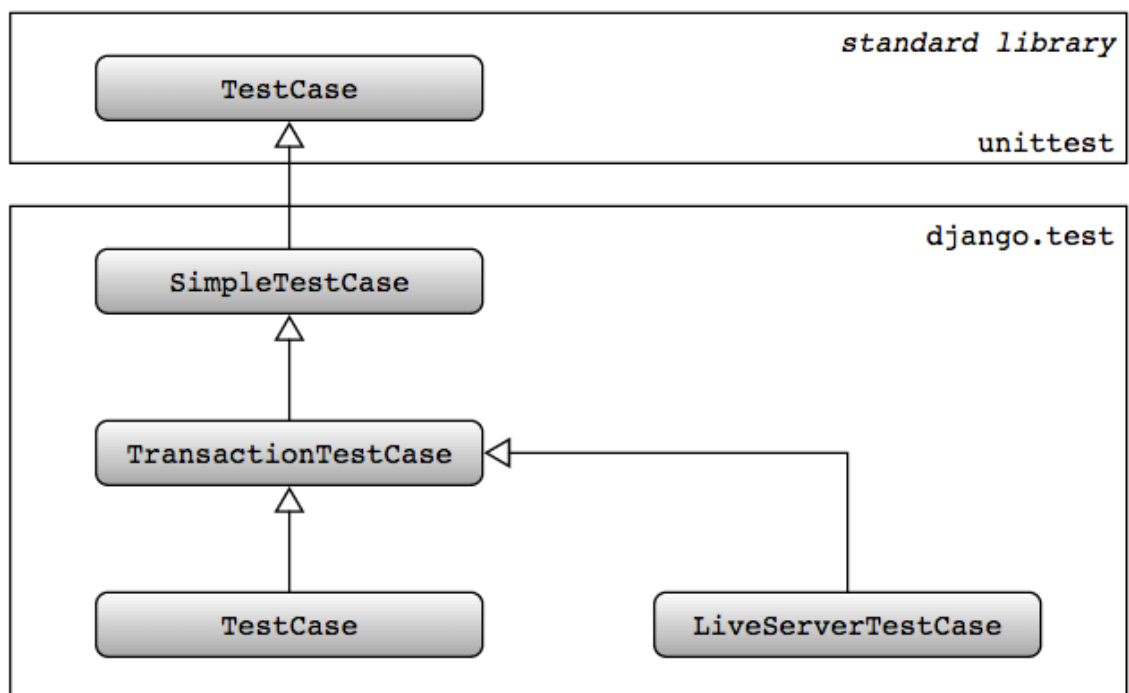


Figure 3.1: Django Unittest Classes Hierarchy

- Both:
 - Automatically load fixtures to load data prior to running tests.
 - Create a [Test Client](#) instance, `self.client`, which is useful for things like resolving URLs.
 - Provide a ton of Django-specific asserts.
- `TransactionTestCase()` allows for testing database transactions and automatically resets the database at the end of the test run by truncating all tables.
- `TestCase()` meanwhile wraps each test case in a transaction, which is then rolled back after each test.

Thus, the major difference is when the data is cleared out - after each test for `TestCase()` versus after the entire test run for `TransactionTestCase()`.

4. `django.test.LiveServerTest()` - Used mainly for GUI Testing with something like [Selenium](#), which
 - Does basically the same thing as `TransactionTestCase()`.
 - Also fires up an actual web server so you can perform GUI tests.

Which test base class should you use?

Keep in mind that, with each test base class, there's always a trade off between *speed* and *test complexity*. As a general rule testing becomes easier and less complex the more you rely on Django's internal functionality. However, there is a cost- speed.

So, which test class is best? Unfortunately, there is no easy answer to that. It all depends on the situation.

From a purely performance-oriented point of view, you should always use `unittest.TestCase()` and not load any Django functionality. However, that may make it very difficult/cumbersome to test certain parts of your application.

Any time you use one of the `django.test.*` classes you will be loading all the Django infrastructure and thus your test might be doing a lot more than what you bargained for. Take the following example:

```

1 def test_uses_index_html_template(self):
2     index = self.client.get('/')
3     self.assertTemplateUsed(index, "index.html")

```

It's only two lines of code, right? True. But behind the scenes there's a lot going on:

- The Django Test Client is fired up.
- The Django Router is then called from `self.client.get`.
- A Django request object is automatically created and passed to `self.client.get`.
- A Django response object is returned.
- Third party middleware is called.
- Application middleware is also called.
- Context Managers are called.

As you can see there is a lot going on here, we are testing multiple parts of the system. Tests that cover multiple parts of a system and verify the result when all these parts act in concert are generally referred to as an **Integration Test**.

By contrast a unit test by definition should test a small defined area of code (i.e. a unit of code), in isolation. Which in most cases translates to a single function.

To rewrite this test as a unit test we could inherit from `'django.test.SimpleTestCase()'` and then our test would look like this:

```

1 from django.test import RequestFactory
2
3 def test_uses_index_html_template_2(self):
4     #create a dummy request
5     request_factory = RequestFactory()
6     request = request_factory.get('/')
7     request.session = {} #make sure it has a session associated
8
9     #call the view function
10    resp = index(request)
11
12    #check the response
13    self.assertContains(resp, "<title>Your MVP</title>")

```

Obviously this test is a bit longer than two lines, but since we are creating a dummy request object and calling our view function directly, we no longer have so much going on behind the scenes. There is no router, there is no middleware; basically all the Django “magic” isn’t happening. This means we are testing our code in isolation (it’s a unit test), which has the following consequences:

- Tests will likely run faster, because there is less going on (again, behind the scenes).
- Tracking down errors will be significantly easier because only one thing is being tested.

- Writing tests can be harder because you may have to create dummy objects or stub out various parts of the Django system to get your tests to work.

NOTE: A Note on `assertTemplateUsed` Notice that in the above test we have changed the `assertTemplateUsed` to `assertContains`. This is because `assertTemplateUsed` does not work with the `RequestFactory()`. In fact, if you use the two together the `assertTemplateUsed` assertion will always pass, no matter what template name you pass in! So be careful not to use the two together!

Integration vs unit testing

Both integration and unit tests have their merits, and you can use both types together in the same test suite - so it's not all-or-nothing. But it is important to understand the differences of each approach so you can choose which option is best for you. There are some other considerations, specifically designing testable code and mocking, which we will touch upon later.

For now the important thing to understand is the difference between the two types and the amount of work that the Django testing framework is doing for you behind the scenes.

Testing Routes and Views

Setup

If you haven't gone through the second Real Python course, *Web Development with Python*, yet and you want to get a broad overview of several of the most popular web frameworks out there - like Flask, Django, and web2py - you really should have a look at it [here](#). *If you ask nicely, by emailing info@realpython.com, you can probably get a discount.*

However, if you just want to just get started on this course, you can grab a copy of the Django MVP application by using the supplied [repo](#) for this course. Click [here](#) to download the compressed file directly to your file system. Once downloaded, extract the files.

Unit Testing in Django

Django makes unit testing pretty straight-forward. Unit testing is generally accomplished by using the Django helper classes defined in `django.test.testcases`. These helper classes extend the standard Python `unittest` framework by providing a number of helper functions that make it much simpler to test Django applications.

If you think about how most Django applications are structured, they loosely follow the MVC (Model-View-Controller) model. But to make things confusing, a view in Django is actually a controller in MVC and a Django template is an MVC view. See the Django [docs](#) for more details. Either way, Django applications generally follow the standard activity patterns (as do most web frameworks):

1. Receive a web request.
2. Route that request to the appropriate Django view.
3. Perform some business logic, which could include accessing a datastore through a Django model.
4. Return an HTTP Response.

Thus, we want to test:

1. That requests are routed properly.
2. Appropriate responses are returned.
3. The underlying business logic.
4. Datastore access through a Django model.

Out of the box, Django gives us two standard ways to write unit tests - using the standard `unittest` style or the `DocTest` style.

SEE ALSO: Please consult the Python documentation for more info in both standard [unittest](#) and [doctests](#). Jump to [this](#) StackOverflow question to get more info on when to use `unittest` vs. `doctest`.

Both are valid and the choice is more a matter of preference than anything else. Since the standard `unittest` style is the most common, we will focus on that.

Built-in Django tests

Regardless of which you choose, the first thing you should do before you start creating your own tests is to verify that your Django environment is setup correctly. This can be done by running Django's built-in tests. If you have worked through the previous course, you should have a virtual environment setup and ready to go for the MVP app. If not, set one up. Or you can use a more powerful tool called [virtualenvwrapper](#), which you will find instructions for below.

WARNING: Stop here if you either don't know how to set up a basic virtual environment with [virtualenv](#) or if you don't know what it's used for. These are **strong** indicators that you need to go through the second Real Python course. Baby steps.

If you're continuing from the previous course, make sure you have your `virtualenv` activated and ready to go by running:

```
1 $ source myenv/bin/activate
```

If you're starting from the downloaded application:

1. Ensure you have [virtualenvwrapper](#) installed, which is a wrapper around `virtualenv` that makes it easier to use:

```
1 $ pip install virtualenvwrapper
```

2. Then set up the virtual environment with `virtualenvwrapper`:

```
1 $ cd django_project
2 $ mkvirtualenv chp4
3 $ pip install -r ./requirements.txt
```

This will read the *requirements.txt* file in the root directory of the *django_ecommerce* project and use pip to install all the dependencies listed in the *requirements.txt* file, which at this point should just be Django and Stripe.

NOTE: Using a *requirements.txt* file as part of a Python project is a great way to manage dependencies. It's a common convention that you need to adhere to for any Python project, not just Django. A *requirements.txt* file is a simple text file that lists one dependency per line of your project; pip will read this file and all the dependencies will then be downloaded and installed automatically.

After you have your virtual environment set up and all your dependencies installed, type `pip freeze > requirements.txt` to automatically create the *requirements.txt* file that can later be used to quickly install the project dependencies. Then check the file into git and share it with your team, and everybody will be using the same set of dependencies.

More information about *requirements.txt* files can be found on the [official docs page](#).

3. OPTIONAL: If for whatever reason, the *requirements.txt* is not found, then we need to set one up. No worries. It's good practice.

Start by installing the following dependencies:

```
1 $ pip install django==1.5.5
2 $ pip install stripe==1.9.2
```

Now just create the *requirements.txt* file in the root directory ("repo"):

```
1 $ pip freeze > requirements.txt
```

That's it.

4. Take a quick glance at the project structure, paying particular attention to the *tests.py* files. See anything that stands out. Well, there are no tests, except for the generic 2+2 test:

```
1 from django.test import TestCase
2
3
4 class SimpleTest(TestCase):
5     def test_basic_addition(self):
6         """
7         Tests that 1 + 1 always equals 2.
```



```

8         """
9         self.assertEqual(1 + 1, 2)

```

So, we need to first add tests to ensure that this current app works as expected without any bugs, otherwise we cannot confidently implement new features.

5. Once you have all your dependencies in place, let's run a quick set of tests to ensure Django is up and running correctly:

```

1 $ cd django_ecommerce
2 $ python manage.py test

```

I generally just `chmod +x manage.py`. Once you do that, you can just run:

```

1 $ ./manage.py test

```

Not a huge difference, but over the long-run that can save you hundreds of keystrokes.

NOTE: You could also add the alias `alias p="python"` to your `~/.bashrc` file so you can just run `p` instead of `python`: `p manage.py test`.

Nevertheless, after running the tests you should get some output like this:

```

1 Creating test database for alias 'default'...
2 .....
3 .....
4 .....S.....
5 .....
6 .....
7 .....X.....
8 .....
9 .....
10 .....
11 .....
12 -----
13 Ran 530 tests in 15.226s
14
15 OK (skipped=1, expected failures=1)
16 Destroying test database for alias 'default'...

```

Notice that with Django 1.5 there will always be one failure - (expected failures=1); that's expected, so just ignore it. If you get an output like this, then you know your system is correctly configured for Django. Now it's time to test our own app.

However, if you get an error about ‘ValueError: unknown local e: UTF-8’ this is the dreaded Mac [UTF-8 hassle](#). Depending on who you ask, it’s a minor annoyance in Mac’s default terminal to the worst thing that could ever happen. In truth, though, it’s easy to fix. From the terminal simply type the following:

```
1 $ export LC_ALL=en_US.UTF-8
2 $ export LANG=en_US.UTF-8
```

Feel free to use another language if you prefer. With that out of the way, your tests should pass just fine. Before moving on, though, add these two lines to your `.bash_profile` file so you don’t have to retype them each time you start your terminal.

NOTE With `virtualenvwrapper`, to deactivate you still use the `deactivate` command. When you’re ready to reactivate the `virtualenv` to work on your project again simply type `workon django_mvp_app`. You can also just type `workon` to view your current working virtual environments. Simple.

Testing Django Routing

Looking back at our Django MVP app, we can see it provides routing for a main page, admin page, `sign_in`, `sign_out`, `register`, `contact` and `edit`. These can be seen in `django_ecommerce/urls.py`:

```
1 urlpatterns = patterns('',
2     url(r'^admin/', include(admin.site.urls)),
3     url(r'^$', 'main.views.index', name='home'),
4     url(r'^pages/', include('django.contrib.flatpages.urls')),
5     url(r'^contact/', 'contact.views.contact', name='contact'),
6
7     # user registration/authentication
8     url(r'^sign_in$', views.sign_in, name='sign_in'),
9     url(r'^sign_out$', views.sign_out, name='sign_out'),
10    url(r'^register$', views.register, name='register'),
11    url(r'^edit$', views.edit, name='edit'),
12 )
```

Our first set of tests will ensure that this routing works correctly and that the appropriate views are called for the corresponding URLs. In effect we are answering the question, “Is my Django application wired up correctly”?

This may seem like something too simple to test, but keep in mind that Django’s routing functionality is based upon regular expressions, which are notoriously tricky and hard to understand.

NOTE: For more on regular expressions, be sure to check out the excellent chapter in the first Real Python course, Introduction to Python.

Furthermore, Django’s routing functionality follows a first-come-first-serve model, so it’s not at all uncommon to include a new url pattern that accidentally overwrites an old one. Once we have these tests in place and we run them after each change to the application, then we will be quickly notified if we make such a mistake, allowing us to quickly resolve the problem and move on with our work, confidently.

First test: resolve

Here is what our first test might look like:

main/tests.py

```
1 from django.test import TestCase
2 from django.core.urlresolvers import resolve
3 from .views import index
4
5
6 class MainPageTests(TestCase):
7
8     def test_root_resolves_to_main_view(self):
9         main_page = resolve('/')
10        self.assertEqual(main_page.func, index)
```

Here we are using Django’s own urlresolver ([docs](#)) to test that the / URL “resolves”, or maps, to the `main.views.index` function. This way we know that our routing is working correctly.

Want to test this in the shell to see exactly what happens?

```
1 $ ./manage.py shell
2 >>> from django.core.urlresolvers import resolve
3 >>> main_page = resolve('/')
4 >>> print main_page.func
5 <function index at 0x102270848>
```

This just shows that the / URL does in fact map back to the `main.views.index` function. You could also test the url name, which is called home - `url(r'^$', 'main.views.index', name='home'),:`

```
1 $ ./manage.py shell
2 >>> from django.core.urlresolvers import resolve
3 >>> main_page = resolve('/')
4 >>> print main_page.url_name
5 home
```

Perfect.

Now, run the tests:

```
1 $ ./manage.py test main
```

We should see the test pass...

```
1 Creating test database for alias 'default'...
2 .
3 -----
4 Ran 1 test in 0.633s
5
6 OK
```

Great!

You could test this in a slightly different way, using `reverse()`, which is also a `urlresolver`:

```
1 $ ./manage.py shell
2 >>> from django.core.urlresolvers import reverse
3 >>> url = reverse('home')
4 >>> print url
5 /
```

Can you tell what the difference is, though, between `resolve()` and `reverse()`? Check out the Django [docs](#) for more info.

Second test: status code

Now that we know our routing is working correctly let's ensure that we are returning the appropriate view. We can do this by using the `client.get()` function from `django.test.TestCase` ([docs](#)). In this scenario, we will send a request from a dummy web browser known as the [Test Client](#) then assert that the returned response is correct. More on the Test Client later.

A basic test for this might look like:

```
1 def test_returns_appropriate_html(self):
2     index = self.client.get('/')
3     self.assertEqual(index.status_code, 200)
```

NOTE: Note on Test Structure

In Django 1.5 the default place to store tests is in a file called *tests.py* inside the each application's directory - which is why these tests are in *main/tests.py*. For the remainder of this chapter we are going to stick to that standard and put all tests in *app_name/tests.py* files. In a later chapter we will discuss more on the advantages/disadvantages of this test structure. For now let's just focus on testing.

Test again:

```
1 Creating test database for alias 'default'...
2 ..
3 -----
4 Ran 2 tests in 0.759s
5
6 OK
```

Testing Templates and Views

The issue with the last test is that even if you're returning the incorrect HTML, it will still pass.

First test: template

It's better to verify the actual template used:

```
1 def test_uses_index_html_template(self):
2     index = self.client.get('/')
3     self.assertTemplateUsed(index, "index.html")
```

So, `assertTemplateUsed()` is one of those helper functions provided by Django, which just checks to see if a `HttpResponse` object gets generated by a specific template. Check out the Django [docs](#) for more info.

Second test: expected HTML

We can further increase our test coverage to verify that not only are we using the appropriate template but that the HTML being returned is correct as well. In other words, after the template engine has done its thing, do we get the expected HTML?

The test looks like this:

```
1 from django.shortcuts import render_to_response
2
3 def test_returns_exact_html(self):
4     index = self.client.get("/")
5     self.assertEqual(
6         index.content,
7         render_to_response("index.html").content
8     )
```

In the above test, the `render_to_response()` shortcut function is used to run our *index.html* template through the Django template engine and ensure that the response is the same had an end user called the `/` url.

Third test: expected HTML redux

The final thing to test for the `index` view is that it returns the appropriate html for a logged-in user. This is actually a bit trickier than it sounds, because the `index()` function actually performs three separate functions:

1. It checks the session to see if a user is logged in.
2. If the user is logged in, it queries the database to get that user's information.
3. Finally, it returns the appropriate HTTP Response.

Now is the time that we may want to argue for refactoring the code into multiple functions to make it easier to test. But for now let's assume we aren't going to refactor and we want to test the functionality as is. This means that really we are executing a **Integration Test** instead of a **Unit Test**, because we are now testing several parts of the application as opposed to just one individual unit.

NOTE: Notice how during this process we found an issue in our code - e.g., one function having too many responsibilities. This is a side effect of testing: You learn to write better code.

Forgetting about the distinction for the time being, let's look at how we might test this functionality:

1. Create a dummy session with the user entry that we need.
2. Ensure there is a user in the database, so our lookup works.
3. Verify that we get the appropriate response back.

Let's work backwards: ...

```
1 def test_index_handles_logged_in_user(self):
2
3     # test logic will go here
4
5     self.assertTemplateUsed(resp, 'user.html')
```

We have seen this before: See the first test in this section, `test_user_index_html_template()`.

Above we are just trying to verify that when there is a logged in user, we return the *user.html* template instead of the normal *index.html* template.

```
1 def test_index_handles_logged_in_user(self):
2     # create the user needed for user lookup from index page
3     from payments.models import User
4     user = User(
5         name='jj',
6         email='j@j.com',
7     )
8     user.save()
9
10    ...snipped code...
11
12    # verify the response returns the page for the logged in user
13    self.assertTemplateUsed(resp, 'user.html')
```

Here we are creating a user and storing that user in the database. This is more or less the same thing we do when a new user registers.

Take a breather Before continuing, be sure you understand exactly what happens when you run a Django unittest. Flip back to the last chapter if you need a quick refresher.

Do you understand why we need to create a user to test this function correctly if our `main.views.index()` function is working? Yes? Move on. No? Flip back and read about the `django.test.TestCase()`, focusing specifically on how it handles the database layer.

Ready?

Move on Want the answer? It's simple: It's because the `django.test.TestCase` class that we are inheriting from will clear out the database prior to each run. Make sense?

Also, you may not have noticed, but we are testing against the actual database. This is problematic, but there is a solution. Sort of. We will get to this later. But first, think about the request object that is passed into the view. Here is the view code we are testing:

```
1 def index(request):
2     uid = request.session.get('user')
3     if uid is None:
4         return render_to_response('index.html')
5     else:
6         return render_to_response(
7             'user.html',
8             {'user': User.objects.get(pk=uid)}
9         )
```

As you can see, the view depends upon a request object and a session. Usually these are managed by the web browser and will *just be there*, but in the context of unit testing they won't *just be there*. We need a way to make the test *think* there is a request object with a session attached.

And there is a technique in unit testing for doing this.

Mocks, fakes, test doubles, dummy objects, stubs There are actually many names for this technique: **mocking**, **faking**, using **dummy objects**, **test doubling**, **stubbing**, etc.

There are subtle difference between each of these, but the terminology just adds confusion. For the most part, they are all referring to the same thing - which we will define as: *Using a temporary, in-memory object to simulate a real object for the purposes of decreasing test complexity and increasing test execution speed.*

Visually you can think of it as:

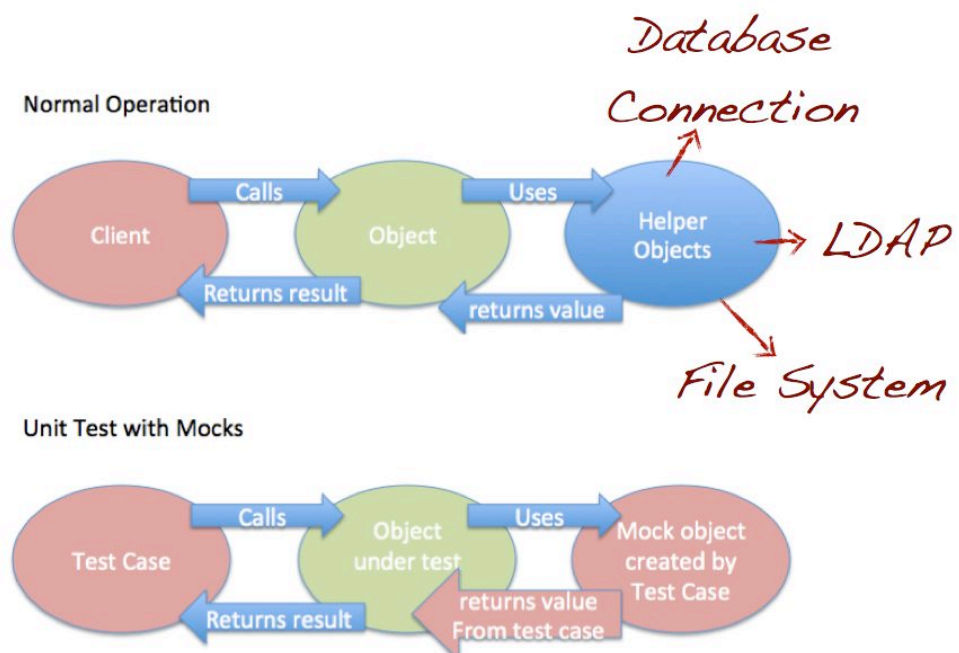


Figure 3.2: mocking example

As you can see from the illustration, a **mock** (or whatever you want to call it) simply takes the place of a real object allowing you to create tests that don't rely on external dependencies.

This is advantageous for two main reasons:

1. Without external dependencies your tests often run much faster; this is important because unit tests should be ran frequently (ideally after every code change).
2. Without external dependencies it's much simpler to manage the necessary state for a test, keeping the test simple and easy to understand.

Coming back to the request and its state - the user value stored in the session - we can mock out this external dependency by using Django's **built-in** `RequestFactory()`:

```
1 # create a dummy request
2 from django.test import RequestFactory
3 request_factory = RequestFactory()
4 request = request_factory.get('/')
5 request.session = {} # make sure it has an associated session
```

By creating the request mock, we can set the state (i.e. our session values) to whatever we want and simply write and execute unit tests to ensure that our view function correctly responds to the session state.

Step 1: Verify that we get the appropriate response back And with that we can complete our earlier unit test:

```
1 from payment.models import User
2 from django.test import RequestFactory
3
4 def test_index_handles_logged_in_user(self):
5     #create the user needed for user lookup from index page
6     from payments.models import User
7     user = User(
8         name='jj',
9         email='j@j.com',
10    )
11    user.save()
12
13    #create a Mock request object, so we can manipulate the session
14    request_factory = RequestFactory()
15    request = request_factory.get('/')

```

```

16     #create a session that appears to have a logged in user
17     request.session = {"user": "1"}
18
19     #request the index page
20     resp = index(request)
21
22     #verify it returns the page for the logged in user
23     self.assertEqual(
24         resp.content,
25         render_to_response('user.html', {'user': user}).content
26     )

```

Run the tests:

```

1 Creating test database for alias 'default'...
2 ....
3 -----
4 Ran 4 tests in 0.964s
5
6 OK
7 Destroying test database for alias 'default'...

```

Good.

Let's recap

Our `test_index_handles_logged_in_user()` test:

1. First creates a user in the database.
2. Then we mock the request object and set the session to have a user with an `id` of 1 (the user we just created). In other words, we are setting the (initial) state so we can test that the index view responds correctly when we have a logged-in user.
3. Then we call the index view function, passing in our mock request then verifying that it returns the appropriate HTML.

At this point we have the test completed for our main app. You may want to review the tests. Would you do anything different? Experiment. Challenge yourself. There's always better, more efficient ways of accomplishing a task. Learn how to fail. Struggle. Ask questions. Experience a few wins. Then learn how to succeed.

Before we moving to testing the models, we should have a look at refactoring some of our tests.

Refactoring our Tests

Refactoring, part 1

Looking back at the tests, there are some discrepancies, as some of the tests use the `RequestFactory()` mock while others don't. So, let's clean up/refactor the tests a bit so that all of them will use request mocks. Why? Simple: This should make them run faster and help keep them isolated (more on this below). However, the `test_root_resolves_to_main_view()` tests the routing so we should not mock out anything for that test. We really do want it to go through all the heavy Django machinery to ensure that it's properly wired up.

To clean up the tests, the first thing we should do is create a `setUp` method which will run prior to test execution. This method will create the mock so we don't have to recreate it for each of our tests:

```
1 @classmethod
2 def setUpClass(cls):
3     request_factory = RequestFactory()
4     cls.request = request_factory.get('/')
5     cls.request.session = {}
```

Notice the above method, `setUpClass()`, is a `@classmethod`. This means it only gets run once when the `MainPageTest` class is initially created, which is exactly what we want for our case. In other words, the mock is created when the class is initialized, and it gets used for each of the tests. See the Python [docs](#) for more info.

If we instead wanted the setup code to run before each test, we could write it as:

```
1 def setUp(self):
2     request_factory = RequestFactory()
3     self.request = request_factory.get('/')
4     self.request.session = {}
```

This will run prior to each test running. Use this when you need to isolate each test run.

After we have created our `setUp` method, we can then update our remaining test cases as follows:

```
1 from django.test import TestCase
2 from django.core.urlresolvers import resolve
3 from .views import index
4 from django.shortcuts import render_to_response
```

```

5 from payments.models import User
6 from django.test import RequestFactory
7
8
9 class MainPageTests(TestCase):
10
11     #####
12     ##### Setup #####
13     #####
14
15     @classmethod
16     def setUpClass(cls):
17         request_factory = RequestFactory()
18         cls.request = request_factory.get('/')
19         cls.request.session = {}
20
21     #####
22     ##### Testing routes #####
23     #####
24
25     def test_root_resolves_to_main_view(self):
26         main_page = resolve('/')
27         self.assertEqual(main_page.func, index)
28
29     def test_returns_appropriate_html_response_code(self):
30         resp = index(self.request)
31         self.assertEqual(resp.status_code, 200)
32
33     #####
34     ##### Testing templates and views #####
35     #####
36
37     def test_returns_exact_html(self):
38         resp = index(self.request)
39         self.assertEqual(
40             resp.content,
41             render_to_response("index.html").content
42         )
43
44     def test_index_handles_logged_in_user(self):

```

```

45     # Create the user needed for user lookup from index page
46     user = User(
47         name='jj',
48         email='j@j.com',
49     )
50     user.save()
51
52     # Create a session that appears to have a logged in user
53     self.request.session = {"user": "1"}
54
55     # Request the index page
56     resp = index(self.request)
57
58     #ensure we return the state of the session back to normal so
59     #we don't affect other tests
60     self.request.session = {}
61
62     #verify it returns the page for the logged in user
63     expectedHtml = render_to_response('user.html', {'user':
64         user}).content
65     self.assertEqual(resp.content, expectedHtml)

```

Looks pretty much the same.

But all the tests, with the notable exception of `test_root_resolves_to_main_view()`, are now isolated from any Django URL routing or middleware, so the tests should run faster be more robust and easier to debug should there be any problems.

Sanity check

Run the test!

```

1 Creating test database for alias 'default'...
2 ....
3 -----
4 Ran 4 tests in 0.667s
5
6 OK
7 Destroying test database for alias 'default'...

```

Refactoring, part 2

We still have one test, though, `test_index_handles_logged_in_user()` that is dependent on the database and, thus, needs to be refactored. Why? As a general rule, you do not want to test the database with a unit test. That would be an integration test. Mocks are perfect for handling calls to the database.

First, install the [mock library](#):

```
1 $ pip install mock
```

Our test then might look like this:

```
1 import mock
2 from payments.models import User
3
4 def test_index_handles_logged_in_user(self):
5     # Create the user needed for user lookup from index page
6     # Note that we are not saving to the database
7     user = User(
8         name='jj',
9         email='j@j.com',
10    )
11
12    # Create a session that appears to have a logged in user
13    self.request.session = {"user": "1"}
14
15    with mock.patch('main.views.User') as user_mock:
16
17        # Tell the mock what to do when called
18        config = {'get.return_value': user}
19        user_mock.objects.configure_mock(**config)
20
21        # Run the test
22        resp = index(self.request)
23
24        # Ensure that we return the state of the session back to
25        # normal
26        self.request.session = {}
27
28        expectedHtml = render_to_response(
29            'user.html', {'user': user}).content
```

29

```
self.assertEqual(resp.content, expectedHtml)
```

Test it again. All four should still pass.

What's happening here?

1. with `mock.path('main.views.User')` as `user_mock` says, “When we come across a `User` object in the `main.views` module, call our mock object instead of the actual user object.”
2. Next we call `user_mock.objects.configure_mock` and pass in our config dictionary, which says, “when `get` is called on our `User.objects` mock just return our dummy user that we created at the top of the function.
3. Finally we just ran our test as normal and asserted that it did the correct thing.

Now our view method gets the value it needs from the database without ever actually touching the database.

Since we now have no reliance on the Django testing framework, we actually could inherit directly from `unittest.TestCase()`. This would further isolate our tests from Django, which should speed them up.

Because of this, many Python developers believe you should mock out everything. Is that the best choice, though? Probably not. But we'll talk more about why in the next chapter when we test our models. Regardless, the mock framework is extremely powerful, and it can be a very useful tool to have in your Django unit testing tool belt. You can learn more about the mock library [here](#).

Now, let's move on to testing models.

Testing Models

In the previous section we used the Mock library to mock out the Django ORM so we could test a Model without actually hitting the database. This can make tests faster, but it's actually *NOT* a good approach to testing models. *Let's face it, ORM or not, your models are tied to a database - and are intended to work with a database. So testing them without a database can often lead to a false sense of security.*

The vast majority of model-related errors happen within the actual database. Generally database errors fall into one of the four following categories:

- Migration: Working off an out of date database.
- Datatype: Trying to store a string in a number column.
- Referential / integrity Constraints: Storing two rows with the same id.
- Retrieval: Using incorrect table JOINS.

All of these issues won't occur if you're mocking out your database, because you are effectively cutting out the database from your tests where all these issues originate from. Hence, testing models with mocks provides a false sense of security, letting these issues slip through the cracks.

This does not mean that you should never use mocks when testing models; just be very careful you're not convincing yourself that you have fully tested your model when you haven't really tested it at all.

That said, let's look at several techniques for safely testing Django Models.

WARNING: It's worth noting that many developers do not run their tests in development with the same database engine used in their production environment. This can also be a source of errors slipping through the cracks. In a later chapter, we'll look at using [Travis CI](#) to automate testing after each commit including how to run the tests on multiple databases.

Technique #1 - Just don't test them

Don't laugh. It's not a joke. After all, models are simply just a collection of fields that rely on standard Django functionality. We shouldn't test standard Django functionality because we can always run `manage.py test` and all the tests for standard Django functionality is tested for us. If the code is built-in to Django do not test it. This includes the fields on a model.

Specifically, look at the `payments.models.User` model:

```

1 from django.db import models
2 from django.contrib.auth.models import AbstractBaseUser
3
4
5 class User(AbstractBaseUser):
6     name = models.CharField(max_length=255)
7     email = models.CharField(max_length=255, unique=True)
8     # password field defined in base class
9     last_4_digits = models.CharField(max_length=4, blank=True,
10                                     null=True)
11     stripe_id = models.CharField(max_length=255)
12     created_at = models.DateTimeField(auto_now_add=True)
13     updated_at = models.DateTimeField(auto_now=True)
14
15     USERNAME_FIELD = 'email'
16
17     def __str__(self):
18         return self.email

```

What do we really need to test here?

Many tutorials have you writing tests like this:

```

1 from django.test import TestCase
2 from .models import User
3
4
5 class UserModelTest(TestCase):
6
7     def test_user_creation(self):
8         User(email = "j@j.com", name='test user').save()
9
10        users_in_db = User.objects.all()
11        self.assertEqual(users_in_db.count(), 1)
12
13        user_from_db = users_in_db[0]
14        self.assertEqual(user_from_db.email, "j@j.com")
15        self.assertEqual(user_from_db.name, "test user")

```

Congratulations. You have verified that the Django ORM can indeed store a model correctly. But we already knew that, didn't we? What's the point then? Not much, actually. Remember: *We do not need to test Django's inherent functionality.*

What about custom functionality?

Technique #2 - Create data on demand

Rather than spend a bunch of time testing stuff we don't really need to test, try to follow this rule: *Only test custom functionality that you created for models.*

Is there any custom functionality in our User model? Not much.

Arguably, we may want to test that our model uses an email address for the USERNAME_FIELD just to make sure that another developer doesn't change that. We could also test that the user prints out with the associated email.

To make things a bit more interesting, let's add a custom function to our model.

We can add a `find_user()` function. This will replace calls to `User.objects.get(pk)` like in the `main.views.index()` function:

```
1 def index(request):
2     uid = request.session.get('user')
3     if uid is None:
4         return render_to_response('index.html')
5     else:
6         return render_to_response(
7             'user.html',
8             {'user': User.objects.get(pk=uid)}
9         )
```

Isn't that the same thing? Not exactly. In the above code the view would have to know about the implementation details in our model, which breaks encapsulation. In reality, for small projects this doesn't really matter, but in large projects it can make the code difficult to maintain.

Imagine you had 40 different views, and each one uses your User model in a slightly different way, calling the ORM directly. Do you see the problem? *In that situation it's very difficult to make changes to your User model, because you don't know how other objects in the system are using it.*

Thus, it's a much better practice to encapsulate all that functionality within the User model and as a rule everything should only interact with the User model through its API, making it much easier to maintain.

Update the model

Without further ado, here is the change to our payments *models.py* file:

```
1 from django.db import models
2 from django.contrib.auth.models import AbstractBaseUser
3
4
5 class User(AbstractBaseUser):
6     name = models.CharField(max_length=255)
7     email = models.CharField(max_length=255, unique=True)
8     #password field defined in base class
9     last_4_digits = models.CharField(max_length=4, blank=True,
10                                     null=True)
11     stripe_id = models.CharField(max_length=255)
12     created_at = models.DateTimeField(auto_now_add=True)
13     updated_at = models.DateTimeField(auto_now=True)
14
15     USERNAME_FIELD = 'email'
16
17     @classmethod
18     def get_by_id(cls, uid):
19         return User.objects.get(pk=uid)
20
21     def __str__(self):
22         return self.email
```

Notice the `@classmethod` at the bottom of the code listing. We use a `@classmethod` because the method itself is stateless, and when we are calling it we don't have to create the `User` object, rather we can just call `User.get_by_id`.

Test the model

Now we can write some tests for our model. Create a *tests.py* file in the “payments” folder and add the following code:

```
1 from django.test import TestCase
2 from payments.models import User
3
4
5 class UserModelTest(TestCase):
```

```

6
7     @classmethod
8     def setUpClass(cls):
9         cls.test_user = User(email="j@j.com", name='test user')
10        cls.test_user.save()
11
12    def test_user_to_string_print_email(self):
13        self.assertEqual(str(self.test_user), "j@j.com")
14
15    def test_get_by_id(self):
16        self.assertEqual(User.get_by_id(1), self.test_user)

```

Test!

```

1 Creating test database for alias 'default'...
2 ..
3 -----
4 Ran 2 tests in 0.003s
5
6 OK
7 Destroying test database for alias 'default'...

```

What's happening here? In these tests, we don't use any mocks; we just create the User object in our setUpClass() function and save it to the database. **This is our second technique for testing models in action: Just create the data as needed..**

WARNING: Be careful with the setUpClass(). While using it in the above example can speed up our test execution - because we only have to create one user - it does cause us to share state between our tests. In other words, all of our tests are using the same User object, and if one test were to modify the User object, it could cause the other tests to fail in unexpected ways. This can lead to VERY difficult debugging sessions because tests could seemingly fail for no apparent reason. A safer strategy, albeit a slower one, would be to create the user in the setUp() method, which is run before each test run. You may also want to look at the tearDown() method and use it to delete the User model after each test is run if you decide to go the safer, latter route. As with most things in development its a trade-off. If in doubt though I would go with the safer option and just be a little more patient.

So, these tests provide some reassurance that our database is in working order because we are “round-tripping” - writing to and reading from - the database. If we have some silly configuration issue with our database, this will catch it, in other words.

NOTE: In effect, testing that the database appears sane is a side effect of our test here - which again means, by definition, these are not unit tests - they are integration tests. But for testing models the majority of your tests should really be integration tests.

Update the view and the associated test

Now that we have added the `get_by_id()` function update the `main.views.index()` function accordingly.

```
1 def index(request):
2     uid = request.session.get('user')
3     if uid is None:
4         return render_to_response('index.html')
5     else:
6         return render_to_response(
7             'user.html', {'user': User.get_by_id(uid)}
8         )
```

Along with the test:

```
1 import mock
2
3 def test_index_handles_logged_in_user(self):
4     # Create a session that appears to have a logged in user
5     self.request.session = {"user": "1"}
6
7     with mock.patch('main.views.User') as user_mock:
8
9         # Tell the mock what to do when called
10        config = {'get_by_id.return_value': mock.Mock()}
11        user_mock.configure_mock(**config)
12
13        # Run the test
14        resp = index(self.request)
15
16        # Ensure we return the state of the session back to normal
```

```

17         self.request.session = {}
18
19         expected_html = render_to_response(
20             'user.html', {'user': user_mock.get_by_id(1)}
21         )
22         self.assertEqual(resp.content, expected_html.content)

```

This small change actually simplifies our tests. Before we had to mock out the ORM - i.e. `User.objects`, and now we just mock the model.

This makes a lot more sense as well, because previously we were mocking the ORM when we were testing a view. That should raise a red flag. We shouldn't have to care about the ORM when we are testing views; after all, that's why we have models.

Technique #3 - Use fixtures

It should be mentioned that Django provides built-in functionality for automatically loading and populating model data: **fixtures**.

The idea is you can create a data file in such formats as XML, YAML or JSON and then ask Django to load that data into your models. This process is described in the official [documentation](#) and is sometimes used to load test data for unit testing.

I'll come right out and say it: I don't like this approach. There are several reasons why:

- It requires you to store data in a separate file, creating an additional place to look if you're debugging test errors.
- It can be difficult to keep the file up-to-date as your model changes.
- It's not always obvious what data in your test fixture applies to which test case.

Slow performance is often given as another reason why developers don't like it as well, but that can generally be rectified by using a SQLite database.

Our example MVP application is already using a SQLite database, but if your application is using some other SQL database engine, and you want your tests to run in SQLite so they will run faster, add the following line to your `settings.py` file. Make sure it comes after your actual database definition:

```

1 import sys
2 # Covers regular testing and django-coverage
3 if 'test' in sys.argv or 'test_coverage' in sys.argv:
4     DATABASES['default']['ENGINE'] = 'django.db.backends.sqlite3'

```

Now anytime you run Django tests, Django will use a SQLite database so you shouldn't notice much of a performance hit at all. Awesome!

NOTE: Do keep in mind the earlier warning about testing against a different database engine in development than production. We'll come back to this and show you how you can have your cake and eat it to! :)

Even though out of the box fixtures are difficult, there are libraries that make using fixtures much more straight-forward.

Fixture libraries

[Model Mommy](#) and [django-dynamic-fixtures](#) are both popular libraries.

These fixture libraries are actually closer to the second technique, *Create data on Demand*, than they are to the current technique. Why? Because you don't use a separate data file. Rather, these type of libraries create models for you with random data.

The school of thought that most of these libraries subscribe to is that testing with static data is bad. However, these arguments are far from convincing. That said, these libraries do make it super easy to generate test data, and if you're willing to add another library to your *requirements.txt* file then you can pull them in and use them.

Check out these write-ups on Django fixture libraries for more info:

- [ModelMommy writeup](#)
- [Django-dynamic-fixtures writeup](#)

Have a look at those articles to get started if you so choose. I prefer just using the models directly to create the data that I need for two reasons:

1. I don't really buy into the whole "static test data is bad".
2. If it's cumbersome to create test data with your models, maybe it's a code smell telling you to make your models easier to use. Or in more general terms, one of the most valuable aspects of unit testing is its ability to drive good design. Think of unit tests not only as functional tests of your code, but also as usability tests of your API. If it's hard to use, it's time to refactor.

Of course you're free to use these libraries if you like. I won't give you too much of a hard time about it.

Testing Forms

We have a number of forms in our application, so let's go ahead and test those as well.

`payments.forms` has several forms in there. Let's start simple with the `SignInForm`. To refresh your memory, it looks like this:

```
1 from django import forms
2 from django.core.exceptions import NON_FIELD_ERRORS
3
4 class SignInForm(PaymentForm):
5     email = forms.EmailField(required=True)
6     password = forms.CharField(
7         required=True,
8         widget=forms.PasswordInput(render_value=False)
9     )
```

When it comes to forms in Django, they generally have the following lifecycle:

1. A form is generally instantiated by a view.
2. Some data is populated in the form.
3. Data is validated against form rules.
4. Data is then cleaned and passed on.

We want to test by first populating data in the form and then ensuring that it validates correctly.

Mixins

Because most form validation is very similar, let's create a mixin to help us out. If you're unfamiliar with mixins [StackOverflow](#) can help.

Add the following code to `payments.tests`:

```
1 class FormTesterMixin():
2
3     def assertFormError(self, form_cls, expected_error_name,
4                         expected_error_msg, data):
5
6         from pprint import pformat
7         test_form = form_cls(data=data)
```

```

8         #if we get an error then the form should not be valid
9         self.assertFalse(test_form.is_valid())
10
11         self.assertEqual(
12             test_form.errors[expected_error_name],
13             expected_error_msg,
14             msg="Expected {} : Actual {} : using data {}".format(
15                 test_form.errors[expected_error_name],
16                 expected_error_msg, pformat(data)
17             )
18         )

```

This guy makes it super simple to validate a form. Just pass in the class of the form, the name of the field expected to have an error, the expected error message, and the data to initialize the form. Then, this little function will do all the appropriate validation and provide a helpful error message that not only tells you the failure - but what data triggered the failure.

It's important to know what data triggered the failure because we are going to use a lot of data combinations to validate our forms. This way it becomes easier to debug the test failures.

Test

Here is an example of how you would use the mixin:

```

1 from payments.forms import SigninForm
2 import unittest
3
4
5 class FormTests(unittest.TestCase, FormTesterMixin):
6
7     def test_signin_form_data_validation_for_invalid_data(self):
8         invalid_data_list = [
9             {'data': {'email': 'j@j.com'},
10              'error': ('password', [u'This field is required.'])},
11             {'data': {'password': '1234'},
12              'error': ('email', [u'This field is required.'])}
13         ]
14
15         for invalid_data in invalid_data_list:
16             self.assertFormError(SigninForm,
17                                  invalid_data['error'][0],

```

```
18         invalid_data['error'][1],
19         invalid_data["data"])
```

Let's test: `./manage.py test payments`

```
1 Creating test database for alias 'default'...
2 ...
3 -----
4 Ran 3 tests in 0.004s
5
6 OK
```

1. We first use [multiple inheritance](#) to inherit from the Python standard `unittest.TestCase` as well as the helpful `FormTesterMixin`.
2. Then in the `test_signin_form_data_validation_for_invalid_data()` test, we create an array of test data called `invalid_data_list`. Each `invalid_data` item in `invalid_data_list` is a dictionary with two items: the `data` item, which is the data used to load the form, and the `error` item, which is a set of `field_names` and then the associated `error_messages`.

With this setup we can quickly loop through many different data combinations and check that each field in our form is validating correctly.

Why is form validation so important?

Validation

Form validations are one of those things that developers tend to breeze through or ignore because they are so simple to do. But for many applications they are quite critical (I'm looking at you banking and insurance industries). QA teams seek out form validation errors like a politician looking for donations.

What's more, since the purpose of validation is to keep corrupt data out of the database, it's especially important to include a number of detailed tests. Think about edge cases.

Cleaning

The other thing that forms do is clean data. If you have custom data cleaning code, you should test that as well. Our `UserForm` does, so let's test that, starting with the validation then moving on to cleaning.

Please note: you'll have to inherit from SimpleTestCase to get the assertRaisesMessage function

Test Form Validation

```
1 from payments.forms import UserForm
2 from django import forms
3
4 def test_user_form_passwords_match(self):
5     form = UserForm(
6         {
7             'name': 'jj',
8             'email': 'j@j.com',
9             'password': '1234',
10            'ver_password': '1234',
11            'last_4_digits': '3333',
12            'stripe_token': '1'}
13    )
14    # Is the data valid? -- if not print out the errors
15    self.assertTrue(form.is_valid(), form.errors)
16
17    #this will throw an error if the form doesn't clean
18    #correctly
19    self.assertIsNotNone(form.clean())
20
21 def test_user_form_passwords_dont_match_throws_error(self):
22     form = UserForm(
23         {
24             'name': 'jj',
25             'email': 'j@j.com',
26             'password': '234',
27             'ver_password': '1234', # bad password
28             'last_4_digits': '3333',
29             'stripe_token': '1'
30         }
31    )
32
33    # Is the data valid?
34    self.assertFalse(form.is_valid())
```

```

35         self.assertRaisesMessage(forms.ValidationError,
36                                 "Passwords do not match",
37                                 form.clean)

```

Run the tests!

```

1 Creating test database for alias 'default'...
2 .....
3 -----
4 Ran 5 tests in 0.006s
5
6 OK
7 Destroying test database for alias 'default'...

```

That's it for now.

Testing Summary

This chapter provided a solid overview of unit testing for Django applications.

1. Part 1: Tested our Django url routing, templates and views using unit tests.
2. Part 2: Refactored those tests with Mocks to simply and further isolate the tests.
3. Part 3: Tested our models using the Django ORM by creating data on demand
4. Part 4: Tested our forms using mixins.

Summary

We covered quite a bit in this chapter, so hopefully this table will help summarize everything:

Testing Technique	When to use
Unit Testing	Early and often to test code you write
Integration Testing	Test integration between components
GUI Testing	Test end to end from the website
Mocking	to make unit tests more independent
Fixtures	load necessary data for a test

Testing Technique	Avantages
Unit Testing	test individual pieces of code, fast

Testing Technique	Advantages
Integration Testing	simple to write, test many things at one
GUI Testing	testing final product soup to nuts, test browser behavior
Mocking	separate concerns and test one thing at a time
Fixtures	quick and easy data management

Testing Technique	Disadvantages
Unit Testing	may require mocking
Integration Testing	test many things at once, slow
GUI Testing	can be slow and tests easily break
Mocking	complicated to write
Fixtures	sometimes difficult to know where data is coming from

This should provide you with the background you need to start unit testing your own applications. While it may seem strange at first to write unit tests alongside the code, keep at it and before long you should be reaping the benefits of unit testing in terms of less defects, ease of re-factoring, and having confidence in your code. If you're still not sure what to test at this point, I will leave you with this quote:

In god we trust. Everything else we test.

-anonymous

Exercises

Most chapters have exercises at the end. They are here to give you extra practice on the concepts covered in the chapter. Working through them will improve your mastery of the subjects. If you get stuck, answers can be found in *Appendix A*.

1. Our URL routing testing example only tested one route. Write tests to test the other routes. Where would you put the test to verify the `pages/` route? Do you need to do anything special to test the admin routes?
2. Write a simple test to verify the functionality of the `sign_out` view. Do you recall how to handle the session?
3. Write a test for `contact.models`. What do you really need to test? Do you need to use the database backend?

4. QA teams are particularly keen on 'boundary checking'. Research what it is, if you are not familiar with it, then write some unit tests for the CardForm from the payments app to ensure that boundary checking is working correctly.

Chapter 4

Test Driven Development

Test Driven Development (TDD) is a software development practice that puts a serious emphasis on writing automated tests as, or immediately before, writing code. The tests not only serve to verify the correctness of the code being written but they also help to evolve the design and overall architecture. This generally leads to high quality, simpler, easier to understand code, which are all goals of Software Craftsmanship.

Programmer (noun): “An organism that converts caffeine or alcohol into code.”

~Standard Definition

While the above definition from [Uncyclopedia](#) is somewhat funny and true it doesn't tell the whole story. In particular it's missing the part about half-working, buggy code that takes a bit of hammering to make it work right. In other words, programmers are humans, and we often make mistakes. We don't fully think through the various use cases a problem presents, or we forget some silly detail which can lead to some strange edge case bug that only turns up eight years later and costs the company \$465 million dollars in trading losses in about 45 minutes. It [happened](#).

Wouldn't it be nice if there were some sort of tool that would alert a programmer when the code isn't going to do what you think it's going to do? Well, there is.

It's called TDD.

How can TDD do that? It all comes down to timing. If you write a test to ensure a certain functionality is working, then immediately write the code to make the test pass, you have a pretty good idea that the code does what you think it does. *TDD is about rapid feedback*. And that rapid feedback helps developers who are not just machines (e.g., all of us) deliver great code.

The TDD Workflow

Now that we have covered how to write unit tests for Django apps, we can talk about TDD, which tells us **WHEN** to write unit tests. TDD describes a particular workflow for a developer to follow when writing code. The workflow is depicted below:

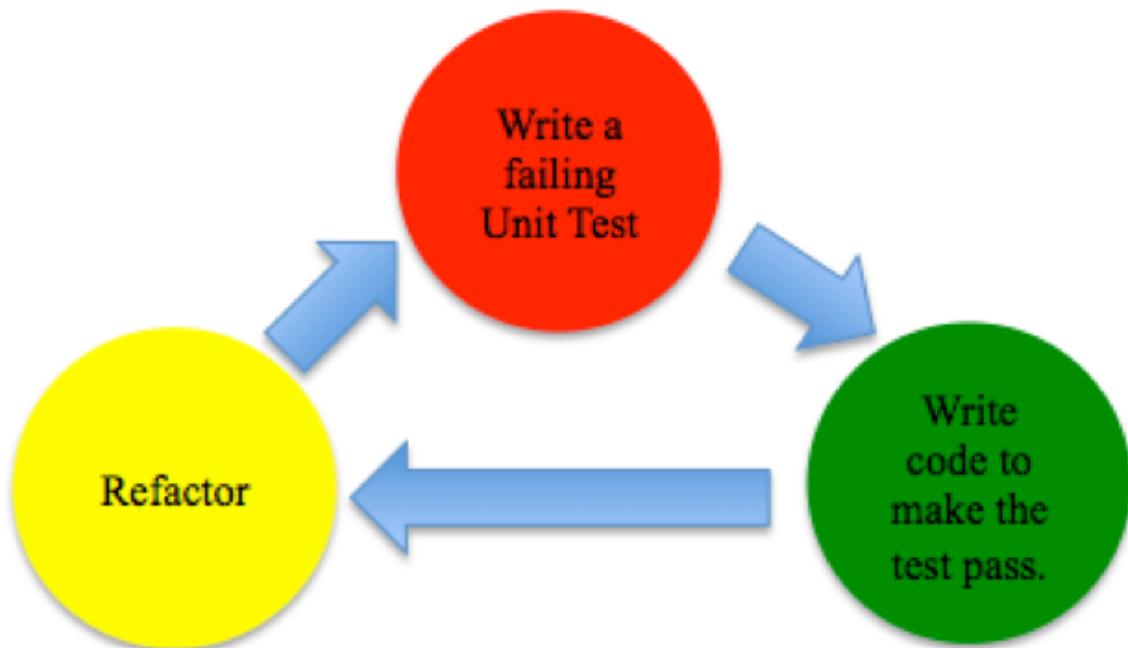


Figure 4.1: TDD workflow

1. Red Light

The above workflow starts with writing a test **before** you write any code. The test will of course fail because the code that it's supposed to verify doesn't exist yet. This may seem like a waste of time, writing a test before you even have code to test, but remember TDD is as much about design as it is about testing and writing a test before you write any code is a good way to think about how you expect your code to be used.

You could think about this in any number of ways, but a common checklist of simple considerations might look like this:

- Should it be a class method or an instance method?
- What parameters should it take?

- How do I know the function was successful?
- What should the function return?
- Do I need to manage any state?

These are just a few examples, but the idea is to think through how the code is going to be used before you start writing the code.

2. Green Light

Once you have a basic test written, write **just enough** code to make it pass. Just enough is important here. You may have heard of the term *YAGNI* - *You Ain't Going to Need It*. Words to live by when practicing TDD.

Since we are writing tests for everything, it's trivial to go back and update/extend some functionality later if we decide that we indeed are going to need it. But often times you won't. So don't write anything unless you really need it.

3. Yellow Light

Getting the test to pass isn't enough. After the test passes, we should review our code and look at ways we can improve it. There are entire books devoted to re-factoring, so I won't go into too much detail here, but just think about ways you can make the code easier to maintain, more reusable, cleaner, and simpler.

Focus on those things and you can't go wrong. So re-factor the code, make it beautiful, and then write another test to further verify the functionality or to describe new functionality.

4. Wash, Rinse, Repeat...

That's the basic process; we just follow it over and over and every few minutes this process will produce code that's well tested, thoughtfully designed and re-factored. This *should* avoid costing your company \$465 million dollars in trading losses.

Implementing TDD with Existing Code

If you are starting a new project with TDD, that's the simple case. **Just Do It!** For most people reading this, though, odds are you're somewhere half-way through a project with ever-looming deadlines, possibly some large quality issues, and a desire to get started with this whole TDD thing. Well fear not, you don't have to wait for your next project to get started with TDD. You can start applying TDD to existing code, and in this section you're going to learn just how to do that.

Before we jump into the code let's first talk briefly about where to start with TDD. There are three places where it really makes sense to kickstart TDD in an existing project:

1. **New Functionality** - Anytime you are writing new functionality for an ongoing project, write it using TDD.
2. **Defects** - When defects are raised, write at least two unit tests. One that passes and verifies the existing desired functionality and one that reproduces the bug, and thus fails. Now re-factor the code until both tests pass and continue on with the TDD cycle.
3. **Re-factoring** - There comes a time in every project when it just becomes apparent that the 600 line method that no one wants to own up to needs re-factoring. Start with some unit tests to guide your re-factoring, and then re-factor. We will explore this case in more detail using our existing system.

With that in mind, let's look at our running code example.

Django Ecommerce Project

To refresh your memory the application we have been working on is a simple Django application that provides a registration function, which is integrated with Stripe to take payments, as well as the basic login/logout functionality.

If you've been following along up to this point (and doing all the exercises) your application should be the same as what we are about to dissect in further detail. If you haven't or you just want to make sure you are looking at the same application you can get the correct version of the code from the git repo.

Registration Functionality

For this application, when a user registers, his or her credit card information will be passed to [Stripe](#), a third party payment processor, for processing. This logic is handled in the *payments/views.py* file. The `register()` function basically grabs all the information from the user, checks it against Stripe and stores it in the database.

Let's look at that function now. It's called `payments.views.register`:

```
1 def register(request):
2     user = None
3     if request.method == 'POST':
4         form = UserForm(request.POST)
5         if form.is_valid():
6
7             #update based on your billing method (subscription vs
              one time)
8             customer = stripe.Customer.create(
9                 email=form.cleaned_data['email'],
10                description=form.cleaned_data['name'],
11                card=form.cleaned_data['stripe_token'],
12                plan="gold",
13            )
14            # customer = stripe.Charge.create(
15            #     description=form.cleaned_data['email'],
16            #     card=form.cleaned_data['stripe_token'],
17            #     amount="5000",
18            #     currency="usd"
19            # )
20
```

```

21         user = User(
22             name=form.cleaned_data['name'],
23             email=form.cleaned_data['email'],
24             last_4_digits=form.cleaned_data['last_4_digits'],
25             stripe_id=customer.id,
26         )
27
28         #ensure encrypted password
29         user.set_password(form.cleaned_data['password'])
30
31         try:
32             user.save()
33         except IntegrityError:
34             form.addError(user.email + ' is already a member')
35         else:
36             request.session['user'] = user.pk
37             return HttpResponseRedirect('/')
38
39     else:
40         form = UserForm()
41
42     return render_to_response(
43         'register.html',
44         {
45             'form': form,
46             'months': range(1, 12),
47             'publishable': settings.STRIPE_PUBLISHABLE,
48             'soon': soon(),
49             'user': user,
50             'years': range(2011, 2036),
51         },
52         context_instance=RequestContext(request)
53     )

```

It is by no means a horrible function (this application is small enough that there aren't many bad functions), but it is the largest function in the application and we can probably come up with a few ways to make it simpler so it will serve as a reasonable example for re-factoring.

The first thing to do here is to write some simple test cases to verify (or in the case of the 600 line function, determine) what the function actually does.

So let's start in `payments.tests`.

In the previous chapters exercises, the first question asked, "Our URL routing testing example only tested one route. Write tests to test the other routes. Where would you put the test to verify the pages/route? Do you need to do anything special to test the admin routes?" And in **Appendix A** we proposed the answer of using `aViewTesterMixin()`.

If you were to implement that solution here you would probably come up with a class, like `RegisterPageTests()`, that tests that the registration page returns the appropriate HTML/template data.

The code for those tests are shown below:

```
1 from django.shortcuts import render_to_response
2 import django_ecommerce.settings as settings
3 from payments.views import soon, register
4
5
6 class RegisterPageTests(TestCase, ViewTesterMixin):
7
8     @classmethod
9     def setUpClass(cls):
10         html = render_to_response(
11             'register.html',
12             {
13                 'form': UserForm(),
14                 'months': range(1, 12),
15                 'publishable': settings.STRIPE_PUBLISHABLE,
16                 'soon': soon(),
17                 'user': None,
18                 'years': range(2011, 2036),
19             }
20         )
21         ViewTesterMixin.setupViewTester(
22             '/register',
23             register,
24             html.content,
25         )
26
27     def setUp(self):
28         request_factory = RequestFactory()
```

```
29 self.request = request_factory.get(self.url)
```

If you don't have that code you can grab the appropriate tag from git by typing the following from your code directory:

```
1 git checkout tags/tdd_start
```

Notice there are a few paths through the view that we need to test:

Condition

1. `request.method != POST`
2. `request.method == POST` and `not form.is_valid()`
3. `request.method == POST` and `user.save()` is OK
4. `request.method == POST` and `user.save()` throws an `IntegrityError`:

Expected Results

1. Return basic *register.html* template
2. Return *register.html* (possibly with an error msg)
3. Set `session['user'] == user.pk` and return `HttpResponseRedirect('/')`
4. Return *register.html* with a “user is already a member” error.

Condition 1

Our existing tests covers the first case, by using the `ViewTesterMixin()`. Let's write tests for the rest.

Condition 2

- Condition: `request.method == POST` and `not form.is_valid()`
- Expected Results: Return *register.html* (possibly with an error msg)

```
1 from django.test import RequestFactory
2 import mock
3
4 def setUp(self):
5     request_factory = RequestFactory()
6     self.request = request_factory.get(self.url)
```

```

7
8 def test_invalid_form_returns_registration_page(self):
9
10     with mock.patch('payments.forms.UserForm.is_valid') as
        user_mock:
11
12         user_mock.return_value = False
13
14         self.request.method = 'POST'
15         self.request.POST = None
16         resp = register(self.request)
17         self.assertEqual(resp.content, self.expected_html)
18
19         #make sure that we did indeed call our is_valid function
20         self.assertEqual(user_mock.call_count, 1)

```

Make sure to also update the imports:

```

1 from django.test import RequestFactory
2 import mock

```

To be clear, in the above example we are only mocking out the call to `is_valid()`. Normally we might need to mock out the entire object or more functions of the object, but since it will return `False`, no other functions on from the `UserForm` will be called.

NOTE: Also take note that we are now re-creating the mock request before each test is executed in our `setUp()` method. This is because this test is changing properties of the request object and we want to make sure that each test is independent.

Condition 3

- Condition: `request.method == POST` and `user.save()` is OK
- Expected Results: Set `session['user'] == user.pk` and return `HttpResponseRedirect('/')`

For condition 3, we might start with something like this:

```

1 def test_registering_new_user_returns_successfully(self):
2
3     self.request.session = {}
4     self.request.method = 'POST'

```



```

5     self.request.POST = {
6         'email': 'python@rocks.com',
7         'name': 'pyRock',
8         'stripe_token': '4242424242424242',
9         'last_4_digits': '4242',
10        'password': 'bad_password',
11        'ver_password': 'bad_password',
12    }
13
14    resp = register(self.request)
15    self.assertEqual(resp.status_code, 200)
16    self.assertEqual(self.request.session, {})

```

The problem with this condition is that the test will actually attempt to make a request to the Stripe server - which will reject the call because the `stripe_token` is invalid.

Let's mock out the Stripe server to get around that.

```

1 def test_registering_new_user_returns_successfully(self):
2
3     self.request.session = {}
4     self.request.method = 'POST'
5     self.request.POST = {
6         'email': 'python@rocks.com',
7         'name': 'pyRock',
8         'stripe_token': '...',
9         'last_4_digits': '4242',
10        'password': 'bad_password',
11        'ver_password': 'bad_password',
12    }
13
14    with mock.patch('stripe.Customer') as stripe_mock:
15
16        config = {'create.return_value': mock.Mock()}
17        stripe_mock.configure_mock(**config)
18
19        resp = register(self.request)
20        self.assertEqual(resp.content, "")
21        self.assertEqual(resp.status_code, 302)
22        self.assertEqual(self.request.session['user'], 1)
23

```

```

24         # verify the user was actually stored in the database.
25         # if the user is not there this will throw an error
26         User.objects.get(email='python@rocks.com')

```

So now we don't have to actual call Stripe to run our test. Also, we added a line at the end to verify that our user was indeed stored correctly in the database.

NOTE: A side note on syntax: In the above code example we used the following syntax to configure the mock:

```

1 config = {'create.return_value': mock.Mock()}
2 stripe_mock.configure_mock(**config)

```

In this example we created a python dictionary called `config` and then passed it to `configure_mock` with two stars `**` in front of it. This is referred to as argument unpacking. In effect its the same thing as calling `configure_mock(create.return_value=mock.Mock())`. However, that syntax isn't allowed in python so we got around it with argument unpacking. More details about this technique can be found [here](#).

Coming back to testing, you may have noticed that these tests are getting large and testing several different things. To me that's a [code smell](#) indicating that we have a function that is doing too many things.

We should really be able to test most functions in three or four lines of code. Functions that require mocks generally need a few more, though. That said, as a rule of thumb, if your test methods start getting longer than about 10 lines of code, look for places to re-factor.

Before we do that, though, and for the sake of completeness, let's do the last test for Condition 4. This is a **BIG ONE**.

Condition 4

- Condition: `request.method == POST` and `user.save()` throws an `IntegretyError`:
- Expected Results: Return `register.html` with a "user is already a member" error.

```

1 def test_registering_user_twice_cause_error_msg(self):
2
3     #create a user with same email so we get an integrity error
4     user = User(name='pyRock', email='python@rocks.com')

```

```

5     user.save()
6
7     #now create the request used to test the view
8     self.request.session = {}
9     self.request.method = 'POST'
10    self.request.POST = {
11        'email': 'python@rocks.com',
12        'name': 'pyRock',
13        'stripe_token': '...',
14        'last_4_digits': '4242',
15        'password': 'bad_password',
16        'ver_password': 'bad_password',
17    }
18
19    #create our expected form
20    expected_form = UserForm(self.request.POST)
21    expected_form.is_valid()
22    expected_form.addError('python@rocks.com is already a
23        member')
24
25    #create the expected html
26    html = render_to_response(
27        'register.html',
28        {
29            'form': expected_form,
30            'months': range(1, 12),
31            'publishable': settings.STRIPE_PUBLISHABLE,
32            'soon': soon(),
33            'user': None,
34            'years': range(2011, 2036),
35        }
36    )
37
38    #mock out stripe so we don't hit their server
39    with mock.patch('stripe.Customer') as stripe_mock:
40
41        config = {'create.return_value': mock.Mock()}
42        stripe_mock.configure_mock(**config)
43
44    #run the test

```

```

44         resp = register(self.request)
45
46         #verify that we did things correctly
47         self.assertEqual(resp.status_code, 200)
48         self.assertEqual(self.request.session, {})
49
50         #assert there is only one record in the database.
51         users = User.objects.filter(email="python@rocks.com")
52         self.assertEqual(len(users), 1)

```

As you can see we are testing pretty much the entire system, and we have to set up a bunch of expected data and mocks so the test works correctly.

This is sometimes required when testing view functions that touch several parts of the system. But it's also a warning sign that the function may be doing too much. One other thing: If you look closely at the above test, you will notice that it doesn't actually verify the html being returned.

We can do that by adding this line:

```

1 self.assertEqual(resp.content, html.content)

```

Now if we run it... BOOM! Run it. Test Fails. Examining the results we can see the following error:

- Expected: Login
- Actual: Logout

It looks like even though a user didn't register correctly the system thinks a user is logged into the system. We can quickly verify this with some manual testing. Sure enough, after we try to register the same email the second time this is what we see:

Further, clicking on Logout will give a nasty `KeyError` as the `logout()` function tries to remove the value in `session["user"]`, which is of course not there.

There you go: We found a defect. :D

While this one won't cost us \$465 million dollars, it's still much better that we find it before our customers do. This is a type of subtle error that easily slips through the cracks, but that unit tests are good at catching. The fix is easy, though. Just set `user=None` in the `except IntegrityError:` handler within `payments/views`:

```

1 try:
2     user.save()

```

Register Today!

j@j.com is already a member

×

Name

jj

Email

j@j.com

Oops that shouldn't be there.

Figure 4.2: Registration Error

```
3 except IntegrityError:
4     form.addError(user.email + ' is already a member')
5     user = None
6 else:
7     request.session['user'] = user.pk
8     return HttpResponseRedirect('/')
```

Run the test.

```
1 Creating test database for alias 'default'...
2 .....
3 .
4 .....
5 -----
6 Ran 18 tests in 0.244s
7
8 OK
9 Destroying test database for alias 'default'...
```

Perfect.

Offense is only as good as your worst defense

Now with some good defense in place, in the form of our unit tests, we can move on to refactoring our register view, TDD style. *The important thing to keep in mind is to make one small change at a time, rerun your tests, add any new tests that are necessary, and then repeat.*

Start by pulling out the section of code that creates the new user and place it in the `User()` model class. This way, we leave all the User processing stuff to the `User()` class.

To be clear, take this section:

```
1 user = User(  
2     name = form.cleaned_data['name'],  
3     email = form.cleaned_data['email'],  
4     last_4_digits = form.cleaned_data['last_4_digits'],  
5     stripe_id = customer.id,  
6 )  
7  
8 #ensure encrypted password  
9 user.set_password(form.cleaned_data['password'])  
10  
11 try:  
12     user.save()
```

And put it in the `User()` model class, within `payments/models.py` so it looks like this:

```
1 @classmethod  
2 def create(cls, name, email, password, last_4_digits, stripe_id):  
3     new_user = cls(name=name, email=email,  
4                   last_4_digits=last_4_digits, stripe_id=stripe_id)  
5     new_user.set_password(password)  
6  
7     new_user.save()  
8     return new_user
```

Using classmethods as alternative to constructors is a idiomatic way to do things in Python. We are just following acceptable patterns here. The call to this function from the `registration()` view function is as follows:

```
1 cd = form.cleaned_data  
2 try:  
3     user = User.create(  
4         name=cd['name'],  
5         email=cd['email'],  
6         password=cd['password'],  
7         last_4_digits=cd['last_4_digits'],  
8         stripe_id=cd['stripe_id']  
9     )
```

```

4         cd['name'],
5         cd['email'],
6         cd['password'],
7         cd['last_4_digits'],
8         customer.id
9     )
10 except IntegrityError:
11     form.addError(cd['email'] + ' is already a member')
12     user = None
13 else:
14     request.session['user'] = user.pk
15     return HttpResponseRedirect('/')

```

No big surprises here.

Note: the line `cd = form.cleaned_data` is there as a placeholder. Also note that we don't actually need the previous fix of `user = None` in the `IntegrityError` handler anymore because our new `create()` function will just return `None` if there is an integrity error.

So after making the change, we can rerun all of our tests and everything should still pass!

Updating Tests

Taking a defensive approach, let's update some of our tests before moving on with the refactoring.

The first thing is to 'move' the testing of the user creation to the user model and test the `create` class method. So, in the `UserModelTest` class we can add the following two methods:

```

1 def test_create_user_function_stores_in_database(self):
2     user = User.create("test", "test@t.com", "tt", "1234", "22")
3     self.assertEqual(User.objects.get(email="test@t.com"), user)
4
5 def test_create_user_already_exists_throws_IntegrityError(self):
6     from django.db import IntegrityError
7     self.assertRaises(
8         IntegrityError,
9         User.create,
10        "test user",
11        "j@j.com",
12        "jj",
13        "1234",

```

```
14         89
15     )
```

With these tests we know that user creation is working correctly in our model and therefore if we do get an error about storing the user from our view test, we know it must be something in the view and not the model.

At this point we could actually leave all the existing register view tests as is. Or we could mock out the calls to the database since we don't strictly need to do them any more, as we already tested that the `User.create` method functions work as intended.

Let's go ahead and mock out the `test_registering_new_user_returns_successfully()` function to show how it works:

```
1 @mock.patch('stripe.Customer.create')
2 @mock.patch.object(User, 'create')
3 def test_registering_new_user_returns_successfully(
4     self, create_mock, stripe_mock
5 ):
6
7     self.request.session = {}
8     self.request.method = 'POST'
9     self.request.POST = {
10         'email': 'python@rocks.com',
11         'name': 'pyRock',
12         'stripe_token': '...',
13         'last_4_digits': '4242',
14         'password': 'bad_password',
15         'ver_password': 'bad_password',
16     }
17
18     #get the return values of the mocks, for our checks later
19     new_user = create_mock.return_value
20     new_cust = stripe_mock.return_value
21
22     resp = register(self.request)
23
24     self.assertEqual(resp.content, "")
25     self.assertEqual(resp.status_code, 302)
26     self.assertEqual(self.request.session['user'], new_user.pk)
27     #verify the user was actually stored in the database.
28     create_mock.assert_called_with(
```



```

29         'pyRock', 'python@rocks.com', 'bad_password', '4242',
        new_cust.id
30     )

```

Let's quickly explain how it works:

There are two `@mock.patch` decorators that mock out the specified method call. The return value of the mocked method call can be accessed using the `return_value()` function.

You can see that here:

```

1  #get the return values of the mocks, for our checks later
2  new_user = create_mock.return_value
3  new_cust = stripe_mock.return_value

```

We store those return values so we can use them for our assertions later on, which are used in these two assertions:

```

1  self.assertEqual(self.request.session['user'], new_user.pk)
2  #verify the user was actually stored in the database.
3  create_mock.assert_called_with(
4      'pyRock', 'python@rocks.com', 'bad_password', '4242',
        new_cust.id
5  )

```

1. The first assertion just verifies that we are setting the session value to the id of the object returned from our `create_mock`.
2. The second assertion, meanwhile, is an assertion that is built into the mock library, allowing us to check and see if a mock was called - and with what parameters. So basically we are asserting that the `User.create()` function was called as expected.

Run the tests.

```

1  Creating test database for alias 'default'...
2  .....
3  .
4  .....
5  -----
6  Ran 20 tests in 0.301s
7
8  OK
9  Destroying test database for alias 'default'...

```

A few words about mocking

While mocks can be handy at cutting out part of the system or isolating a test, in this example they are probably more trouble than they are worth. It is of course necessary to mock out the `stripe.Customer.create` call, as otherwise we would always be trying to hit their servers - NOT GOOD.

But in terms of the database, don't bother mocking it. We already have tests in the `User()` class for `create()` so we know it works. This means technically we don't need to test `create()` again in this function. But it's probably easier to go ahead and test it again, so we can avoid messing around with the mocks too much. And further, since we do already have the tests in place, and they pass, we can be fairly certain that any failure in this test is not related to user creation.

As with all things in development, there are trade offs with mocking. In terms of guidelines I generally think about it like this:

When should you Mock?

- When a function is horribly slow and you don't want to run it each time.
- Similarly, when a function needs to call something not on your development machine (i.e., web APIs).
- When a function requires setup, or state, that is more difficult to manage than setting up the mock (i.e., unique data generation).
- When a function is somebody else's code and you really don't trust it or don't want to be responsible for it (but see When not to Mock below).
- Time-dependent functionality is good to mock as it's often difficult to recreate the dependency in a unit test.

When should you not Mock?

- Not to Mock should be the default choice, when in doubt don't mock
- When it's easier or faster or simpler not to mock.
- If you are going to mock out web API's or code you don't want to be responsible for, it's a good idea to include some separate integration tests that you can run from time to time to ensure the expected functionality is still working, or hasn't been changed. As changing API's can be a huge headache when developing against third party code.

Some people swear that the speed of your unit test is paramount to all else. While it is true mocking will generally speed up the execution time of your unit test suite, unless its a significant speed increase (like in the case for `stripe.Customer.create`) my advice is: *Mocking*

is a great way to kill brain cycles; and actually improving your code is a much better use of those brain cycles.

Conclusion

The final thing you might want to re-factor out of the `register()` function is the actual Stripe customer creation.

This makes a lot of sense if you have a use case where you may want to choose between subscription vs one-time billing. You could just write a customer manager function that takes in the billing type as an argument and calls Stripe appropriately returning the customer `.id`.

We'll leave this one as an exercise for you, dear reader.

We have discussed the general approach to TDD and how following the three steps, and making one small change at a time is very important to getting TDD right.

As a review the three steps of TDD are:

1. **RED LIGHT** - write a test that fails.
2. **GREEN LIGHT** - write just enough code to make the test pass.
3. **YELLOW LIGHT** - re-factor your code/tests until you have something beautiful.
4. **RINSE and REPEAT**

Follow those steps and they should serve you well in your path to Elite Software Craftsmanship.

We also touched on applying TDD and unit testing to existing/legacy code and how to think defensively so you don't break existing functionality while trying to improve the system.

With the two testing chapters now completed we can turn our focus on some of the cool and exciting features offered in Django 1.6, 1.7, and 1.8, but first here are a couple of exercises to cement all this TDD stuff in your brain.

Exercises

1. Try mocking out the `test_registering_user_twice_cause_error_msg()` test if you really want to get your head around mocks. Start by mocking out the `User.create` function so that it throws the appropriate errors.

HINT: [this documentation](#) is a great resource and the best place to start. In particular, search for `side_effect`.

Want more? Mock out the `UserForm` as well. Good luck.

2. As alluded to in the conclusion, remove the customer creation logic from `register()` and place it into a separate `CustomerManager()` class. Re-read the first paragraph of the conclusion before you start, and don't forget to update the tests accordingly.

Chapter 5

Git Branching at a Glance

We talked about testing and re-factoring in the first few chapters of this book because they are so important to the art of Software Craftsmanship. And we will continue to use them throughout this book. But there are other tools which are important to master in the never-ending quest to become a better craftsman. In this chapter we will briefly introduce **Git Branching** so we can use it in the next chapter, which is about the art of the upgrade - i.e., Upgrading to Django 1.8.

It is assumed that the reader has some basic familiarity with how to push, pull and setup a repo with git prior to reading this chapter. If you don't, you can revisit the "Getting Started" chapter in the second Real Python course. Or you can go through this quick and to the point [git-guide](#).

Git Branching

When developing applications we are often faced with situations where we want to experiment or try new things. It would be nice if we could do this in a way that wouldn't affect our main line of code and that we could easily roll back from if our experiment didn't quite work out, or save it off and work on it later. This is exactly what Git Branching provides. Oftentimes referred to as Git's "killer feature", the branching model in Git is incredibly useful.

There are four main reasons that Git's branching model is so much more useful than branching in traditional Version Control Systems (I'm looking at you SVN):

1. *Git Branching is done in the same directory as your main line code.* No need to switch directories. No need to take up a lot of additional disk space. Just do a `git checkout <branchname>` and Git switches all the files for you automatically.
2. *Git Branching is lightning fast.* Since it doesn't copy all the files but rather just applies the appropriate [diffs](#), it's really fast.
3. *Git Branching is local.* Since your entire Git repository is stored locally, there is no need for network operations when creating a branch, allowing for branching and merging to be done completely offline (and later pushed back up to the origin server if desired). This is also another reason why Git branching is so fast; in the time it takes SVN to connect to the remote server to perform the branch operation, Git will have already completed.
4. *But the most important and useful feature of Git Branching is not Branching at all - it's merging.* Git is fantastic at merging (in fact, every time you do a `git pull` you are merging) so developers don't need to be afraid of merging branches. It's more or less automatic and Git has fantastic tools to clean up those cases where the merge needs some manual intervention.
5. *(Yeah I know I said only 4, count this one as a bonus) Merging can be undone.* Trying to merge and everything goes bad? No problem, just `git reset --hard`.

The main advantage to all of these features is that it makes it simple - in fact, advisable - to include branching as part of your daily workflow (something you generally wouldn't do in SVN). With SVN we generally branch for big things, like upgrading to Django 1.8 or another new release. With Git we can branch for everything.

One common branching pattern in Git is the **feature branch pattern** where a new branch is created for every requirement and only merged back into the main line branch after the feature is completed.

Branching Models

That's a good segue into talking about the various branching models that are often employed with Git.

A branching model is just an agreed upon way to manage your branches. It usually talks about when and for what reasons a branch is created, when it is merged (and to what branch it is merged into) and when the branch is destroyed. It's helpful to follow a branching model so you don't have to keep asking yourself, "Should I create a branch for this?". As developers, we have enough to think about, so following a model to a certain degree takes the thinking out of it - just do what the model says. That way you can focus on writing great software and not focus so much on how your branching model is supposed to work.

Let's look at some of the more common branching models used in Git.

The "I just switched from SVN and I don't want to really learn Git" Branching Model

Otherwise known as the no-branching model. This is most often employed by Git noobs, especially if they have a background in something like CVS or SVN where branching is hard. I admit when I started using Git this is exactly the model that I followed. For me, and I think for most people who follow this model, it's because branching seems scary and there is code to write, and tests to run, and who has time to learn about the intricacies of branching yadda, yadda, yadda.

The problem with this model is there is such a huge amount of useful features baked into Git that you simply are not getting access to them if you're not branching. How do you deal with production issues? How do you support multiple versions of your code base? What do you do if you have accidentally gone down the wrong track and have screwed up all your code? How do you get back to a known good state? The list goes on.

There are a ton of reasons why this isn't a good way to use Git. So do yourself a favor, take some time and at least learn the basics of git branching. Your future self will thank you. :)

Git-Flow

Vincent Driessen first documented this excellent branching model [here](#). In his article (which you *should* really read) entitled *A Successful Git Branching Model*, he goes through a somewhat complex branching model that handles just about any situation you could think of when working on a large software project. His model eloquently handles:

- Managing multiple versions of a code base simultaneously;
- Always having a known good, deployable version of the code;
- Handling production issues and hot fixes;
- Feature branches;
- Latest/development branches; and,
- Easily handling UAT and last minute release type of work, before a branch of code makes it into production.

A quick snapshot of what the branching model looks like can be seen below:

For large software projects, when working with a team, this is the recommended model. Since Vincent does such a good job of describing it on his website, we're not going to go into detail about it here, but if you are working on a large project this is a good branching model to use. For smaller projects with less team members you may want to look at the model below. Also I have reports from a number highly skilled developers use combinations of git-flow and the Github pull-request model. So choose what fits your style best - or use them all. :)

Also to make git-flow even easier to use, Vincent has created an open source tool called git-flow that is a collection of scripts that make using this branching model pretty straight-forward, which can save a lot of keystrokes. Git-Flow can be found on Github [here](#).

Thank you Vincent!

The Github Pull-Request Model

Somewhere in between the no-branching model and Git-Flow is something we like to call **The Github Pull-Request Model**. We call it the Github model because the folks at GitHub are the ones that really made the pull-request popular and easy to use. Git out of the box doesn't provide pull-request capabilities but many git front ends now do. For example, [gitorious](#) now offers pull-request functionality so you don't strictly have to be using Github - but it helps.

Let's first look at a picture of how the model works:

As you can see from the model above there are only two branches to deal with in the GitHub Pull-Request Model, as opposed to the five in the git-flow model. This makes for a very straight-forward model.

Basically all you do is create a feature branch for each requirement you are working on, then issue a pull request to merge the feature branch back into master after you are done. So really you only have one permanent branch "Master" and a bunch of short-lived feature branches. Then of course there is the pull-request which allows for an easy way to review code / alert others that the code is ready to merge back into master. Don't be fooled by its simplicity, this model works pretty effectively.

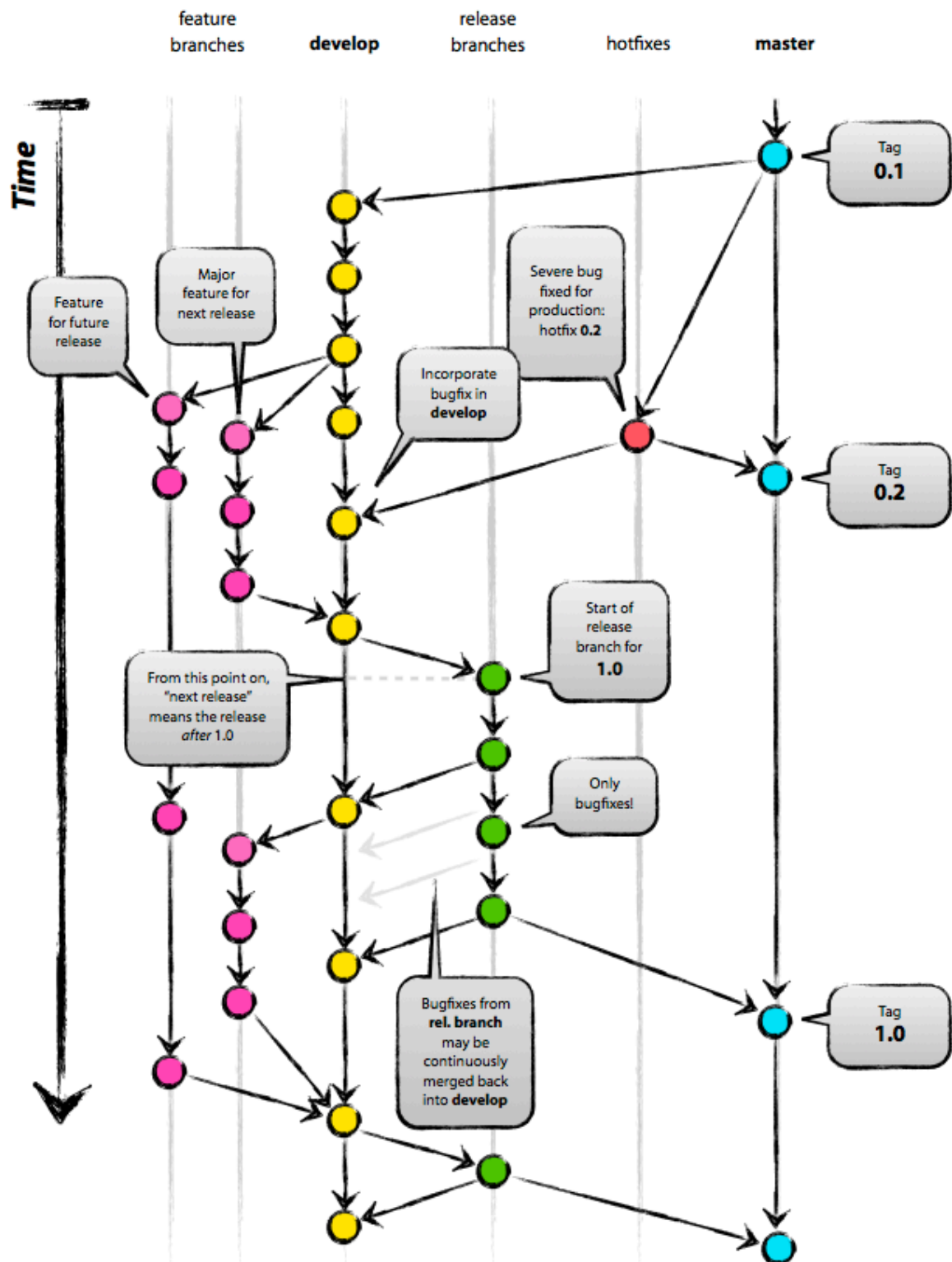


Figure 5.1: git-flow branching model

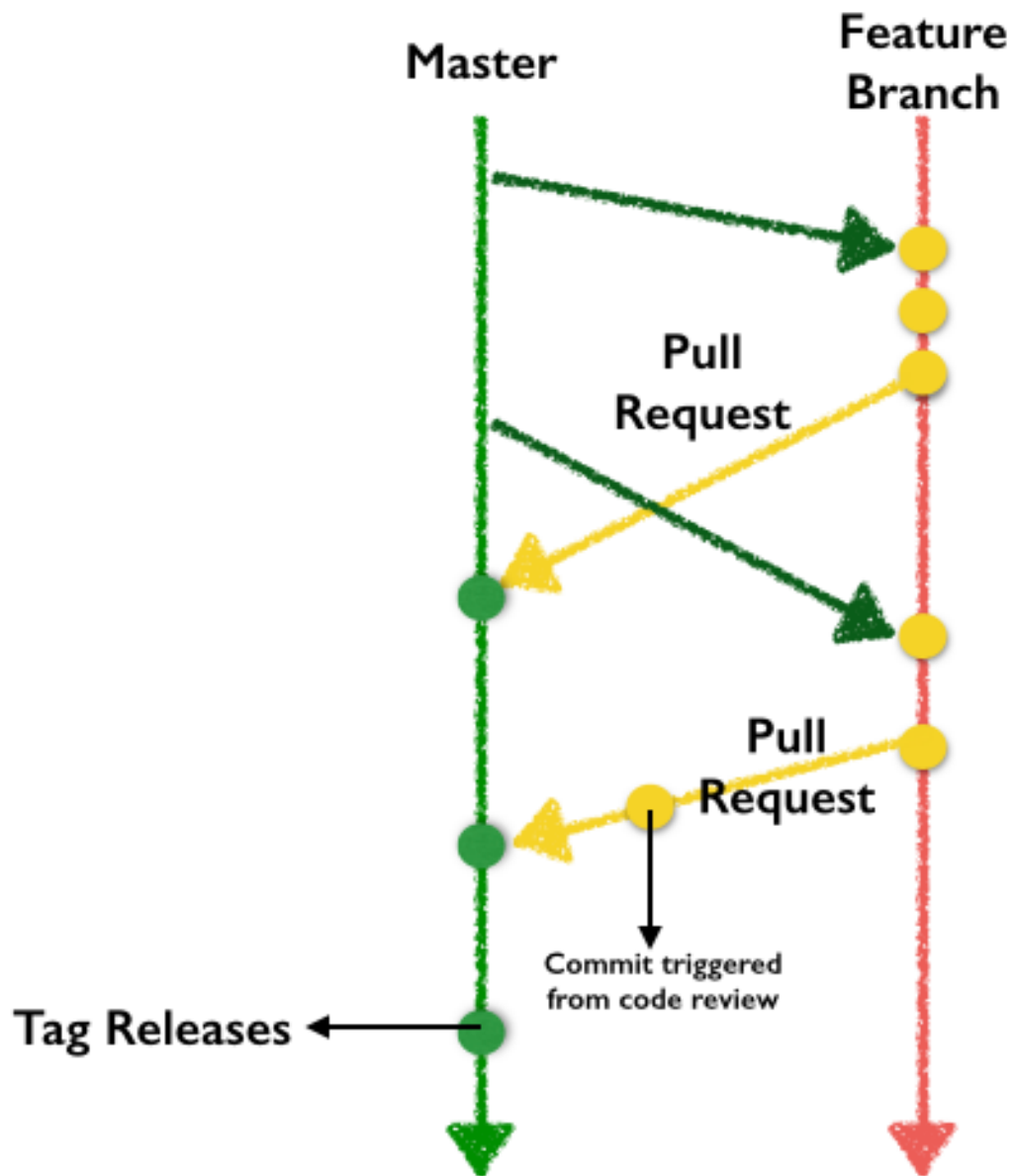


Figure 5.2: Github Pull-Request Model

Let's go through the typical workflow:

1. Create a feature branch from Master.
2. Write your code and tests on the feature branch.
3. Issue a pull request to merge your feature branch back to master when it is completed and ready to merge back into master.
4. Pull requests are used to perform code reviews and potentially make any changes necessary so the feature branch can be merged into Master.
5. Once approved, merge the feature branch back into Master.
6. Tag releases.

That's the basic outline. Pretty straight-forward. Let's look each of the steps in a bit more detail.

Create a feature branch from Master

For this model to work effectively the Master branch must always be deployable. It must be kept clean at all times, and you should *never* commit directly to master. The point of the Pull Request is to ensure that somebody else reviews your code before it gets merged into Master to try to keep Master clean. There is even a tool called [git-reflow](#) that will enforce a LGTM (Looks Good to Me) for each pull request before it is merged into master.

Write your code and tests on the feature branch

All work should be done on a feature branch to keep it separate from Master and to allow each to go through a code review before being merged back into master.

Create the branch by simply typing:

```
1 $ git checkout -b feature_branch_name
```

Then do whatever work it is you want to do. Keep in mind that commits should be incremental and atomic. Basically that means that each commit should be small and affect only one aspect of the system. This makes things easy to understand when looking through history. Wikipedia has a detailed [article](#) on atomic commit conventions.

Once you're done with your commits but before you issue the pull request, it's important to rebase against Master and tidy things up, with the goal of keeping the history useful.

Rebasing in git is a very powerful tool and you can do all kinds of things with it. Basically it allows you to take the commits from one branch and replay them on another branch. Not only

can you replay the commits but you can modify the commits as they are replayed as well. For example you can ‘squash’ several commits into a single commit. You can completely remove or skip commits, you can change the commit message on commits, or you can even edit what was included in the commit.

Now that is a pretty powerful tool. A great explanation of rebase can be found at the [git-scm book](#). For our purposes though we just want to use the `-i` or interactive mode so we can clean up our commit history so our fellow developers don’t have to bear witness to all the madness that is our day to day development. :)

Start with the following command:

```
1 $ git rebase -i origin/master
```

NOTE: the `-i` flag is optional but that gives you an interactive menu where you can reorder commits, squash them or fix them up all nice and neat before you share your code with the rest of the team.

Issuing the above command will bring up your default text editor (mine is vim) showing a list of commits with some commands that you can use:

```
1 pick 7850c5d tag: Ch3_before_exercises
2 pick 6999b75 Ch3_ex1
3 pick 6598edc Ch3_ex1_extra_credit
4 pick 8779d40 Ch3_ex2
5
6 # Rebase 7850c5d..8779d40 onto dd535a5
7 #
8 # Commands:
9 # p, pick = use commit
10 # r, reword = use commit, but edit the commit message
11 # e, edit = use commit, but stop for amending
12 # s, squash = use commit, but meld into previous commit
13 # f, fixup = like "squash", but discard this commit's log message
14 # x, exec = run command (the rest of the line) using shell
15 #
16 # These lines can be re-ordered; they are executed from top to
17 # bottom.
18 # If you remove a line here THAT COMMIT WILL BE LOST.
19 #
```

```

20 # However, if you remove everything, the rebase will be aborted.
21 #
22 # Note that empty commits are commented out

```

Above we can see there are five commits that are to be rebased. Each commit has the word `pick` in front of it followed by the SHA ID of the commit and the commit message. You can modify the word in front of the commit `pick` to any of the following commands:

```

1 # Commands:
2 # p, pick = use commit
3 # r, reword = use commit, but edit the commit message
4 # e, edit = use commit, but stop for amending
5 # s, squash = use commit, but meld into previous commit
6 # f, fixup = like "squash", but discard this commit's log message
7 # x, exec = run command (the rest of the line) using shell

```

The commands are all self explanatory. Note that both `squash` and `fixup` come in handy. `edit` is also nice. If you were to change the commits to look like the following:

```

1 pick 7850c5d Ch3_before_exercises
2 pick 6999b75 Ch3_ex1
3 edit 6598edc Ch3_ex1_extra_credit
4 pick 8779d40 Ch3_ex2

```

Then after you saved and closed the file, `rebase` would run and it would play back from the top of the file, applying commit `7850c5d` then commit `6999b75` and then the changes for commit `6598edc` will be applied and `rebase` will drop you back to the shell allowing you to make any changes you want. Then just `git add` your changes and once you issue the command-

```

1 $ git rebase --continue

```

-the changes that you `git add`-ed will be replayed as part of the `6598edc` commit. Then finally the last commit `8779d40` will be applied. This is really useful in situations where you forgot to add a file to `git` or you just need to make some small changes.

Have a play with `git rebase`; its a really useful tool and worth spending some time learning.

Once you're done with the `rebase` and you have your commit history looking just how you want it to look then it's time to move on to the next step.

Does this make sense? Would you like to see a video depicting an example? If so, please let us know! info@realpython.com

Issue a pull request to merge your feature branch back to master

Technically you don't have to use GitHub's pull request feature. You could simply push your branch up to origin and have others check it like this:

```
1 $ git push origin feature_branch_name
```

But Github's pull requests are cool and provide threaded comments, auto-merging and other good stuff as well. If you are going to use a pull request you don't need to fork the entire repository, as GitHub now allows you to issue a pull request from a branch. So after you have pushed up your branch you can just create the pull request on it through the GitHub website. Instructions for this are [here](#).

There is also an excellent set of command line tools called hub that make working with Github from the command line a breeze. You can find hub [here](#). If you're on OSX and you use homebrew (which you should) just brew install hub. Then you don't have to push your feature_branch at all just do something like this:

```
1 $ git checkout feature_branch_name
2 $ git pull-request
```

This will open a text editor allowing you to put in comments about the pull request. Be aware though that this will create a fork of the repo. If you don't want to create the fork (which you probably don't) you can do a slightly longer command such as:

```
1 $ git pull-request "pull request message" -b repo_owner:master -h
   repo_owner:feature_branch_name
```

This will create a pull request from your feature branch to the master branch for the repo owned by "repo_owner". If you're also tracking issues in Github you can link the pull request to an issue by adding -i ### where '###' is the issue to link to. Awesome! :)

Perform the Code Review

This is where the GitHub pull request feature really shines. Your team can review the pull request, make comments or even checkout the pull-request and add some commits. Once the review is done the only thing left to do is merge the pull request into Master.

Merge Pull Request into Master

Process:

```

1 $ git checkout master
2 $ git pull origin master
3 $ git merge --no-ff feature_branch_name

```

The `--no-ff` flag is important for keeping the history clean. Basically this groups all the commits from your `feature_branch` and breaks them out into a ‘merge bubble’ so they are easy to see in the history. An example would look like this:

```

1 * 5340d4b Merge branch 'new_feature_branch'
2 |\
3 | * 4273443 file2
4 | * af348dd file1.py
5 | /
6 * 9b8326c updated dummy
7 * e278c5c my first commit

```

While some people really don’t like having that “extra” merge commit - i.e., commit number 5340d4b - around, it’s actually quite helpful to have. Let’s say for example you accidentally merged and you want to undo. Then because you have that merge commit, all you have to do is:

```

1 git revert -m 1 5340d4b
2
3 # and then the entire branch merge will be undone
4 # and your history will look like
5
6 git log --graph --oneline --all
7
8 * 5d4a11c Revert "Merge branch 'new_feature_branch'"
9 * 5340d4b Merge branch 'new_feature_branch'
10 |\
11 | * 4273443 file2
12 | * af348dd file1.py
13 | /
14 * 9b8326c updated dummy
15 * e278c5c my first commit

```

So the merge bubbles are a good defensive way to cover your ass (CYA) in case you make any mistakes.

Tag your releases

The only other thing you do in this branching model is periodically tag your code when it's ready to release.

Something like:

```
$ git tag -a v.0.0.1 -m "super alpha release"
```

This way you can always checkout the tag, if for example you have production issues, and work with the code from that specific release.

That's all there is to it.

Enough about git branching

That should be enough to make you dangerous. :) But more importantly you should have a good idea as to why branching strategies are useful and how to go about using them. We will try it out for real in the next chapter on upgrading to Django 1.8.

Exercises

The exercises for this chapter are pretty fun. Enjoy!

1. Peter Cottle has created a great online resource to help out people wanting to really understand git branching. Your task is to play around with this awesome interactive tutorial. It can be found here: <http://pcottle.github.io/learnGitBranching/>. What are your thoughts? Write them down. Blog about them. Email us about it.
2. There is a great talk about the “GitHub Pull Request Model” by Zach Holman of GitHub entitled, “How GitHub uses GitHub to Build GitHub”. It’s worth watching the entire video, but the branching part starts [here](#).

Chapter 6

Upgrade, Upgrade, and Upgrade some more

Django 1.8

Up until now we have been working on Django 1.5, but it's worthwhile to look at some of the new features offered by Django 1.6, 1.7, and 1.8 as well. In addition, it will give us a chance to look at the upgrade process and what is required when updating a system. Let's face it: if you're lucky enough to build a system that lasts for a good amount of time, you'll likely have to upgrade. This will also give us a chance to try out our git branching strategy and just have some good clean code wrangling fun.

The Upgrade

Let's start off with creating a feature branch for our work.

```
1 $ git checkout -b django1.8
2 Switched to a new branch 'django1.8'
```

Then we also need to create a separate virtualenv and upgrade the version of Django in that virtualenv to 1.8. Before you do this, make sure to deactivate the main virtualenv.

```
1 $ pip freeze > requirements.txt
2 $ mkvirtualenv django1.8
3 $ pip install -r requirements.txt
4 $ pip install django==1.8.2
5 $ pip freeze > requirements.txt
```

Want to check the Django version? Enter the shell...

```
1 >>> import django
2 >>> django.VERSION
3 (1, 8, 2, 'final', 0)
```

Want to look at the branches currently available?

```
1 $ git branch
2 * django1.8
3 master
```

Test the Install

We have created a new virtualenv called *django1.8*, installed all of our dependencies in that environment, and upgraded to Django 1.8. Now to see if everything worked, we want to run all of our tests. There is a nice option you can use when running your tests, `--traceback`, that will show all warnings in addition to errors; it's a good idea to turn that on now as well so we can spot any other potential gotchas. Cross your fingers as you type...

```
1 $ ./manage.py test --traceback
```

Let's look at the error...

```
1 =====
2 ERROR: test_signin_form_data_validation_for_invalid_data
3 (payments.tests.FormTests)
```

```

4 -----
5 Traceback (most recent call last):
6 File "tests.py", line 65, in
7 test_signin_form_data_validation_for_invalid_data
8 invalid_data["data"])
9 File "/site-packages/django/test/testcases.py", line 398, in
10 assertFormError
11 contexts = to_list(response.context)
12 AttributeError: type object 'SignInForm' has no attribute 'context'
13 -----

```

Would you look at that. Django decided to take the name of our function `assertFormError`. Guess the Django folks also thought we needed something like that. Okay. Let's just change the name of our function to something like `should_have_form_error()` so it won't get confused with internal Django stuff.

You should also see the same error thrown from each one of your test cases. It looks like this:

```

1 =====
2 ERROR: tearDownClass (tests.contact.testContactModels.UserModelTest)
3 -----
4 Traceback (most recent call last):
5   File "/site-packages/django/test/testcases.py", line 962, in
6     tearDownClass
7     cls._rollback_atomics(cls.cls_atomics)
8 AttributeError: type object 'UserModelTest' has no attribute
9     'cls_atomics'

```

What's that? This is actually due to a cool new feature in Django 1.8 called `setUpTestData` meant to make it easier to load data for a test case. The documentation is [here](#). To achieve this little bit of awesomeness the `django.test.TestCase` in 1.8 uses `setUpClass` to wrap the entire test class in an 'atomic' block (see migrations chapter). Anyways the solution for this is to change our `setUpClass` function (in all of our test classes) to call the `setUpClass` function in `django.test.TestCase`. We do this with the `super` function. Here is an example for the `MainPageTetss`:

```

1 class MainPageTetss(TestCase):
2
3     @classmethod
4     def setUpClass(cls):
5         super(MainPageTetss, cls).setUpClass()
6         ... snipped the rest of the function ...

```

Basically the line starting with `super` can be read as, “Call the `setUpClass` method on the parent class of `UserModelTest` and pass in one argument `cls`.” You then have to do the same change to the `setUpClass` function for each `TestCase` class substituting the name of the class for `UserModelTest`.

NOTE: Since we will later in this chapter upgrade to Python 3 it's worth noting that the syntax to call `super` is a bit cleaner in Python 3. Instead of `super(UserModelTest, cls).setUpClass()` with Python 3 you can just call `super().setUpClass()` and it will figure out the correct arguments to use!

While it will certainly work to update all of our test cases with the called to `super().setUpClass()` Django 1.8 introduces a new test case feature which may be more appropriate for some of our test cases. The new feature called `setUpTestData` and it's a class function that is actually called from `setUpClass` is meant to be used for creating test data. From a functional perspective there are no differences between creating data for your `TestCase` in `setUpClass` vs `setUpTestData`. However due to the name of the latter, it is arguably a cleaner way to express intent, making your test cases that much easier to understand. To show an example, let's update our `payments.testUserModel.UserModelTest` class. To do this we would completely remove the `setUpClass` function and replace it with this:

```
1 @classmethod
2 def setUpTestData(cls):
3     cls.test_user = User(email="j@j.com", name='test user')
4     cls.test_user.save()
```

Doing this will create the `test_user` only one time, just before all the test functions in the class run. Also note because this is created inside an atomic block, there is no need to call `tearDownTest` to delete the data.

After updating all your test classes, re-run the tests and now they are all at least passing. Nice. But we still have a few warnings to deal with...

First warning

```
1 ../Users/michaelherman/Documents/repos/realpython/book3-exercises/_chapters/_chp06
   RemovedInDjango18Warning: Creating a ModelForm without either
   the 'fields' attribute or the 'exclude' attribute is deprecated
   - form ContactView needs updating
2 class ContactView(ModelForm):
```

This is Django being helpful. `RemovedInDjango18Warning` refers to functionality that you will have to change for the next version. But since we are upgrading anyway, let's go ahead and get rid of the warning.

This particular issue (explained in detail in the [Django 1.6 release notes](#)) is an attempt by the Django folks to make things more secure. With `ModelForms` you can automatically inherit the fields from the model, but this can lead to all fields in the model being editable by accident. To prevent this issue and to make things more explicit, you will now have to specify the fields you want to show in your `Meta` class.

The fix is simple: just add one line to the `Meta` class for `ContactView()`:

```
1 class ContactView(ModelForm):
2     message = forms.CharField(widget=forms.Textarea)
3
4     class Meta:
5         fields = ['name', 'email', 'topic', 'message']
6         model = ContactForm
```

It's only one line of extra work, and it prevents a potential security issue. If we now re-run our unit tests, you should no longer see that warning.

Let's just add one more test, to `contact.tests` to make sure that we are displaying the expected fields for our `ContactForm`:

```
1 from django.test import TestCase, SimpleTestCase
2 from contact.forms import ContactView
3
4 class ContactViewTests(SimpleTestCase):
5
6     def test_displayed_fields(self):
7         expected_fields = ['name', 'email', 'topic', 'message']
8         self.assertEqual(ContactView.Meta.fields, expected_fields)
```

Why build such a simple, seemingly worthless test? The main reason for a test like that is to communicate to other developers on our team that we intentionally have set the fields and want only those fields set. This means if someone decided to add or remove a field they will be alerted that they have broken the intended functionality which should hopefully prompt them to at least think twice and discuss with us as to why they may need to make this change.

Run tests again. They should still pass. Now let's address the other warning.

Second warning


```

1 ?:(1_6.W001) Some project unit tests may not execute as expected.
2   HINT: Django 1.6 introduced a new default test runner. It looks
      like this project was generated using Django 1.5 or
      earlier. You should ensure your tests are all running &
      behaving as expected. See
      https://docs.djangoproject.com/en/dev/releases/1.6/#discovery-of-tests-in-a
      for more information.

```

This is basically telling us there is a new test runner in town and we should be aware of and make sure all of our tests are still running. We already know that all of our tests are running so we are okay there. However, this new `django.test.runner.DiscoverRunner()` test runner is pretty cool as it allows us to structure our tests in a much more meaningful way.

Before we get into the `DiscoverRunner` though let's get rid of the warning message.

Basically Django looks at the `settings.py` file to try to determine if your original Project was generated with something previous to 1.6. Which means you will have certain default settings like `SITE_ID`, `MANAGERS`, `TEMPLATE_LOADERS` or `ADMINS`. Django also checks to see if you have any of the Django 1.6 default settings that weren't present in previous versions and then attempts to guess if your project was created with 1.6 or something older. If Django determines that you project was generated prior to 1.6 django warns you about the new `DiscoverRunner`. Conversely if django thinks your project was created with 1.6 or something newer it assume you know what your doing and doesn't give you the warning message.

The simplest way to remove the error is to specify a `TEST_RUNNER`. Let's use the new one. :)

```

1 TEST_RUNNER = 'django.test.runner.DiscoverRunner'

```

The `DiscoverRunner` is the default test runner starting in Django 1.6 and now since we explicitly set it in our `settings.py`, Django won't give us the warning anymore. Don't believe me? Run the tests again.

```

1 Creating test database for alias 'default'...
2 .....
3 .
4 .....
5 -----
6 Ran 29 tests in 0.329s
7
8 OK
9 Destroying test database for alias 'default'...

```

Boom.

NOTE: It's also worth noting that the the check management command - `./manage.py check` would have highlighted this warning as well. This command checks our configuration and ensure that it is compatible with the installed version.

With the warning out of the way, let's talk a bit about the `DiscoverRunner`.

DiscoverRunner

In Django 1.5 you were “strongly encouraged” to put your test code in the same folder as your production code in a module called *tests.py*. While this is okay for simple projects, it can make it hard to find tests for larger projects, and it breaks down when you want to test global settings or integrations that are outside of Django. For example, it might be nice to have a single file for all routing tests to verify that every route in the project points to the correct view. Starting in Django 1.6 you can do just that because you can more or less structure your tests any way that you want.

Ultimately the structure of your tests is up to you, but a common and useful structure (in many different languages) is to have your source directory (“django_ecommerce”, in our case) as well as a test top level directory. The test directory mirrors the source directory and only contains tests. That way you can easily determine which test applies to which model in the project.

Graphically that structure would look like this:

```
1
2  django_ecommerce
3      contact
4      django_ecommerce
5      main
6      payments
7  tests
8      contact
9          userModelTests.py
10         contactViewTests.py
11     django_ecommerce
12     main
13         mainPageTests.py
14     payments
```

I’ve omitted a number of the tests, but you get the idea. With the test directory tree separate and identical to the code directory, we get the following benefits:

- Separation of tests and production code.
- Better test code structure (like separate files for helper test functions; for example, the `ViewTesterMixin()` from could be refactored out into a utility module, like `tests\testutils.py`, so each utility can be easily shared amongst the various tests classes).

- Simple to see the linkage between the test cases and the code it's testing.
- Helps ensure that test code isn't accidentally copied over to a production environment.

NOTE Users of [Nose](#) or other third party testing tools for Django won't think this is a big deal because they have had this functionality for quite some time. The difference now is that it's built into Django so you don't need a third party tool.

Let's go ahead and make the changes.

Update Project Structure

You can find these changes in the “chpo6” directory in the repository. Or you can move the tests to the new directory structure yourself. If you move them yourself keep in mind that each folder must have a **init.py** or else the `DiscoverRunner` won't find the tests.

Once you have all the tests separated in the test folder, to run them all you need to do is type:

```
1 $ ./manage.py test ../tests
```

This will find all the tests in the `../tests` directory and all sub-directories. Run them now and make sure everything passes:

```
1 Creating test database for alias 'default'...
2 .....
3 .
4 .....
5 -----
6 Ran 29 tests in 0.349s
7
8 OK
9 Destroying test database for alias 'default'...
```

Once everything is working, remember we need to issue a pull request so the code can be reviewed and eventually merged back into our main line. You have been making atomic commits as described in the previous chapter correct?

To show the commits we have made we could do the following from the command line:

```
1 $ git log --oneline
2 a9b81a1 moved all tests out to a separate tests directory
3 f5dab51 updated ContactView to explicitly list fields to be
   displayed
4 7cda748 wouldn't you know it, Django took my assertFormError name...
5 58f1503 yes we are upgrading to Django 1.8
```

Now all we need to do is issue a pull request.

Pull Request with Hub

If you installed **hub** from the previous chapter, you can just type:

```
1 $ git pull-request "upgraded to django 1.8 fixed all errors" -b
   repo_owner:master -h repo_owner:django_1.8
```

NOTE: Keep in mind that `hub` must also be aliased to `git` by adding the line `alias git=hub` to your `.bash_profile` for the previous line to work.

Pull Request without Hub

If you did not install [hub](#), we can just issue a pull request by pushing the code up to Github:

```
1 $ git push origin django1.8
```

And then manually creating the pull request through the GitHub web UI.

Merge

This will issue the merge request which can be discussed and reviewed from the GitHub website. If you're unfamiliar with this process, please check out the official Github [docs](#). After the merge request has been reviewed/approved then it can be merged from the command line with:

```
1 $ git checkout master
2 $ git pull origin master
3 $ git merge --no-ff django1.8
```

That will merge everything back into master and leave us a nice little merge bubble (as described in the previous chapter) in the git history.

Summary

That's it for the upgrade to Django 1.8. Pretty straight-forward as we have a small system here. With larger systems it's just a matter of repeating the process for each error/warning and looking up any warning or errors you are not sure about in the [Django 1.6 release notes](#) as well as the [Django 1.7](#) and [Django 1.8](#) release notes. It's also a good idea to check the release notes and make sure there aren't any new changes or gotchas that may cause bugs, or change the way your site functions in unexpected ways. And of course (dare I say) manual testing shouldn't be skipped either, as automated testing won't catch everything.

Upgrading to Python 3

Django officially started supporting Python 3 in version 1.6. In the Django [docs](#) they highly recommend Python 3.3 or greater, so let's go ahead and upgrade to 3.4.1, which is the latest version as of writing.

NOTE: If you do decide to upgrade to a different version greater than 3.3, the instructions below should remain the same. Please let us know if this is not the case.

To Upgrade or Not to Upgrade - a Philosophical Debate

The first question to consider is: Why upgrade to Python 3 in the first place? It would seem pretty obvious, but lately there has been a lot of grumbling in the Python community about Python 3, so let's look at both sides of the debate.

The negative

Armin Ronacher has been one of the most popular “anti-Python 3” [posts](#). It's worth a read, but to sum it up, it basically comes down to: Porting to Python 3 is hard, and unicode/byte string support is worse than it is in Python 2. To be fair, he isn't wrong and some of the issues still exist, but a good amount of the issues he refers to are now fixed in Python 3.3.

More recently Alex Gaynor wrote a similar [post](#) to Armin's with the main point that upgrading is hard, and the Python Core Devs should make it easier.

While I don't want to discount what Armin and Alex have said, as they are both quite intelligent guys, I don't see any value in keeping 2.x around any longer than necessary. I agree that the upgrade process has been kind of confusing and there haven't seemed to be a ton of incentives to upgrade, but right now we have this scenario where we have to support two non-compatible versions, and that is never good. Look, we should all just switch to Python 3 as fast as possible (we've had 5 years already) and never look back. The sooner we put this behind us, the sooner we can get back to focusing on the joy of Python and writing awesome code.

Of course the elephant in the room is third party library support. While the support of Python 3 is getting better, there are still a number of third party libraries that don't support Python 3. So before you upgrade, make sure the libraries that you use support Python 3, otherwise you may be in for some headaches.

With that said, let's look at some of the features that are in Python 3 that would make upgrading worthwhile.

The positive

Here's a short list: Unicode everywhere, virtualenv built-in, Absolute import, Set literals, New division, Function Signature Objects, `print()` function, Set comprehension, Dict Comprehension, multiple context managers, C-based IO library, memory views, numbers modules, better exceptions, Dict Views, Type safe comparisons, standard library cleanup, logging dict config, WSGI 1.0.1, `super()` no args, Unified Integers, Better OS exception hierarchy, Metaclass class arg, lzma module, ipaddress module, faulthandler module, email packages rewritten, key-sharing dicts, nonlocal keyword, extended iterable unpacking, stable ABI, qualified names, yield from, import implemented in python, `__pycache__`, fixed import deadlock, namespace packages, keyword only arguments, function annotations, and much, much more!

While we won't cover each of these in detail, there are two really good places to find information on all the new features:

- The Python docs what's new [page](#).
- Brett Cannon's talk (core python developer) on [why 3.3 is better than 2.7](#).

There are actually a lot of really cool features, and throughout the rest of this book we'll introduce several of them to you.

So now you know both sides of the story. Let's look at the upgrade process.

Final checks before the upgrade

The first thing to do is to make sure that all of the third party Django applications and tools as well as any Python libraries that you are using support Python 3.

You can find a list of packages hosted by PyPI that support Python 3 [here](#). Some developers don't update PyPI so frequently, so it's worth checking on the project page or GitHub repo for a project if you don't find it in the list, as you may often find a fork that supports Python 3, or even a version that has the support that just hasn't been pushed to PyPI yet. Of course if the support is not there, you could always do the Python 3 upgrade yourself (I'm about to show you how) and submit a patch. :)

NOTE: You could also check whether your project is ready for Python 3 based on it's dependencies [here](#).

What type of upgrade is best?

There are several ways to perform an upgrade to Python 3. It all depends upon what end result you want. Let's look at a couple of the more common ways.

I want to support Python 2 and 3 at the same time

If you're writing a reusable Django app that you expect others to use, or some sort of library that you want to give to the community, then this is a valid option and what you most likely want to do. If you're writing your own web app (as we are doing in this course) it makes no sense to support both 2 and 3. Just upgrade to 3 and move on. Still, let's examine how you might support both 2 and 3 at the same time.

1. *Drop support for python 2.5* - while it is possible to support all versions of Python from 2.5 to 3.4 on the same code base, it's probably more trouble than it's worth. Python 2.6 was released in 2008, and given that it's a trivial upgrade from 2.5 to 2.6, it shouldn't take anybody more than 6 years to do the upgrade. So just drop the support for 2.5 and make your life easier. :)
2. *Set up a Python 3 virtualenv* - assuming you already have an existing Python 2 virtualenv that you've been developing on, set up a new one with Python 3 so you can test your changes against both Python 2 and Python 3 and make sure they work in both cases. You can set up a virtualenv using Python 3 with the following command:

```
$ virtualenv -p /usr/local/bin/python3 <path/to/new/virtualenv/>
```

where `/usr/local/bin/python3` is the path to wherever you installed Python 3 on your system.

3. *Use [2to3](#)* - 2to3 is a program that will scan your Python project and report on what needs to change to support Python 3. It can be used with the `-w` option asking it to change the files automatically, but you wouldn't want to do that if you want to continue to offer Python 2 support. However, the report it generates can be invaluable in helping you make the necessary changes.
4. *Make the changes* - be sure to test the code on your Python 2 virtualenv as well as your Python 3 virtualenv. When you have everything working, you're done. Pat yourself on the back.

Finally, here's a few resources that may help if you get stuck:

- Python’s official [documentation](#) on Porting to Python 3
- Also the “six” compatibility layer is a wonderful library that takes care of a lot of the work of supporting both Python 2 and 3 at the same time. You can find it [here](#). Also note that six is what Django uses internally to support both 2 and 3 on the same code base, so you can’t go wrong with six.
- As an example of a project upgraded to support both Python 2 and 3 on the same code base, have a look at the following [pull request](#). (Recognize the name of the contributor?)

I just want to upgrade to Python 3 and never look back

This is what we are going to do for this project. Let’s look at it step by step.

1. *Create a virtualenv for Python 3:*

```
1 $ mkvirtualenv -p /usr/local/bin/python3 py3
```

2. *Install all our dependencies in the upgraded environment:*

```
1 $ pip install -r requirements.txt
```

The previous commands creates a new virtualenv with Python 3, switches you into that environment, and then installs all the project dependencies.

3. *Run 2to3 - now let’s see what we need to change to support Python 3:*

```
1 $ 2to3 django_ecommerce
```

It will return a diff listing of what needs to change, which should look like:

```
1 RefactoringTool: Skipping implicit fixer: buffer
2 RefactoringTool: Skipping implicit fixer: idioms
3 RefactoringTool: Skipping implicit fixer: set_literal
4 RefactoringTool: Skipping implicit fixer: ws_comma
5 RefactoringTool: No changes to django_ecommerce/manage.py
6 RefactoringTool: No changes to django_ecommerce/contact/forms.py
7 RefactoringTool: No changes to django_ecommerce/contact/models.py
8 RefactoringTool: No changes to django_ecommerce/contact/views.py
9 RefactoringTool: No changes to
  django_ecommerce/django_ecommerce/settings.py
10 RefactoringTool: No changes to
  django_ecommerce/django_ecommerce/urls.py
```

```

11 RefactoringTool: No changes to
    django_ecommerce/django_ecommerce/wsgi.py
12 RefactoringTool: No changes to django_ecommerce/main/views.py
13 RefactoringTool: Refactored django_ecommerce/payments/forms.py
14 --- django_ecommerce/payments/forms.py (original)
15 +++ django_ecommerce/payments/forms.py (refactored)
16 @@ -29,12 +29,12 @@
17     email = forms.EmailField(required=True)
18     password = forms.CharField(
19         required=True,
20 -     label=(u'Password'),
21 +     label=('Password'),
22     widget=forms.PasswordInput(render_value=False)
23 )
24     ver_password = forms.CharField(
25         required=True,
26 -     label=(u' Verify Password'),
27 +     label=(' Verify Password'),
28     widget=forms.PasswordInput(render_value=False)
29 )
30
31 RefactoringTool: No changes to django_ecommerce/payments/models.py
32 RefactoringTool: Refactored django_ecommerce/payments/views.py
33 --- django_ecommerce/payments/views.py (original)
34 +++ django_ecommerce/payments/views.py (refactored)
35 @@ -33,7 +33,7 @@
36     else:
37         form = SigninForm()
38
39 -     print form.non_field_errors()
40 +     print(form.non_field_errors())
41
42     return render_to_response(
43         'sign_in.html',
44 @@ -92,11 +92,11 @@
45         'register.html',
46         {
47             'form': form,
48 -             'months': range(1, 12),
49 +             'months': list(range(1, 12)),

```

```

50         'publishable': settings.STRIPE_PUBLISHABLE,
51         'soon': soon(),
52         'user': user,
53 -         'years': range(2011, 2036),
54 +         'years': list(range(2011, 2036)),
55     },
56     context_instance=RequestContext(request)
57 )
58 @@ -133,8 +133,8 @@
59         'form': form,
60         'publishable': settings.STRIPE_PUBLISHABLE,
61         'soon': soon(),
62 -         'months': range(1, 12),
63 -         'years': range(2011, 2036)
64 +         'months': list(range(1, 12)),
65 +         'years': list(range(2011, 2036))
66     },
67     context_instance=RequestContext(request)
68 )
69 RefactoringTool: Files that need to be modified:
70 RefactoringTool: django_ecommerce/manage.py
71 RefactoringTool: django_ecommerce/contact/forms.py
72 RefactoringTool: django_ecommerce/contact/models.py
73 RefactoringTool: django_ecommerce/contact/views.py
74 RefactoringTool: django_ecommerce/django_ecommerce/settings.py
75 RefactoringTool: django_ecommerce/django_ecommerce/urls.py
76 RefactoringTool: django_ecommerce/django_ecommerce/wsgi.py
77 RefactoringTool: django_ecommerce/main/views.py
78 RefactoringTool: django_ecommerce/payments/forms.py
79 RefactoringTool: django_ecommerce/payments/models.py
80 RefactoringTool: django_ecommerce/payments/views.py

```

The diff listing shows us a number of things that need to change. We could just run `2to3 -w` and let it handle these changes for us, but that doesn't teach us much. So let's look at the errors and see where we need to make changes.

2to3

```

1 RefactoringTool: Refactored django_ecommerce/payments/forms.py
2 --- django_ecommerce/payments/forms.py (original)

```

```

3 +++ django_ecommerce/payments/forms.py (refactored)
4 @@ -29,12 +29,12 @@
5     email = forms.EmailField(required=True)
6     password = forms.CharField(
7         required=True,
8 -     label=(u'Password'),
9 +     label=('Password'),
10    widget=forms.PasswordInput(render_value=False)
11 )
12    ver_password = forms.CharField(
13        required=True,
14 -    label=(u'Verify Password'),
15 +    label=('Verify Password'),
16    widget=forms.PasswordInput(render_value=False)
17 )

```

In the first change, we just need to add u in front of the strings. Remember in Python 3 everything is unicode. There are no more pure ansi strings (there could however be byte strings, denoted with a b), so this change is just denoting the strings as unicode. Strictly speaking in Python 3 this is not required, in fact its redundant, since strings are always treated as unicode. 2to3 puts it in so you can support both Python 2 and 3 at the same time, which we are not doing. So we can skip this one.

On to the next issues:

```

1 -     print form.non_field_errors()
2 +     print(form.non_field_errors())

```

In Python 3 `print` is a function, and like any function, you have to surround the arguments with (). This has been changed just to standardize things. So we need to make that change.

The next couple of issues are the same:

```

1 -         'years': range(2011, 2036),
2 +         'years': list(range(2011, 2036)),

```

and

```

1 -         'months': range(1, 12),
2 -         'years': range(2011, 2036)
3 +         'months': list(range(1, 12)),
4 +         'years': list(range(2011, 2036))

```

Above 2to3 is telling us that we need to convert the return value from a `range()` to a `list`. This is because in Python 3, `range()` effectively behaves the same as `xrange()` in Python 2. The switch to returning a generator - `range()` in Python 3 and `xrange()` in Python 2 - is for efficiency purposes. With generators there is no point in building an entire list if we are not going to use it. In our case here we are going to use all the values, always, so we just go ahead and convert the generator to a list.

That's about it for code.

You can go ahead and make the changes manually or let 2to3 do it for you:

```
1 $ 2to3 django_ecommerce -w
```

Test 2to3 changes

If we try to run our tests we will get a bunch of errors because the tests also need to be converted to Python 3. Running 2to3 on our tests directory - `2to3 tests` - will give us the following errors.

```
1 RefactoringTool: Skipping implicit fixer: buffer
2 RefactoringTool: Skipping implicit fixer: idioms
3 RefactoringTool: Skipping implicit fixer: set_literal
4 RefactoringTool: Skipping implicit fixer: ws_comma
5 RefactoringTool: No changes to tests/contact/testContactModels.py
6 RefactoringTool: No changes to tests/main/testMainPageView.py
7 RefactoringTool: No changes to tests/payments/testCustomer.py
8 RefactoringTool: Refactored tests/payments/testForms.py
9 --- tests/payments/testForms.py (original)
10 +++ tests/payments/testForms.py (refactored)
11 @@ -29,9 +29,9 @@
12     def test_signin_form_data_validation_for_invalid_data(self):
13         invalid_data_list = [
14             {'data': {'email': 'j@j.com'},
15 -           'error': ('password', [u'This field is required.'])},
16 +           'error': ('password', ['This field is required.'])},
17             {'data': {'password': '1234'},
18 -           'error': ('email', [u'This field is required.'])}
19 +           'error': ('email', ['This field is required.'])}
20         ]
21
22         for invalid_data in invalid_data_list:
23 @@ -78,14 +78,14 @@
```

```

24         'data': {'last_4_digits': '123'},
25         'error': (
26             'last_4_digits',
27 -         [u'Ensure this value has at least 4 characters
28 +         ['Ensure this value has at least 4 characters
29         (it has 3).']
30         )
31     },
32     {
33         'data': {'last_4_digits': '12345'},
34         'error': (
35 -             'last_4_digits',
36 +             [u'Ensure this value has at most 4 characters
37 +             ['Ensure this value has at most 4 characters
38         (it has 5).']
39         )
40     }
41 ]
42 RefactoringTool: No changes to tests/payments/testUserModel.py
43 RefactoringTool: Refactored tests/payments/testviews.py
44 --- tests/payments/testviews.py (original)
45 +++ tests/payments/testviews.py (refactored)
46 @@ -80,11 +80,11 @@
47         'register.html',
48         {
49 -            'form': UserForm(),
50 -            'months': range(1, 12),
51 +            'form': UserForm(),
52 +            'months': list(range(1, 12)),
53 +            'publishable': settings.STRIPE_PUBLISHABLE,
54 +            'soon': soon(),
55 +            'user': None,
56 -            'years': range(2011, 2036),
57 +            'years': list(range(2011, 2036)),
58         }
59     )
60     ViewTesterMixin.setupViewTester(
61 RefactoringTool: Files that need to be modified:
62 RefactoringTool: tests/contact/testContactModels.py

```

```

60 RefactoringTool: tests/main/testMainPageView.py
61 RefactoringTool: tests/payments/testCustomer.py
62 RefactoringTool: tests/payments/testForms.py
63 RefactoringTool: tests/payments/testUserModel.py
64 RefactoringTool: tests/payments/testviews.py

```

Looking through the output we can see the same basic errors that we had in our main code:

1. Include `u` before each string literal,
2. Convert the generator returned by `range()` to a `list`, and
3. Call `print` as a function.

Applying similar fixes as we did above will fix these, and then we should be able to run our tests and have them pass. To make sure after you have completed all your changes, type the following from the terminal:

```

1 $ cd django_ecommerce
2 $ ./manage.py test ../tests

```

Whoa! Errors all over the place.

```

1 Creating test database for alias 'default'...
2 F.....F.....
3 .
4 ...F.
5 =====
6 FAIL: test_contactform_str_returns_email
   (tests.contact.testContactModels.UserModelTest)
7 -----
8 Traceback (most recent call last):
9   File "../testContactModels.py", line 23, in
10     test_contactform_str_returns_email
11     self.assertEqual("first@first.com", str(self.firstUser))
12 AssertionError: 'first@first.com' != 'ContactForm object'
13 - first@first.com
14 + ContactForm object
15
16 =====
17 FAIL: test_registering_new_user_returns_successfully
   (tests.payments.testviews.RegisterPageTests)

```



```

18 -----
19 Traceback (most recent call last):
20   File "../py3/lib/python3.4/site-packages/mock.py", line 1201, in
      patched
21     return func(*args, **kwargs)
22   File "../testviews.py", line 137, in
      test_registering_new_user_returns_successfully
23     self.assertEqual(resp.content, "")
24 AssertionError: b'' != ''
25
26 =====
27 FAIL: test_returns_correct_html
      (tests.payments.testviews.SignOutPageTests)
28 -----
29 Traceback (most recent call last):
30   File "../testviews.py", line 36, in test_returns_correct_html
31     self.assertEqual(resp.content, self.expected_html)
32 AssertionError: b'' != ''
33
34 -----
35 Ran 29 tests in 0.297s
36
37 FAILED (failures=3)
38 Destroying test database for alias 'default'...

```

As it turns out, 2to3 can't catch everything, so we need to go in and fix the issues. Let's deal with them one at a time.

First error

```

1 =====
2 FAIL: test_contactform_str_returns_email
      (tests.contact.testContactModels.UserModelTest)
3 -----
4 Traceback (most recent call last):
5   File
      "/Users/michaelherman/Documents/repos/realpython/book3-exercises/_chapters/ch
      line 23, in test_contactform_str_returns_email
6     self.assertEqual("first@first.com", str(self.firstUser))
7 AssertionError: 'first@first.com' != 'ContactForm object'

```

```
8 - first@first.com
9 + ContactForm object
```

It appears that calling `str` and passing our `ContactForm` no longer returns the user's name. If we look at `contact/models.py/ContactForm` we can see the following function:

```
1 def __unicode__(self):
2     return self.email
```

In Python 3 the `unicode()` built-in function has gone away and `str()` always returns a unicode string. Therefore the `__unicode__()` function is just ignored.

Change the name of the function to fix this issue:

```
1 def __str__(self):
2     return self.email
```

This and various other Python 3-related errata for Django can be found [here](#).

Run the tests again. A few left, but lucky for us they all have the same solution

Unicode vs Bytestring errors

They basically all look like the following:

```
1 FAIL: test_registering_new_user_returns_successfully
   (tests.payments.testviews.RegisterPageTests)
2 -----
3 Traceback (most recent call last):
4   File "/py3/lib/python3.4/site-packages/mock.py", line 1201, in
   patched
5     return func(*args, **kwargs)
6   File "../testviews.py", line 137, in
   test_registering_new_user_returns_successfully
7     self.assertEqual(resp.content, "")
8 AssertionError: b'' != ''
```

In Python 2 a bytestring (`b'some_string'`) is effectively the same as a unicode string (`u'some_string'`) as long as 'some_string' only contains ASCII data. However in Python 3 bytestrings should be used only for binary data or when you actually want to get at the bits and bytes. In Python 3 `b'some_string' != u'some_string'`.

BUT, and the big but here, is that according to [PEP 3333](#), input and output streams are always byte objects. What is `response.content`? It's a stream. Thus, it should be a byte object.

Django's recommended [solution](#) for this is to use `assertContains()` or `assertNotContains()`. Unfortunately `assertContains` doesn't handle redirects - e.g., a `status_code` of 302. And from our errors we know it's the redirects that are causing the problems. The solution then is to change the `setUpClass` method for the classes that test for redirects in `test/payments/testViews` in our `SignOutPageTest` class:

```
1 class SignOutPageTests(TestCase, ViewTesterMixin):
2
3     @classmethod
4     def setUpClass(cls):
5         ViewTesterMixin.setUpViewTester(
6             '/sign_out',
7             sign_out,
8             b"", # a redirect will return an empty bytestring
9             status_code=302,
10            session={"user": "dummy"},
11        )
```

Here, we just changed the third argument from `""` to `b""`, because `response.context` is now a bytestring, so our `expected_html` must also be a bytestring. The last error in `tests/payments/testviews/` in the `test_registering_new_user_returns_successfully` test is the same type of bytestring error, with the same type of fix:

Update:

```
python self.assertEqual(resp.content, b)
```

To:

```
1 self.assertEqual(resp.content, b"")
```

Same story as before: We need to make sure our string types match. And with that, we can rerun our tests and they all pass.

```
1 Creating test database for alias 'default'...
2 .....
3 .
4 .....
5 -----
6 Ran 29 tests in 0.430s
7
8 OK
9 Destroying test database for alias 'default'...
```

Success!

So now our upgrade to Python 3 is complete. We can commit our changes, and merge back into the master branch. We are now ready to move forward with Django 1.8 and Python 3. Awesome.

Python 3 Changes Things Slightly

Before moving on with the rest of the chapter, have a look around the directory tree for the project. Notice anything different? You should now have a bunch of `__pycache__` directories. Let's look at the contents of one of them:

```
1 $ ls -al django_ecommerce/django_ecommerce/__pycache__
2 total 24
3 drwxr-xr-x  5 michaelherman  staff   170 Jul 25 19:12 .
4 drwxr-xr-x 11 michaelherman  staff  374 Jul 25 19:12 ..
5 -rw-r--r--  1 michaelherman  staff  208 Jul 25 19:12
   __init__.cpython-34.pyc
6 -rw-r--r--  1 michaelherman  staff 2690 Jul 25 19:12
   settings.cpython-34.pyc
7 -rw-r--r--  1 michaelherman  staff  852 Jul 25 19:12
   urls.cpython-34.pyc
```

There are a few things to note about the directory structure:

1. Each of these files is a *.pyc*. No longer are pyc files littered throughout your project; with Python 3 they are all kept in the appropriate `__pycache__` directory. This is nice as it cleans things up a bit and un-clutters your code directory.
2. Further, *.pyc* files are now in the format `<file_name>.<vm_name>-<python_version>.pyc`. This allow for storing multiple *.pyc* files; if you're testing against Python 3.4 and 2.7, for example, you don't have to regenerate your *.pyc* files each time you test against a different environment. Also each VM (jython, pypy, etc.) can store its own *.pyc* files so you can run against multiple VMs without regenerating *.pyc* files as well. This will come in handy later when we look at running multiple test configurations with Travis CI.

All in all the `__pycache__` directory provides a cleaner, more efficient way of handling multiple versions of Python and multiple VMs for projects that need to do that. For projects that don't... well, at least it gets the *.pyc* files out of your code directories.

As said before, there are a ton of new features in Python 3 and we will see several of them as we progress throughout the course. We've seen a few of the features already during our upgrade. Some may seem strange at first; most of them can be thought of as cleaning up the Python API and making things clearer. While it may take a bit to get used to the subtle changes in Python 3, it's worth it. So stick with it.

Upgrading to PostgreSQL

Since we are on the topic of upgrading, let's see if we can fit one more quick one in.

Up until now we have been using SQLite, which is OK for testing and prototyping purposes, but you would probably wouldn't want to use SQLite in production. (At least not for a high traffic site. Take a look at SQLite's own [when to use page](#) for more info). Since we are trying to make this application production-ready, let's go ahead and upgrade to [PostgreSQL](#). The process is pretty straightforward, especially since Postgres fully supports Python 3.

By now it should go without saying that we are going to start working from a new branch..so I won't say it. :). Also as with all upgrades, it's a good idea to back things up first, so let's pull all the data out of our database like this:

```
1 $ ./manage.py dumpdata > db_backup.json
```

This is a built-in Django [command](#) that exports all the data to JSON. The nice thing about this is that JSON is database-independent, so we can easily pass this data to other databases. The downside of this approach is that metadata like primary and foreign keys are not saved. For our example that isn't necessary, but if you are trying to migrate a large database with lots of tables and relationships, this may not be the best approach. Let's first get Postgres set up and running, then we can come back and look at the specifics of the migration.

Step 1: Install PostgreSQL

There are many ways to install PostgreSQL depending upon the system that you are running, including the prepackaged binaries, `apt-get` for debian-based systems, `brew` for Mac, compiling from source and others. We're not going to go into the nitty-gritty details of every single means of installation; you'll have to figure that out on your own. Use your Google searching skills. Check out the official docs for help - [PostgreSQL Installation Page](#)

Windows Users

The easiest way to install Postgres on Windows is with a product called EnterpriseDB. Download it from [here](#). Make sure to choose the newest version available for your operating system. Download the installer, run it and then follow these [instructions](#).

Mac OS Users

Brew it baby. We talked about homebrew before so now that you have it installed from the command line just type:

```
1 $ brew install postgresql
```

Debian Users

1. Install Postgres

```
1 $ sudo apt-get install postgresql
```

2. Verify installation is correct:

```
1 $ psql --version
```

And you should get something like the following:

```
1 psql (PostgreSQL) 9.2.4
```

3. Now that Postgres is installed, you need to set up a database user and create an account for Django to use. When Postgres is installed, the system will create a user named postgres. Let's switch to that user so we can create an account for Django to use:

```
1 $ sudo su postgres
```

4. Creates a Django user in the database:

```
1 $ createuser -P.djangouser
```

Enter the password twice and remember it; you will use it in your *settings.py* later.

5. Now using the postgres shell, create a new database to use in Django. **Note:** don't type the lines starting with a #. These are comments for your benefit.

```
1 $ psql
2
3 # Once in the shell type the following to create the database
4 CREATE DATABASE django_db OWNER.djangouser ENCODING 'UTF8';
5
6 # Then to quit the shell type
7 $ \q
```

6. Then we can set up permissions for Postgres to use by editing the file `/etc/postgresql/9.1/main/pg_hba.conf`. Just add the following line to the end of the file:

```
1 local    django_db         .djangouser    md5
```

Then save the file and exit the text editor. The above line basically says `djangouser` user can access the `django_db` database if they are initiating a local connection and using an md5-encrypted password.

7. Finally restart the postgres service.

```
1 $ sudo /etc/init.d/postgresql restart
```

This should restart postgres and you should now be able to access the database. Check that your newly created user can access the database with the following command:

```
1 $ psql django_db djangouser
```

This will prompt you for the password; type it in and you should get to the database prompt. You can execute any SQL statements that you want from here, but at this point we just want to make sure we can access the database, so just do a \q and exit out of the database shell. You're all set- Postgres is working! You probably want to do a final exit from the command line to get back to the shell of your normal user.

If you do encounter any problems installing PostgreSQL, check the [wiki](#). It has a lot of good troubleshooting tips.

Now that Postgres is installed, you will want to install the Python bindings.

Step 2: Install Python bindings

Ensure that you are in the virtualenv that we created earlier, and then install the PostgreSQL binding, [psycopg2](#) with the following command:

```
1 $ pip install psycopg2
```

This will require a valid build environment and is prone to failure, so if it doesn't work you can try one of the several pre-packaged installations. For example, if you're on a Debian-based system, you'll probably need to install a few dependencies first:

```
1 $ sudo apt-get install libpq-dev python-dev libpython3.3-dev
```

Once done, try re-running the `pip install psycopg2` command.

Alternatively, if you want to install psycopg2 globally you can use apt-get for that:

```
1 $ sudo apt-get install install postgresql-plpython-9.1
```

WARNING: To be clear installing Postgres globally is not an issue as far as this course is concerned. It can however become an issue if you are working on multiple projects each using a different version of Postgres. If you have Postgres installed globally then it makes it much more difficult to have multiple version installed for different applications, which is exactly the problem that virtualenv is meant to fix. If you don't plan on running multiple versions of Postgres then by all means install it globally and don't worry about it.

Since we are adding another Python dependency to our project, we should update the `requirements.txt` file in our project root directory to reflect this.

```
1 $ pip freeze > requirements.txt
```

The should now look something like this:

```
1 Django==1.8.2
2 mock==1.0.1
3 psycopg2==2.5.4
4 requests==2.3.0
5 stripe==1.9.2
```

That's all the dependencies we need for now.

OSX Homebrew Users: If you installed Python and/or Postgres through homebrew and the `pip install psycopg2` command fails, ensure that you're using a version of Python installed by homebrew and that you installed Postgres via homebrew. If you did, then everything should work properly. If however you have some combination of the OSX system Python and a Homebrew Postgres - or vice versa - things are likely to fail.

Windows users: Probably the easiest user experience is to use the pre-packaged installers, however these will install Postgres globally and not to your specific virtualenv. The installer is available [here](#). Be sure to get the Python 3 version!

Step 3: Configure Django

You've got postgres set up and working, and you've got the Python bindings all installed. We are almost there. Now let's configure Django to use Postgres instead of SQLite. This can be done by modifying `settings.py`. Remove the SQLite entry and replace it with the following entry for Postgres:

```
1 DATABASES = {
2     'default': {
3         'ENGINE': 'django.db.backends.postgresql_psycopg2',
4         'NAME': 'django_db',
5         'USER': 'djangouser',
6         'PASSWORD': 'your_password_here',
7         'HOST': 'localhost',
8         'PORT': '5432',
9     }
10 }
```

Please note that 5432 is the default postgres port, but you can double-check your system setup with the following command:

```
1 $ grep postgres /etc/services
2 postgresql      5432/udp      # PostgreSQL Database
3 postgresql      5432/tcp      # PostgreSQL Database
```

Once that's done just sync the new database:

```
1 $ ./manage.py syncdb
```

NOTE You will get a very different output from running syncdb than you are used to after you upgrade to Django 1.8 This is because of the new key feature of Django 1.8: migrations. We have an entire chapter coming up dedicated to migrations, so for now just ignore the output, and we will come back to syncdb and migrations in an upcoming chapter.

Step 4: Import backup data

The final thing is to import the `db_backup.json` file we created earlier:

```
1 $ ./manage.py loaddata db_backup.json
2 Installed 51 object(s) from 1 fixture(s)
```

Of course this isn't essential if you're following this course strictly as you won't yet have much, if any, data to load. But this is the general idea of how you might go about migrating data from one database to the next.

Don't forget to make sure all of our tests still run correctly:

```
1 $ ./manage.py test ../tests
```

Did you get this error?

```
1 Got an error creating the test database: permission denied to
   create database
```

If so, check out [this](#) StackOverflow answer. Then enter the Postgres shell and run the following command:

```
1 ALTER USER djangouser CREATEDB;
```

Test again. Everything should pass. Congratulations! You've upgraded to PostgreSQL. Be sure to commit all your changes and merge your branch back to master - and then you're ready to move on.

Conclusion

Whoa - that's a lot of upgrading. Get used to it. Software moves so fast these days, that software engineers can't escape the reality of the upgrade. If it hasn't happened to you yet, it will. Hopefully this chapter has provided you with some food for thought on how to approach an upgrade and maybe even given you a few upgrade tools to keep in your tool-belt for the next time you have to upgrade.

Exercises

There are some really good videos out there on upgrading to Python 3 and Django 1.6/1.7/1.8. So for this chapter's exercises, watch the following videos to gain a better understanding of why you might want to upgrade:

1. "Porting Django apps to Python 3" [talk](#) by the creator of Django Jacob Kaplan-Moss
2. "Python 3.3 Trust Me, It's Better than 2.7" by Brett Cannon (core python developer) - [video](#)

Chapter 7

Graceful Degradation and Database Transactions with Django 1.8

Graceful Degradation used to be a hot topic. Today you don't hear about it too much, at least not in the startup world where everyone is rushing to get their MVP out the door and get sign-ups. "We'll fix it later, right?" Starting in Django 1.6 database transactions were drastically simplified. So we can actually apply a little bit of Graceful Degradation to our sign-in process without having to spend too much time on it.

What is Graceful Degradation?

Let's start with a quick definition of **Graceful Degradation**: *Put simple, it's the property that enables a system to continue operating properly in the event of a failure to some of its components.* If something breaks, we just keep on ticking, in other words.

A common example of this in web development is support for Internet Explorer. Although Internet Explorer is getting better, a lot of the cool functionality (WebGL for example) that makes web sites fun simply is not supported. To compensate, developers must write the front end code with a bunch of IF-type statements - IF browser == IE THEN do something basic" - because the cool stuff is broken.

Let's try to apply this thinking to our registration function.

User Registration

If Stripe is down, you still want your users to be able to register, right? We just want to hold their info and then re-verify it when Stripe is back up, otherwise we will probably lose that user to a competitor if we don't allow them to register until Stripe comes back up. Let's look at how we can do that.

Following what we learned before, let's first create a branch for this feature:

```
1 $ git checkout -b transactions
```

Now that we are working in a clean environment, let's write a unit test in 'tests/payments/testViews' in the `RegisterPageTests()` class:

```
1 import socket
2
3 def test_registering_user_when_stripe_is_down(self):
4
5     #create the request used to test the view
6     self.request.session = {}
7     self.request.method = 'POST'
8     self.request.POST = {
9         'email': 'python@rocks.com',
10        'name': 'pyRock',
11        'stripe_token': '...',
12        'last_4_digits': '4242',
13        'password': 'bad_password',
14        'ver_password': 'bad_password',
15    }
16
17    #mock out Stripe and ask it to throw a connection error
18    with mock.patch(
19        'stripe.Customer.create',
20        side_effect=socket.error("Can't connect to Stripe")
21    ) as stripe_mock:
22
23        #run the test
24        register(self.request)
25
26        #assert there is a record in the database without Stripe id.
27        users = User.objects.filter(email="python@rocks.com")
```

```

28         self.assertEqual(len(users), 1)
29         self.assertEqual(users[0].stripe_id, '')

```

That should do it. We have just more or less copied the test for `test_registering_new_user_returns_success` but removed all the database mocks and added a `stripe_mock` that throws a `socket.error` every time it's called. This should simulate what would happen if Stripe goes down.

Of course running this test is going to fail - `OSError: Can't connect to Stripe`. But that's exactly what we want (remember the TDD Fail, Pass, Refactor loop). Don't forget to commit that change.

Okay. So, how can we get it to work?

Well, we want to hold their info and then re-verify it when Stripe is back up.

First thing is to update the `Customer.create()` in `payments/views/` method to handle that pesky `socket.error` so we don't have to deal with it.

```

1 import socket
2
3 class Customer(object):
4
5     @classmethod
6     def create(cls, billing_method="subscription", **kwargs):
7         try:
8             if billing_method == "subscription":
9                 return stripe.Customer.create(**kwargs)
10            elif billing_method == "one_time":
11                return stripe.Charge.create(**kwargs)
12        except socket.error:
13            return None

```

This way when Stripe is down, our call to `Customer.create()` will just return `None`. This design is preferable so we don't have to put try-except blocks everywhere.

Rerun the test. It should still fail:

```

1 AttributeError: 'NoneType' object has no attribute 'id'

```

So now we need to modify the `register()` function. Basically all we have to do is change the part that saves the user.

Change:

```

1 try:
2     user = User.create(

```



```

3         cd['name'],
4         cd['email'],
5         cd['password'],
6         cd['last_4_digits'],
7         customer.id
8     )
9 except IntegrityError:

```

To:

```

1 try:
2     user = User.create(
3         cd['name'],
4         cd['email'],
5         cd['password'],
6         cd['last_4_digits'],
7         stripe_id=''
8     )
9
10    if customer:
11        user.stripe_id = customer.id
12        user.save()
13
14 except IntegrityError:

```

Here, we broke up the single insert on the database into an insert and then an update.

Running the test still fails, though.

Since we are not passing in the `stripe_id` when we initially create the user, we have to change the `test_registering_new_user_returns_successfully()` test to not expect that. In fact, let's remove all the database mocks from that test, because in a minute we are going to start adding some transaction management.

As a generally rule, it's best not to use database mocks when testing code that directly manages or uses transactions.

Why? This is because the mocks will effectively ignore all transaction management, and thus subtle defects can often slide into play with the developer thinking that the code is well tested and free of defects.

After we take the database mocks out of `test_registering_new_user_returns_successfully()` the test looks like this:

```

1 def get_mock_cust():
2
3     class mock_cust():
4
5         @property
6         def id(self):
7             return 1234
8
9     return mock_cust()
10
11 @mock.patch('payments.views.Customer.create',
12             return_value=get_mock_cust())
13 def test_registering_new_user_returns_succesfully(self,
14         stripe_mock):
15
16     self.request.session = {}
17     self.request.method = 'POST'
18     self.request.POST = {
19         'email': 'python@rocks.com',
20         'name': 'pyRock',
21         'stripe_token': '...',
22         'last_4_digits': '4242',
23         'password': 'bad_password',
24         'ver_password': 'bad_password',
25     }
26
27     resp = register(self.request)
28
29     self.assertEqual(resp.content, b'')
30     self.assertEqual(resp.status_code, 302)
31
32     users = User.objects.filter(email="python@rocks.com")
33     self.assertEqual(len(users), 1)
34     self.assertEqual(users[0].stripe_id, '1234')
35
36 def get_MockUserForm(self):
37
38     from django import forms
39
40     class MockUserForm(forms.Form):

```

```

39
40     def is_valid(self):
41         return True
42
43     @property
44     def cleaned_data(self):
45         return {
46             'email': 'python@rocks.com',
47             'name': 'pyRock',
48             'stripe_token': '...',
49             'last_4_digits': '4242',
50             'password': 'bad_password',
51             'ver_password': 'bad_password',
52         }
53
54     def addError(self, error):
55         pass
56
57     return MockUserForm()

```

In the above test we want to explicitly make sure that the `stripe_id` **IS** being set, so we mocked the `Customer.create()` function and had it return a dummy class that always provides 1234 for its `id`. That way we can assert that the new user in the database has the `stripe_id` of 1234.

All good:

```

1 Creating test database for alias 'default'...
2 .....
3 .
4 .....
5 -----
6 Ran 30 tests in 0.505s
7
8 OK
9 Destroying test database for alias 'default'...

```

Time for a commit.

Handling Unpaid Users

At this point we are now letting users register even if Stripe is down.

In effect this means we are letting the user in for free. Obviously if everybody starts getting in for free, it won't be long before our site tanks, so let's fix that.

WARNING: The solution we're about to propose isn't the most elegant, but we need an example to use for database transactions that fits into the our current Project! Plus, we'll give you a chance to fix it later in the exercises.

The proposed solution to this endeavor is to create a table of unpaid users so we can harass those users until they cough up their credit cards. Or, if you want to be a bit more politically correct: So our account management system can help the customers with any credit card billing issues they may have had.

To do that, we will create a new table called *unpaid_users* with two columns: the user email and a timestamp used to keep track of when we last contacted this user to update billing information.

First let's create a new model in *payments/models.py*:

```
1 from django.utils import timezone
2
3 class UnpaidUsers(models.Model):
4     email = models.CharField(max_length=255, unique=True)
5     last_notification = models.DateTimeField(default=timezone.now())
```

NOTE: We're intentionally leaving off foreign key constraints for now. We may come back to it later.

Further NOTE: `django.utils.timezone.now()` functions the same as `datetime.now()` except that `timezone` is always timezone aware. This will prevent Django from complaining about naive timezone.

Further Further NOTE: After creating the new model, you'll want to run `./manage.py syncdb`. This will show some different output than you're used to as Django 1.7 now has migration support. For the time being we will stick with `syncdb`, however there is an upcoming chapter on Migrations which will explain Django 1.7's migrations.

Now create the test. We want to ensure that the `UnpaidUsers()` table is populated if/when Stripe is down. Let's modify our `payments/testViews/test_registering_user_when_stripe_is_down` test. All we need to do is add a couple of asserts at the end of that test:

```
1 # check the associated table got updated.
2 unpaid = UnpaidUsers.objects.filter(email="python@rocks.com")
3 self.assertEqual(len(unpaid), 1)
4 self.assertIsNotNone(unpaid[0].last_notification)
```

Make sure to update the imports:

```
1 from payments.models import User, UnpaidUsers
```

This test asserts that we got a new row in the `UnpaidUsers()` table and it has a `last_notification` timestamp. Run the test... watch it fail.

Now let's fix the code.

Of course we have to populate the table during registration if a user fails to validate their card through Stripe. So let's adjust `payments.views.registration` as is shown below:

```
1 def register(request):
2
3     ... snip ...
4
5     cd = form.cleaned_data
6     try:
7         user = User.create(cd['name'], cd['email'],
8                             cd['password'],
9                             cd['last_4_digits'],
10                            stripe_id='')
11
12         if customer:
13             user.stripe_id = customer.id
14             user.save()
15         else:
16             UnpaidUsers(email=cd['email']).save()
17
18     except IntegrityError:
19         import traceback
20         form.addError(cd['email'] + ' is already a member' +
21                       traceback.format_exc())
22
23     user = None
```

```
21         else:
22             request.session['user'] = user.pk
23             return HttpResponseRedirect('/')
24
25     ... snip ...
```

Again, make sure to update the imports:

```
1 from payments.models import User, UnpaidUsers
```

Now re-run the tests... and they should all pass. We're golden.

Right? Not exactly.

Improved Transaction Management

If you've ever devoted much time to Django database transaction management, you know how confusing it can get. In the past (prior to Django 1.6), the documentation provided quite a bit of depth, but understanding only came through building and experimenting.

For example, there was a plethora of decorators to work with, like: `commit_on_success`, `commit_manually`, `commit_unless_managed`, `rollback_unless_managed`, `enter_transaction_management`, `leave_transaction_management`, just to name a few. Fortunately, with Django 1.6 (or greater, of course) that all goes out the door. You only really need to know about a few functions for now, which we'll get to in just a few seconds. First, we'll address these topics:

- **What is transaction management?**
- **What's wrong with transaction management prior to Django 1.6?**

Before jumping into:

- **What's right about transaction management in Django 1.6?**

And then dealing with a detailed example:

- **Stripe Example**
- **Transactions**
 - **The recommended way**
 - **Using a decorator**
 - **Transaction per HTTP Request**
- **SavePoints**
- **Nested Transactions**

What is a transaction?

According to [SQL-92](#), “An SQL-transaction (sometimes simply called a “transaction”) is a sequence of executions of SQL-statements that is atomic with respect to recovery”. In other words, all the SQL statements are executed and committed together. Likewise, when rolled back, all the statements get rolled back together.

For example:

```
1 # START
2 note1 = Note(title="my first note", text="Yay!")
3 note2 = Note(title="my second note", text="Whee!")
4 note1.save()
5 Note2.save()
6 # COMMIT
```

A transaction is a single unit of work in a database, and that single unit of work is demarcated by a start transaction and then a commit or an explicit rollback.

What's wrong with transaction management prior to Django 1.6?

In order to fully answer this question, we must address how transactions are dealt with in the database, client libraries, and within Django.

Databases

Every statement in a database has to run in a transaction, even if the transaction includes only one statement.

Most databases have an AUTOCOMMIT setting, which is usually set to True as a default. This AUTOCOMMIT wraps every statement in a transaction that is immediately committed if the statement succeeds. You can also manually call something like `START_TRANSACTION` which will temporarily suspend the AUTOCOMMIT until you call `COMMIT_TRANSACTION` or `ROLLBACK`.

However, the takeaway here is that the AUTOCOMMIT setting applies an implicit commit after each statement.

Client Libraries

Then there are the Python **client libraries** like `sqlite3` and `mysqldb`, which allow Python programs to interface with the databases themselves. Such libraries follow a set of standards for how to access and query databases. That standard, DB API 2.0, is described in [PEP 249](#). While it may make for some slightly dry reading, an important takeaway is that PEP 249 states that the database AUTOCOMMIT should be *OFF* by default.

This clearly conflicts with what's happening within the database:

- SQL statements always have to run in a transaction, which the database generally opens for you via AUTOCOMMIT.
- However, according to PEP 249, this should not happen.
- Client libraries must mirror what happens within the database, but since they are not allowed to turn AUTOCOMMIT on by default, they simply wrap your SQL statements in a transaction, just like the database.

Okay. Stay with me a little longer...

Django

Enter Django. **Django** also has something to say about transaction management. In Django 1.5 and earlier, Django basically ran with an open transaction and auto-committed that transaction when you wrote data to the database. So every time you called something like `model.save()` or `model.update()`, Django generated the appropriate SQL statements and committed the transaction.

Also in Django 1.5 and earlier, it was recommended that you used the `TransactionMiddleware` to bind transactions to HTTP requests. Each request was given a transaction. If the response returned with no exceptions, Django would commit the transaction, but if your view function threw an error, `ROLLBACK` would be called. In effect, this turned off `AUTOCOMMIT`. If you wanted standard, database-level autocommit style transaction management, you had to manage the transactions yourself - usually by using a transaction decorator on your view function such as `@transaction.commit_manually`, or `@transaction.commit_on_success`.

Take a breath. Or two.

What does this mean?

Yes, there is a lot going on there, and it turns out most developers just want the standard database level autocommits - meaning transactions stay behind the scenes, doing their thing, until you need to manually adjust them.

What's right about transaction management in Django 1.6 and above?

Now, welcome to Django 1.6 and above. Simply remember that in Django 1.6 (or greater), you use database `AUTOCOMMIT` and manage transactions manually when needed. Essentially, we have a much simpler model that basically does what the database was designed to do in the first place!

Coming back to our earlier question: Is our registration function really Golden? We just want to hold user's info and then re-verify it with Stripe.

We wrote a failing test, then made it pass. And now it's time for the refactor portion of TDD.

Thinking about transactions and keeping in mind that by default Django gives us *AUTOCOMMIT* behavior for our database, let's stare at the code a little longer.

```
1 cd = form.cleaned_data
2 try:
3     user = User.create(cd['name'], cd['email'], cd['password'],
4                       cd['last_4_digits'], stripe='')
5
6     if customer:
7         user.stripe_id = customer.id
8         user.save()
9     else:
10        UnpaidUsers(email=cd['email']).save()
11
12 except IntegrityError:
13     import traceback
14     form.addError(cd['email'] + ' is already a member' +
15                  traceback.format_exc())
```

Did you spot the issue? We would just want to hold their info and then re-verify it

What happens if the `UnpaidUsers(email=cd['email']).save()` line fails?

Well, then you have a user registered in the system who never verified their credit card while the system assumes they have. In other words, somebody got in for free. Not good. So this is the perfect case for when to use a transaction, because we want it all or nothing here. In other words, we only want one of two outcomes:

1. User is created (in the database) and has a `stripe_id`.

2. User is created (in the database), doesn't have a `stripe_id` and has an associated row in the `UnpaidUsers` table with the same email address as the User.

This means we want the two separate database statements to either both commit or both rollback. A perfect case for the humble transaction. There are many ways we can achieve this.

First, let's write some tests to verify things behave the way we want them to:

```
1 @mock.patch('payments.models.UnpaidUsers.save',
2             side_effect=IntegrityError)
3 def test_registering_user_when_stripe_is_down_all_or_nothing(self,
4   save_mock):
5
6     #create the request used to test the view
7     self.request.session = {}
8     self.request.method = 'POST'
9     self.request.POST = {
10         'email': 'python@rocks.com',
11         'name': 'pyRock',
12         'stripe_token': '...',
13         'last_4_digits': '4242',
14         'password': 'bad_password',
15         'ver_password': 'bad_password',
16     }
17
18     #mock out stripe and ask it to throw a connection error
19     with mock.patch(
20         'stripe.Customer.create',
21         side_effect=socket.error("can't connect to stripe")
22     ) as stripe_mock:
23
24         #run the test
25         resp = register(self.request)
26
27         #assert there is no new record in the database
28         users = User.objects.filter(email="python@rocks.com")
29         self.assertEqual(len(users), 0)
30
31         #check the associated table has no updated data
```

```

31         unpaid =
            UnpaidUsers.objects.filter(email="python@rocks.com")
32         self.assertEqual(len(unpaid), 0)

```

This test is more or less a copy of `test_register_user_when_stripe_is_down()`, except we added the `@mock.patch` decorator that throws an `IntegrityError` when `save()` is called on `UnpaidUsers`.

Run the test:

```

1  Creating test database for alias 'default'...
2  .....F
3  .....
4  .
5  .....
6  =====
7  FAIL: test_registering_user_when_stripe_is_down_all_or_nothing
    (tests.payments.testviews.RegisterPageTests)
8  -----
9  Traceback (most recent call last):
10     File
        "/Users/michaelherman/Documents/repos/realpython/book3-exercises/_chapters/ch
        line 1201, in patched
11     return func(*args, **kwargs)
12     File
        "/Users/michaelherman/Documents/repos/realpython/book3-exercises/_chapters/ch
        line 273, in
            test_registering_user_when_stripe_is_down_all_or_nothing
13     self.assertEqual(len(users), 0)
14  AssertionError: 1 != 0
15
16  -----
17  Ran 31 tests in 1.381s
18
19  FAILED (failures=1)
20  Destroying test database for alias 'default'...

```

Nice. It failed. Seems funny to say that, but it's exactly what we wanted. And the error message tells us that the User is indeed being stored in the database; we don't want that. Have no fear, transactions to the rescue...

Creating the transaction in Django

There are actually several ways to create the transaction in Django 1.6. Let's go through a couple.

The recommended way

According to Django 1.6 [documentation](#), `atomic` can be used as both a decorator or as a context_manager. So if we use it as a context manager, the code in our register function would look like this:

```
1 def register(request):
2
3     ... snip ...
4
5     cd = form.cleaned_data
6     try:
7         with transaction.atomic():
8             user = User.create(cd['name'], cd['email'],
9                               cd['password'],
10                                cd['last_4_digits'],
11                                stripe_id="")
12
13             if customer:
14                 user.stripe_id = customer.id
15                 user.save()
16             else:
17                 UnpaidUsers(email=cd['email']).save()
18
19     except IntegrityError:
20         import traceback
21         form.addError(cd['email'] + ' is already a member' +
22                      traceback.format_exc())
23         user = None
24     else:
25         request.session['user'] = user.pk
26         return HttpResponseRedirect('/')
27
28     ... snip ...
```

Add the import:

```
1 from django.db import transaction
```

Note the line with `transaction.atomic():`. All code inside that block will be executed inside a transaction. Re-run our tests, and they all pass!

Using a decorator

We can also try adding `atomic` as a decorator. But if we do and rerun our tests... they fail with the same error we had before putting any transactions in at all!

Why is that?

Why didn't the transaction roll back correctly? The reason is because `transaction.atomic` is looking for some sort of `DatabaseError` and, well, we caught that error (e.g., the `IntegrityError` in our try/except block), so `transaction.atomic` never saw it and thus the standard `AUTOCOMMIT` functionality took over.

But removing the try/except will cause the exception to just be thrown up the call chain and most likely blow up somewhere else, so we can't do that either.

The trick is to put the atomic context manager inside of the try/except block, which is what we did in our first solution.

Looking at the correct code again:

```
1 from django.db import transaction
2
3 try:
4     with transaction.atomic():
5         user = User.create(cd['name'], cd['email'], cd['password'],
6                             cd['last_4_digits'], stripe_id="")
7
8         if customer:
9             user.stripe_id = customer.id
10            user.save()
11        else:
12            UnpaidUsers(email=cd['email']).save()
13
14 except IntegrityError:
15     import traceback
16     form.addError(cd['email'] + ' is already a member' +
17                  traceback.format_exc())
```

When `UnpaidUsers` fires the `IntegrityError`, the `transaction.atomic()` `context_manager` will catch it and perform the rollback. By the time our code executes in the exception handler (e.g., the `form.addError` line), the rollback will be complete and we can safely make database calls if necessary. Also note any database calls before or after the `transaction.atomic()` `context_manager` will be unaffected regardless of the final outcome of the `context_manager`.

Transaction per HTTP Request

Django < 1.6 (like 1.5) also allows you to operate in a “Transaction per request” mode. In this mode, Django will automatically wrap your view function in a transaction. If the function throws an exception, Django will roll back the transaction, otherwise it will commit the transaction.

To get it set up you have to set `ATOMIC_REQUEST` to `True` in the database configuration for each database that you want to have this behavior. In our *settings.py* we make the change like this:

```
1 DATABASES = {
2     'default': {
3         'ENGINE': 'django.db.backends.sqlite3',
4         'NAME': os.path.join(SITE_ROOT, 'test.db'),
5         'ATOMIC_REQUEST': True,
6     }
7 }
```

But in practice this just behaves exactly as if you put the decorator on your view function yourself, so it doesn’t serve our purposes here. It is however worthwhile to note that with both `AUTOMIC_REQUESTS` and the `@transaction.atomic` decorator it is possible to still catch and then handle those errors after they are thrown from the view. In order to catch those errors you would have to implement some custom middleware, or you could override [urls.handler](#) or make a [500.html template](#).

SavePoints

We can also further break down transactions into savepoints. Think of savepoints as partial transactions. If you have a transaction that takes four database statements to complete, you could create a savepoint after the second statement. Once that savepoint is created, then if the 3rd or 4th statements fails you can do a partial rollback, getting rid of the 3rd and 4th statement but keeping the first two.

It's basically like splitting a transaction into smaller lightweight transactions, allowing you to do partial rollbacks or commits. But do keep in mind if the main transaction were to get rolled back (perhaps because of an `IntegrityError` that gets raised but not caught), then all savepoints will get rolled back as well.

Let's look at an example of how savepoints work:

```
1 @transaction.atomic()
2 def save_points(self, save=True):
3
4     user = User.create('jj', 'inception', 'jj', '1234')
5     sp1 = transaction.savepoint()
6
7     user.name = 'staring down the rabbit hole'
8     user.stripe_id = 4
9     user.save()
10
11     if save:
12         transaction.savepoint_commit(sp1)
13     else:
14         transaction.savepoint_rollback(sp1)
```

Here the entire function is in a transaction. After creating a new user we create a savepoint and get a reference to the savepoint. The next three statements:

```
1 user.name = 'staring down the rabbit hole'
2 user.stripe_id = 4
3 user.save()
```

Are not part of the existing savepoint, so they stand the potential of being part of the next `savepoint_rollback`, or `savepoint_commit`. In the case of a `savepoint_rollback`. The line `user = User.create('jj', 'inception', 'jj', '1234')` will still be committed to the database even though the rest of the updates won't.

Put in another way, these following two tests describe how the savepoints work:

```

1 def test_savepoint_rollbacks(self):
2
3     self.save_points(False)
4
5     #verify that everything was stored
6     users = User.objects.filter(email="inception")
7     self.assertEqual(len(users), 1)
8
9     #note the values here are from the original create call
10    self.assertEqual(users[0].stripe_id, '')
11    self.assertEqual(users[0].name, 'jj')
12
13
14 def test_savepoint_commit(self):
15     self.save_points(True)
16
17     #verify that everything was stored
18     users = User.objects.filter(email="inception")
19     self.assertEqual(len(users), 1)
20
21     #note the values here are from the update calls
22     self.assertEqual(users[0].stripe_id, '4')
23     self.assertEqual(users[0].name, 'staring down the rabbit hole')

```

After we commit or rollback a savepoint, we can continue to do work in the same transaction, and that work will be unaffected by the outcome of the previous savepoint.

For example, if we update our `save_points` function as such:

```

1 @transaction.atomic()
2 def save_points(self, save=True):
3
4     user = User.create('jj', 'inception', 'jj', '1234')
5     sp1 = transaction.savepoint()
6
7     user.name = 'staring down the rabbit hole'
8     user.save()
9
10    user.stripe_id = 4
11    user.save()
12

```

```
13     if save:
14         transaction.savepoint_commit(sp1)
15     else:
16         transaction.savepoint_rollback(sp1)
17
18     user.create('limbo', 'illbehere@forever', 'mind blown',
19               '1111')
```

Now regardless of whether `savepoint_commit` or `savepoint_rollback` was called, the “limbo” user will still be created successfully, unless something else causes the entire transaction to be rolled-back.

Nested Transactions

In addition to manually specifying savepoints with `savepoint()`, `savepoint_commit`, and `savepoint_rollback`, creating a nested Transaction will automatically create a savepoint for us, and roll it back if we get an error.

Extending our example a bit further we get:

```
1 @transaction.atomic()
2 def save_points(self, save=True):
3
4     user = User.create('jj', 'inception', 'jj', '1234')
5     sp1 = transaction.savepoint()
6
7     user.name = 'staring down the rabbit hole'
8     user.save()
9
10    user.stripe_id = 4
11    user.save()
12
13    if save:
14        transaction.savepoint_commit(sp1)
15    else:
16        transaction.savepoint_rollback(sp1)
17
18    try:
19        with transaction.atomic():
20            user.create('limbo', 'illbehere@forever', 'mind blown',
21                       '1111')
22            if not save: raise DatabaseError
23    except DatabaseError:
24        pass
```

Here we can see that after we deal with our savepoints, we are using the `transaction.atomic` context manager to encase our creation of the “limbo” user. When that context manager is called, it is in effect creating a savepoint (because we are already in a transaction) and that savepoint will be committed or rolled-back upon exiting the context manager.

Thus the following two tests describe their behavior here:

```
1 def test_savepoint_rollbacks(self):
2
```

```

3     self.save_points(False)
4
5     #verify that everything was stored
6     users = User.objects.filter(email="inception")
7     self.assertEqual(len(users), 1)
8
9     #savepoint was rolled back so we should have original values
10    self.assertEqual(users[0].stripe_id, '')
11    self.assertEqual(users[0].name, 'jj')
12
13    #this save point was rolled back because of DatabaseError
14    limbo = User.objects.filter(email="illbehere@forever")
15    self.assertEqual(len(limbo), 0)
16
17
18    def test_savepoint_commit(self):
19        self.save_points(True)
20
21        #verify that everything was stored
22        users = User.objects.filter(email="inception")
23        self.assertEqual(len(users), 1)
24
25        #savepoint was committed
26        self.assertEqual(users[0].stripe_id, '4')
27        self.assertEqual(users[0].name, 'staring down the rabbit hole')
28
29        #save point was committed by exiting the context_manager
30        #without an exception
31        limbo = User.objects.filter(email="illbehere@forever")
32        self.assertEqual(len(limbo), 1)

```

So in reality you can use either atomic or savepoint to create savepoints inside a transaction, but with atomic you don't have to *worry* explicitly about the commit/rollback, where as with savepoint you have full control over when that happens.

Completing the Front-end

Fire up the application, disconnect your Internet, and then try to register a user. What happens? You should still get a failure (or a very strange error), because we need to update the form on the front-end for handling errors.

In *static/application.js* there is an existing function `Stripe.createToken()` that grabs the form inputs, including the stripe token so we don't have to store credit card numbers in the back-end:

```
1 Stripe.createToken(card, function(status, response) {
2   if (status === 200) {
3     console.log(status, response);
4     $("#credit-card-errors").hide();
5     $("#last_4_digits").val(response.card.last4);
6     $("#stripe_token").val(response.id);
7     form.submit();
8   } else {
9     $("#stripe-error-message").text(response.error.message);
10    $("#credit-card-errors").show();
11    $("#user_submit").attr("disabled", false);
12  }
13 });
```

Let's update this to allow for signup even if Stripe is down:

```
1 Stripe.createToken(card, function(status, response) {
2   if (status === 200) {
3     console.log(status, response);
4     $("#credit-card-errors").hide();
5     $("#last_4_digits").val(response.card.last4);
6     $("#stripe_token").val(response.id);
7   }
8   //always submit form even with errors
9   form.submit();
10 });
```

This will get then send the POST method to the `payments.views.register()` function, which will in turn call our recently updated `Customer.create()` function. However now we are likely to get a different error, in fact we will probably get two different errors:

- Connection Errors - `socket.error` or `stripe.error.APIConnectionError`

- Invalid Request Errors - `stripe.error.InvalidRequestError`

Thus we can change the `payments.views.create()` function to catch each of the exceptions like:

```
1 except (socket.error, stripe.APIConnectionError,  
2         stripe.InvalidRequestError):  
3     return None
```

The updated function:

```
1 class Customer(object):  
2  
3     @classmethod  
4     def create(cls, billing_method="subscription", **kwargs):  
5         try:  
6             if billing_method == "subscription":  
7                 return stripe.Customer.create(**kwargs)  
8             elif billing_method == "one_time":  
9                 return stripe.Charge.create(**kwargs)  
10        except (socket.error, stripe.APIConnectionError,  
11                stripe.InvalidRequestError):  
12            return None
```

Test it out now. It should work.

Conclusion

If you have had any previous experience with earlier versions of Django, you can see how much simpler the transaction model is. Also having auto-commit on by default is a great example of “sane defaults” that Django and Python both pride themselves in delivering. For many systems you won’t need to deal directly with transactions, just let auto-commit do its work. But when you do, hopefully this chapter will have given you the information you need to manage transactions in Django like a pro.

Further here is a quick list of reminders to help you remember the important stuff:

Important Transaction Concepts

AUTOCOMMIT

Functions at the database level; implicitly commit after each SQL statement..

```
1 START TRANSACTION
2 SELECT * FROM DEV_JOBS WHERE PRIMARY_SKILL = 'PYTHON'
3 END TRANSACTION
```

Atomic Decorator

Django >= 1.6 based transaction management has one main API which is `atomic`. Using `atomic` wraps a code block in a db transaction.

```
1 with transaction.atomic():
2     user1.save()
3     unpaidUser1.save()
```

Transaction per http Request

This causes Django to automatically create a transaction to wrap each view function call. To activate add `ATOMIC_REQUEST` to your database config in `settings.py`

```
1 DATABASES = {
2     'default': {
3         'ENGINE': 'django.db.backends.sqlite3',
4         'NAME': os.path.join(SITE_ROOT, 'test.db'),
5         'ATOMIC_REQUEST': True,
```



```
6     }
7 }
```

Savepoints

Savepoints can be thought of as partial transactions. They allow you to save/rollback part of a transaction instead of the entire transaction. For example:

```
1 @transaction.atomic()
2 def save_points(self, save=True):
3
4     user = User.create('jj', 'inception', 'jj', '1234')
5     sp1 = transaction.savepoint()
6
7     user.name = 'staring down the rabbit hole'
8     user.stripe_id = 4
9     user.save()
10
11     if save:
12         transaction.savepoint_commit(sp1)
13     else:
14         transaction.savepoint_rollback(sp1)
```

Exercises.

1. For the final example of `savepoints()`, what would happen if you removed the `try/except` altogether?

For reference, the code we're referring to is below (minus the `try/except`):

```
1 @transaction.atomic()
2 def save_points(self, save=True):
3
4     user = User.create('jj', 'inception', 'jj', '1234')
5     sp1 = transaction.savepoint()
6
7     user.name = 'staring down the rabbit hole'
8     user.save()
9
10    user.stripe_id = 4
11    user.save()
12
13    if save:
14        transaction.savepoint_commit(sp1)
15    else:
16        transaction.savepoint_rollback(sp1)
17
18    with transaction.atomic():
19        user.create('limbo', 'illbehere@forever', 'mind blown',
20                  '1111')
21        if not save: raise DatabaseError
```

Verify your expectation with a test or two. Can you explain why that happened? Perhaps these lines from the Django Documentation may help you understand it more clearly.

Under the hood, Django's transaction management code:

- opens a transaction when entering the outermost atomic block;
 - creates a savepoint when entering an inner atomic block;
 - releases or rolls back to the savepoint when exiting an inner block;
 - commits or rolls back the transaction when exiting the outermost block.
2. Build your own transaction management system. Just joking. There's no need to reinvent the wheel. Instead, you could read through the complete Django documentation

on the new transaction management features [here](#) if you really, really wanted to.

Chapter 8

Building a Membership Site

Up until now we have covered the nitty gritty of unit testing; how to do TDD; git branching for teams; upgrading to Django 1.8, Python 3 and PostgreSQL as well as the latest in Django Database Transactions. While these are all necessary tools and techniques that modern day web developers should have in their tool belt, you could say we haven't built much yet.

It's time to make something cool

That's all about to change. Modern software development is roughly split between tools and techniques, front and back-end development. So we have covered quite a bit in terms of tools and techniques and some back-end development strategies as well. At this point it's time to switch gears a bit and focus on front-end development. While many companies still have separate front and back-end developers, the practice of "full-stack" development is becoming more and more the norm.

Full-stack developers can develop complete features from the back to the front, from database design to Javascript animation. About the only thing you wouldn't include in the repertoire of a "full-stack" developer is design.

The ability to develop both in a back-end language such as Python as well as THE front-end language of Javascript (vanilla JS, jQuery, AJAX, and even a front-end framework) is extremely beneficial. Even if you end up working for a team where you are solely a "back-end developer", having a solid understanding of front-end concerns can help you develop a better API that makes your front-end counterparts more productive and efficient.

In order to better illustrate the need to be a full-stack developer we are going to take our Django app to the next level and build a Membership Site. We will still focus on a MVP (Minimum Viable Product) but we will discuss how to create the complete feature set. That is to say, the Python/Django back-end, the Javascript front-end and even just enough CSS to get by.

Before we jump into building the features, though, it's helpful to take a step back and document what features we want.

We're not going to go into high vs low fidelity prototyping or the benefits of developing agile and utilizing user stories, because there are plenty of excellent books devoted to that. In fact, check out the previous Real Python course, *Web Development with Python*, to learn more about the how and why you should utilize user stories in the development process, specifically in the *Flask: Behavior-Driven Development with Behave* chapter.

We'll keep it practical. Let's look at a few of the user stories that we're going to implement in the coming chapters to paint a picture of where we are headed.

A Membership Site

Let's look at what we have before diving into what we're going to add. Currently we have the following implemented:

- Static homepage
- User login/logout (session management)
- User Registration
- Stripe integration supporting one-time and subscription payments
- A Contact Us Page
- About Us Page (using [Django flatpages](#))

Not too shabby, but also not something that's going to make the front page of Hacker News either. Let's see what we can do to flesh this out to into something a bit more interesting.

Let's start out with an overview of what our MVP is and why we are creating it. To make it more interesting let's pretend we are building a membership site for Star Wars lovers. Let's call our wonderful little membership site [Mos Eisley's Cantina](#) or just MEC for short.

Product Vision

Mos Eisley's Cantina (MEC) aims to be the premiere online membership site for Star Wars Enthusiasts. Being a paid site, MEC will attract only the most loyal Star Wars fans and will encourage highly entertaining debate and discussion about the entirety of the Star Wars Universe. A unique polling system will allow us to once and for all decide on important questions such as who is truly the most powerful Jedi, are the separatists good or evil, and seriously what is the deal with Jar Jar Binks? In addition, MEC will provide news and videos of current Star Wars-related happenings and in general be the best place to discuss Star Wars in the entire galaxy.

That's the vision; we're building a real geek site. It probably won't make a dime, but that's not the point. *The techniques we are using here will be applicable to any membership site.* Now that we know what we are building, let's list out a few user stories so we have something to focus on.

The User Stories

US1: Main Page

Upon arriving at the site the “youngling” (or unregistered user) will be greeted with a beautiful overview page describing the great features of MEC with a large sign up button right-smack-dab in the middle of the page. Toward the bottom of the page there should be ‘about’ and ‘contact us’ links so youglings can find out more information about the site. Finally there should be a login button to allow returning “padwans” (or registered users) to log into the site.

US2: Registration

After clicking the sign up button the “applicant padwan” (user wishing to sign up but not yet verified by credit card) will be presented with a screen to collect basic registration information including credit card information. Credit card information should be immediately processed through Stripe, after which the “applicant padwan” should be upgraded to “padwan” status and redirected to the Members Home Page.

US3: Members Home Page

The main page where returning “padwans” are greeted. This members page is a place for announcements and to list current happenings. It should be the single place a user needs to go to know all of the latest news at MEC.

US4: User Polls

Determining the truth of the galaxy and balancing the force are both very important goals at MEC. As such, MEC should provide the functionality to poll “padwans” and correlate the results in order to best determine or predict the answers to important topics of the Star Wars galaxy. This includes questions like Kit Fisto vs Aayla Secura, who has the best dreadlocks? Or who would win in a fight, C3PO or R2-D2? Results should also be displayed to the “padwans” so that all shall know the truth.

US5: Galactic Map

A map displaying the location of each of the registered “padwans”. Useful for physical mee-tups, for the purpose of real life re-enactments of course. By Galactic here we mean global.

This map should provide a graphic view of who is where and allow for zooming in on certain locations.

We could go on for days, but that's enough to fill a course. In the coming chapters we are going to look at each of these user stories and try to implement them. US2 (Registration) is pretty much done, but the others all need to be implemented. We've arranged them in order of increasing difficulty so we can build on the knowledge learned by implementing each of the user stories.

Without further ado... the next chapter will cover **US1**. See you on the next page.

Chapter 9

Bootstrap 3 and Best Effort Design

As we mentioned in the last chapter, traditional web design is usually not expected of most full-stack developers. If you can design, then more power to you, but for the rest of us artistically challenged developers there is [Bootstrap](#). This fantastic library, originally released by the amazing design team at Twitter, makes it so easy to style a website that even somebody with absolutely zero artistic ability can make a site look presentable. Granted, your site is probably not going to win any awards for design excellence, but it won't be embarrassing either. With that in mind, let's look at what we can do to make our site look a little bit better.

Start with the User Story

Last chapter we laid out several User Stories that we will be working through in the coming chapters to develop *MEC*, our awesome Star Wars membership site. To refresh your memory, here's the first user story that we will work through in this chapter:

US1: Main Page

Upon arriving at the site the “youngling” (or unregistered user) will be greeted with a beautiful overview page describing the great features of MEC with a large sign up button right-smack-dab in the middle of the page. Toward the bottom of the page there should be ‘about’ and ‘contact us’ links so youglings can find out more information about the site. Finally there should be a login button to allow returning “padwans” (or registered users) to log into the site.

In other words, we need something that looks cool and gets the “youngling” to click on the signup button (convert).

If we had to do that in straight HTML and CSS, it might take ages. But with Bootstrap we can do it pretty quickly. The plan then is to create a main page with a nice logo, a couple of pictures and descriptions of what the site is about and a nice big signup button to grab the user's (err youngling's) attention.

Let's get started.

Overview of what we have

Although we haven't talked about it much in this course, we are using the simple application that you created in the second course, *Web Development with Python*. You created the application with Bootstrap version 2. Currently Bootstrap is on version 3 (3.2.0 as of writing) and quite a bit has changed between 2 and 3. The most significant change is the “Mobile First” ideology of Bootstrap 3. Basically responsive design is enabled from the start and the idea is that everything should work on a mobile device first and foremost. This is a reflection of the current and growing popularity of mobile.

While we could “upgrade” to Bootstrap 3, however since we are going to significantly redesign the site, let's start over (more or less) with the front end. If you have a large site with a lot of pages, it's not recommended to go this route, but since we only have a few pages, we can get away with it.

Installing Bootstrap 3

The first thing to do is grab Bootstrap file. So let's download it locally and chuck it in our static directory. Before that, go ahead and remove all files from the "django_ecommerce/static" directory except for *application.js*.

Keep in mind that we could serve the two main Bootstrap files, *bootstrap.min.css* and *bootstrap.min.js* from a Content Delivery Network (CDN). When you are developing locally, it's best to download the files on your file system, though, in case you're trying to develop/design and you lose Internet access. There are various benefits to utilizing a CDN when you deploy your app to a production server, which we will detail in a later chapter.

1. Download the Bootstrap distribution files from [here](#).
2. Unzip the files and add the "css", "fonts", and "js" directories to the "django_ecommerce/static" file directory. Add the *application.js* file to the "js" directory. When it's all said and done, your directory structure should look like this:

```
1  .
2  css
3      bootstrap-theme.css
4      bootstrap-theme.css.map
5      bootstrap-theme.min.css
6      bootstrap.css
7      bootstrap.css.map
8      bootstrap.min.css
9  fonts
10     glyphs-halflings-regular.eot
11     glyphs-halflings-regular.svg
12     glyphs-halflings-regular.ttf
13     glyphs-halflings-regular.woff
14  js
15     application.js
16     bootstrap.js
17     bootstrap.min.js
```

3. Another critical file that we must not forget to add is jQuery. As of this writing 1.11.1 is the most recent version. Download the minified version - e.g., *jquery-1.11.1.min.js* - from the jQuery download [page](#) site, and add it to the "js" folder.

NOTE: On a side note, do keep in mind that if you have a “real” site that’s currently using a version of jQuery earlier than 1.9, and you’d like to upgrade to the latest version, you should take the time to look through the release notes and also look at the [jQuery Migrate plugin](#). In our case we are actually upgrading from version 1.6x; however, since our example site only makes very little use of jQuery (at the moment), it’s not much of an issue. But in a production environment ALWAYS make sure you approach the upgrade carefully and test thoroughly.

4. Say goodbye to the current design. Take one last look at it if you’d like, because we are going to wipe it out.

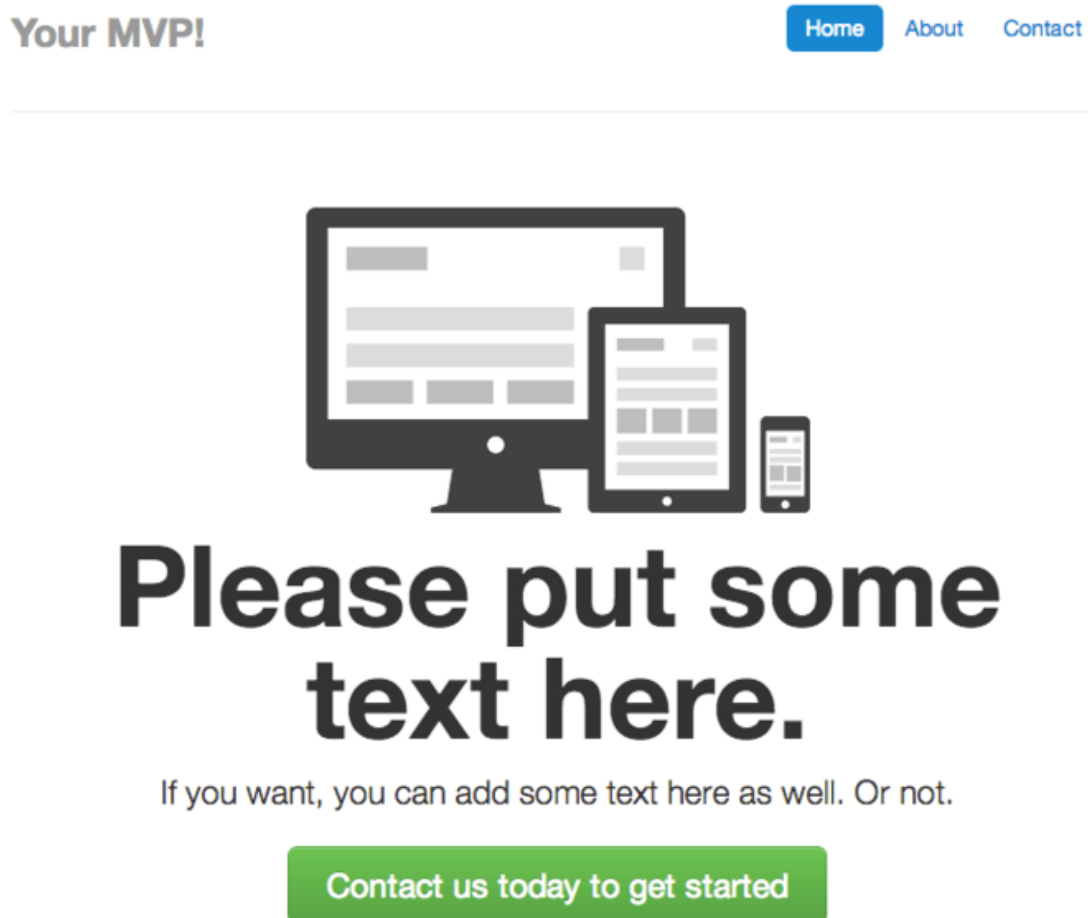


Figure 9.1: Initial look

5. With Bootstrap in the static folder, let’s start with the basic Bootstrap template found

on the [same page](#) that you downloaded Bootstrap from. Take that template and overwrite your *base.html* file:

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="utf-8">
5     <meta http-equiv="X-UA-Compatible" content="IE=edge">
6     <meta name="viewport" content="width=device-width,
7       initial-scale=1">
8     <title>Bootstrap 101 Template</title>
9
10    <!-- Bootstrap -->
11    <link href="css/bootstrap.min.css" rel="stylesheet">
12
13    <!-- HTML5 Shim and Respond.js IE8 support of HTML5 elements
14    and media queries -->
15    <!-- WARNING: Respond.js doesn't work if you view the page via
16    file:// -->
17    <!--[if lt IE 9]>
18      <script
19        src="https://oss.maxcdn.com/html5shiv/3.7.2/html5shiv.min.js"></script>
20      <script
21        src="https://oss.maxcdn.com/respond/1.4.2/respond.min.js"></script>
22    <![endif]-->
23  </head>
24  <body>
25
26    <h1>Hello, world!</h1>
27
28    <script src="https://js.stripe.com/v2/"
29      type="text/javascript"></script>
30    <script type="text/javascript">
31      //<![CDATA[
32      Stripe.publishableKey = '{{ publishable }}';
33      //]]>
34    </script>
35    <!-- jQuery (necessary for Bootstrap's JavaScript plugins) -->
36    <script
37      src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></script>
```

```

31 <!-- Include all compiled plugins (below), or include
    individual files as needed -->
32 <script src="js/bootstrap.min.js"></script>
33 <script src="js/application.js"></script>
34 </body>
35 </html>

```

SEE ALSO: While you're on the "Bootstrap Getting Started" page, check out some of the other [examples](#). These are some basic starter templates that provide the basic Bootstrap components, allowing you to get started faster.

6. Finally, from the *django_ecommerce* directory run `./manage.py runserver` and navigate to the main URL shown - `[http://localhost:8000/](http://localhost:8000/)`. You should see 'Hello, world!'
7. Some of the paths in the above template are not necessarily correct, so let's fix that, then we will put our Star Wars theme on the main page!

The first thing to do is use the Django `{% load static %}` [template tag](#), which will allow us to reference static files by the "STATIC_DIR" entry in our *settings.py* file. After adding `{% load static %}` as the first line in our *base.html* file, we can change the `src` lines that load the CSS and JavaScript files to:

```

1 <link href= "{% static "css/bootstrap.min.css" %}" rel="stylesheet">
2 ...snip...
3 <script src="{% static "js/jquery-1.11.1.min.js" %}"></script>
4 <script src="{% static "js/bootstrap.min.js" %}"></script>
5 <script src="{% static "js/application.js" %}"></script>

```

If you refresh your page, you will see that there is a change in the font, indicating the the CSS file is now loading correctly.

Installation complete. Your final *base.html* file should look like this:

```

1 {% load static %}
2
3 <!DOCTYPE html>
4 <html lang="en">
5   <head>
6     <meta charset="utf-8">
7     <meta http-equiv="X-UA-Compatible" content="IE=edge">

```

```

8     <meta name="viewport" content="width=device-width,
        initial-scale=1">
9     <title>Bootstrap 101 Template</title>
10
11     <!-- Bootstrap -->
12     <link href= "{% static "css/bootstrap.min.css" %}"
        rel="stylesheet">
13
14     <!-- HTML5 Shim and Respond.js IE8 support of HTML5 elements
        and media queries -->
15     <!-- WARNING: Respond.js doesn't work if you view the page via
        file:// -->
16     <!--[if lt IE 9]>
17         <script
            src="https://oss.maxcdn.com/html5shiv/3.7.2/html5shiv.min.js"></script>
18         <script
            src="https://oss.maxcdn.com/respond/1.4.2/respond.min.js"></script>
19     <![endif]-->
20 </head>
21 <body>
22
23     <h1>Hello, world!</h1>
24
25
26     <script src="https://js.stripe.com/v2/"
        type="text/javascript"></script>
27     <script type="text/javascript">
28         //<![CDATA[
29         Stripe.publishableKey = '{{ publishable }}';
30         //]]>
31     </script>
32
33     <!-- jQuery (necessary for Bootstrap's JavaScript plugins) -->
34     <script src="{% static "js/jquery-1.11.1.min.js" %}"></script>
35     <!-- Include all compiled plugins (below), or include
        individual files as needed -->
36     <script src="{% static "js/bootstrap.min.js" %}"></script>
37     <script src="{% static "js/application.js" %}"></script>
38 </body>
39 </html>

```


Making Bootstrap your own

With the installation out of the way, our *base.html* now has the basic setup we need (assuming all went well with the Bootstrap installation, of course). Now we can start customizing.

Page Title

We probably want to change the title since the current title, “Bootstrap 101 Template”, has little to do with Star Wars. You can customize this how you want, but we’ll be using “Mos Eisley’s Cantina” our example.

```
1 <title>Mos Eisley's Cantina</title>
```

Custom Fonts

Let’s also use a custom Star Wars font called [Star Jedi](#).

Custom fonts can be a bit tricky on the web because there are so many different formats you need to support because of cross browser compatibility issues. Basically, you need four different formats of the same font. If you have a TrueType font you can convert it into the four different fonts that you will need with an online font conversion [tool](#). Then you’ll have to create the correct CSS entries.

You can do this conversion on your own if you’d like to practice. Simply download the font from the above URL and then convert it. Or you can grab the fonts already converted in the “chpo8” folder on the exercise [repo](#).

1. Add a new file called *mec.css* in the “static/css” directory. Add the follow styles to start with:

```
1 @font-face {
2   font-family: 'starjedi';
3   /*IE9 */
4   src: url('../fonts/Starjedi.eot');
5   /* Chrome, FF, Opera */
6   src: url('../fonts/Starjedi.woff') format('woff'),
7   /* android, safari, iOS */
8   url('../fonts/Starjedi.ttf') format('truetype'),
9   /* legacy safari */
10  url('../fonts/Starjedi.svg') format('svg');
```

```

11  font-weight: normal;
12  font-style: normal;
13  }
14
15  body {
16      padding-bottom: 40px;
17      background-color: #eee;
18  }
19
20  h1 {
21      font-family: 'starjedi', sans-serif;
22  }
23
24  .center-text {
25      text-align: center;
26  }
27
28  .featurette-divider {
29      margin: 80px 0;
30  }

```

The main thing going on in the CSS file above is the `@font-face` directive. This defines a font-family called `starjedi` (you can call it whatever you want) and specifies the various font files to use based upon the format requested from the browser.

It's important to note that the path to the font is the relative path from the location of the CSS file. Since this is a CSS file, we don't have our Django template tag like `static` available to us.

2. Make sure to add the new *mec.css* to the *base.html* template:

```

1  <!-- Bootstrap -->
2  <link href= "{% static "css/bootstrap.min.css" %}" rel="stylesheet">
3  <!-- custom styles -->
4  <link href= "{% static "css/mec.css" %}" rel="stylesheet">

```

3. Test! Fire up the server. Check out the changes. Our “Hello, world!” is now using the cool *startjedi* font so our site can look a bit more authentic.

Layout

Now let's work on a nice layout. This is where we are going:

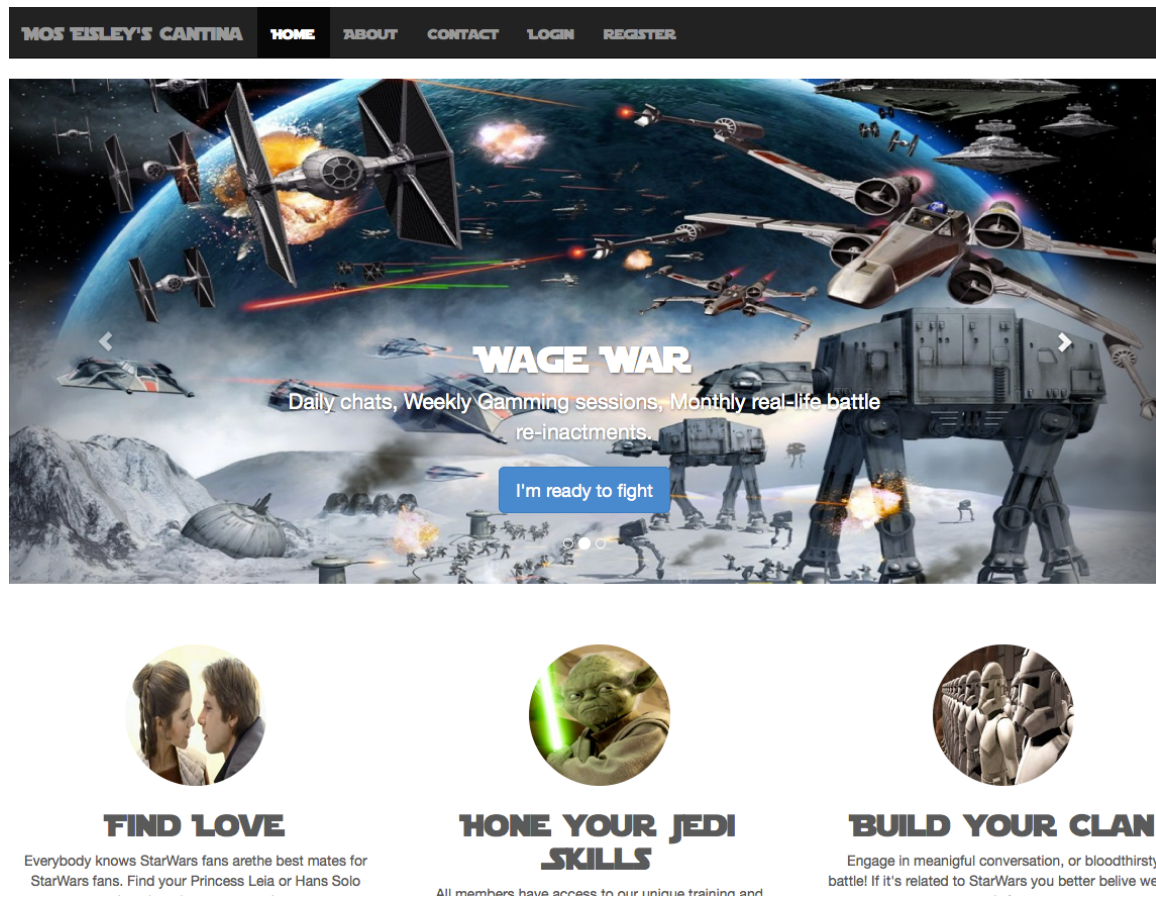


Figure 9.2: main page

Not too shabby for somebody with zero design ability.

Ok, there is a lot going on in the page, so let's break it down a piece at a time (sound familiar?) and look at the implementation for each.

Navbar

Navbars are pretty common these days, and they are a great way to provide quick links for navigation. We already had a navbar on our existing template, but let's put a slightly different one in for the sake of another example. The basic structure of the navbar is the same in Bootstrap 3 as it is in Bootstrap 2, but with a few different classes. The structure should look like this:

```
1 <!-- NAVBAR ===== -->
2 <div class="navbar-wrapper">
3
4   <div class="navbar navbar-inverse navbar-static-top"
5     role="navigation">
6     <div class="container">
7       <div class="navbar-header">
8         <button type="button" class="navbar-toggle"
9           data-toggle="collapse" data-target=".navbar-collapse">
10           <span class="sr-only">Toggle navigation</span>
11           <span class="icon-bar"></span>
12           <span class="icon-bar"></span>
13           <span class="icon-bar"></span>
14         </button>
15         <a class="navbar-brand" href="#">Mos Eisley's Cantina</a>
16       </div>
17       <div class="navbar-collapse collapse">
18         <ul class="nav navbar-nav">
19           <li class="active"><a href="{% url 'home' %}">Home</a></li>
20           <li><a href="/pages/about">About</a></li>
21           <li><a href="{% url 'contact' %}">Contact</a></li>
22           {% if user %}
23             <li><a href="{% url 'sign_out' %}">Logout</a></li>
24             {% else %}
25             <li><a href="{% url 'sign_in' %}">Login</a></li>
26             <li><a href="{% url 'register' %}">Register</a></li>
27           {% endif %}
28         </ul>
29       </div>
30     </div>
31   </div>
32 </div>
```

```

27     </div> <!-- end navbar links -->
28     </div> <!-- end container -->
29 </div> <!-- end navbar -->
30
31 </div><!-- end navbar-wrapper -->

```

Add this to your *base.html* template just below the opening body tag.

Comparing the above to our old navbar, you can see that we still have the same set of list items. However, with the new navbar we have a navbar header and some extra divs wrapping it. Most important are the `<div class="container">` (aka container divs) because they are vital for the responsive design in Bootstrap 3.

Since Bootstrap 3 is based on the “mobile-first” philosophy, the site is responsive from the start. For example, if you re-size your browser to make the window smaller, you will see the navbar links will disappear and a drop down button will be shown instead of all the links (although we do have to insert the correct Javascript for this to happen, which we haven’t done yet). Clicking that drop-down button will show all the links. An image of this is shown below (ignore the colors for now).

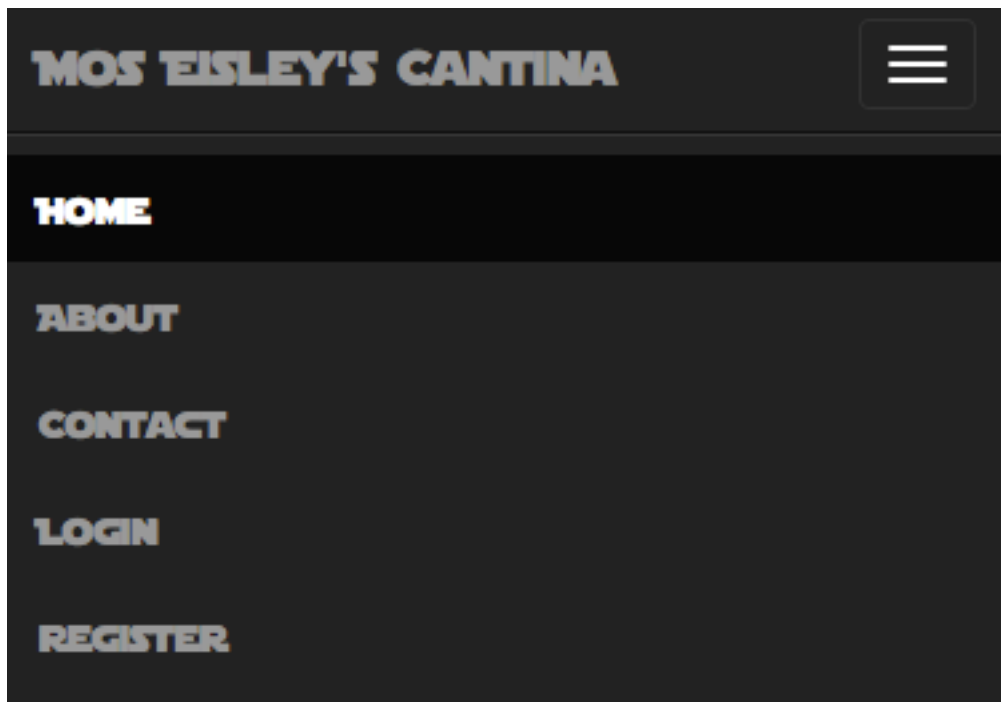


Figure 9.3: responsive dropdown

If you want to test out the mobile-first-ness of Bootstrap on your mobile phone, you can tell

Django to bind to your network adapter when you start it so that the development server will accept connections from other machines. To do this, just type the following:

```
1 $ ./manage.py runserver 0.0.0.0:8000
```

Then from your mobile phone, put in the IP address of your computer (and port 8000), and you can see the site on your mobile phone or tablet. On unix machine or Macs you can do an `ifconfig` from terminal to get your IP address. On windows machines it's `ipconfig` from the command prompt. We tested it on 192.168.1.12:8000.

There's always certain sense of satisfaction that comes out of seeing your websites on your own phone. Right?

There are also simpler ways to test out the responsiveness of bootstrap. The simplest being simply resizing your browser and what bootstrap automatically adjust. Also you can use an online tool like [viewpoint-resizer](#).

Colors

To make the navbar use our cool starjedi font, update *mec.css* by replacing the current `h1` style with:

```
1 h1, .nav li, .navbar-brand {
2   font-family: 'starjedi', sans-serif;
3 }
```

This just applies our font selection to the links in the navbar as well as the `h1` tag. But as is often the case with web development, doing this will make the navbar look a bit off when viewed from the iPhone in portrait mode. To fix that, we have to adjust the sizing of the `.navbar-brand` to accommodate the larger size of the starjedi font. This can be done by adding the following to *mec.css*:

```
1 .navbar-brand {
2   height: 40px;
3 }
```

Now your navbar should look great on pretty much any device. With the navbar more or less taken care of, let's add a footer as well, just below - `<h1>Hello, world!</h1>`:

```
1 <hr class="featurette-divider">
2
3 <footer>
4   <p class="pull-right"><a href="#">Back to top</a></p>
```

```
5 <p class="pull-left">&copy; 2014 <a  
    href="http://realpython.com">The Jedi Council</a></p> <p  
    class="text-center"> &middot; Powered by about 37 AT-AT  
    Walkers, Python 3 and Django 1.8 &middot; </p>  
6 </footer>
```

The actual HTML is more or less the same, right? But you may be saying to yourself, “What is the `<footer>` tag? Why shouldn’t we just use a `<div>`?”

Well I’m glad you asked, because that brings us to our next topic.

HTML5 Sections and the Semantic Web

Have you heard the term “Semantic Web” before? It’s an idea that has been around for a long time, but hasn’t really become commonplace yet. One of the goals behind HTML5 is to change that. Before we get into the HTML5 part though, let’s define “Semantic Web” straight from the Wikipedia [page](#):

The **Semantic Web** is a collaborative movement led by international standards body the World Wide Web Consortium (W3C). The standard promotes common data formats on the World Wide Web. By encouraging the inclusion of semantic content in web pages, the Semantic Web aims at converting the current web, dominated by unstructured and semi-structured documents into a “web of data”. The Semantic Web stack builds on the W3C’s Resource Description Framework (RDF).

According to the W3C, “The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries”. The term was coined by Tim Berners-Lee for a web of data that can be processed by machines.

So in a nutshell, the “Semantic Web” is an idea, started by the creator of the web, that aims to make DATA on the web more accessible, specifically more accessible programmatically. There are a number of techniques used to achieve this such as [Microformats](#), [Resource Description Frameworks](#), [Web Ontology Languages](#), and [others](#), that when combined together can make the web much more accessible programmatically. It’s a huge topic, but we’ll just focus on one small aspect: *the new section tags in HTML5*.

To preface the usefulness of section tags in HTML5, let’s revisit our navigation bar:

```
1 <!-- NAVBAR ===== -->
2 <div class="navbar-wrapper">
3
4     <div class="navbar navbar-inverse navbar-static-top"
5         role="navigation">
6         <div class="container">
7             <div class="navbar-header">
8                 <button type="button" class="navbar-toggle"
9                     data-toggle="collapse" data-target=".navbar-collapse">
10                     <span class="sr-only">Toggle navigation</span>
11                     <span class="icon-bar"></span>
```



```

10         <span class="icon-bar"></span>
11         <span class="icon-bar"></span>
12     </button>
13     <a class="navbar-brand" href="#">Mos Eisley's Cantina</a>
14 </div>
15 <div class="navbar-collapse collapse">
16     <ul class="nav navbar-nav">
17 <li class="active"><a href="{% url 'home' %}">Home</a></li>
18 <li><a href="/pages/about">About</a></li>
19 <li><a href="{% url 'contact' %}">Contact</a></li>
20 {% if user %}
21     <li><a href="{% url 'sign_out' %}">Logout</a></li>
22     {% else %}
23     <li><a href="{% url 'sign_in' %}">Login</a></li>
24     <li><a href="{% url 'register' %}">Register</a></li>
25 {% endif %}
26     </ul>
27 </div> <!-- end navbar links -->
28 </div> <!-- end container -->
29 </div> <!-- end navbar -->
30
31 </div><!-- end navbar-wrapper -->

```

Now imagine for a second that you are a computer program trying to determine the *meaning* of that section of HTML. It's not very hard to tell what is going on there, right? You have a few clues like the classes that are used, but they are not standard across the web, and the best you could do is guess.

NOTE: The astute reader may point out the `role` attribute of the third `div` from the top, which is actually part of the HTML5 semantic web specification. It's used to provide a means for assistive technology to better understand the interface. So it's really just a high-level overview. What we're after is more of a granular look at the HTML itself.

The problem is that HTML is really a language to describe structure; it tells us how this data should look, not what this data is. HTML5 section tags are the first baby steps used to start to make data more accessible. An HTML5 section tag can be used in place of a `div` to provide some meaning as to what type of data is contained in the `div`.

Inheritance

Before moving on, let's update the parent template, *base.html*, to add template inheritance.

Remove:

```
1 <h1>Hello, world!</h1>
```

Then update the body like so:

```
1 <div class="container">
2
3     {% block content %}
4     {% endblock %}
5
6
7     <hr class="featurette-divider">
8
9     <footer>
10         <p class="pull-right"><a href="#">Back to top</a></p>
11         <p class="pull-left">&copy; 2014 <a
            href="http://realpython.com">The Jedi Council</a></p> <p
            class="text-center"> &middot; Powered by about 37 AT-AT
            Walkers, Python 3 and Django 1.8 &middot; </p>
12     </footer>
13
14 </div>
```

HTML5 Section Tags

HTML5 defines the following section (or section-like) [tags](#):

1. **section** – Used for grouping together similar content. It's like a `div` with semantic meaning. So all the data in the `section` tag should be related. An author's bio could be sectioned off from the blog post in a `section` tag, for example.
2. **article** – Same as a `section` tag, but specifically for content that should be syndicated; a blog post is probably the best example.
3. **aside** – Used for somewhat related content that isn't necessarily critical to the rest of the content. There are several places in this course where asides are used. For example, there is an aside (NOTE) just a few paragraphs up that starts with "The astute reader".

4. **header** – Not to be confused with the head tag (which is the head of the document), the header is the header of a section. It is common that headers have h1 tags. But like with all the tags in this section, header describes the data, so it's not the same as h1 which describes the appearance. Also note that there can be several headers in an HTML document, but only one per section tag.
5. **nav** – Used mainly for site navigation, like our navbar above.
6. **footer** – Despite the name, footer doesn't need to be at the bottom of the page. Footer elements contain information about the containing section they are in. Often-times (like we did above) they contain copyright info, information about the author / company. They can of course also contain footnotes. Also like headers, there can be multiple footer tags per HTML document, but one per section tag.

Section Tags in action

So with our new understanding of some of the HTML section elements, let's rewrite our navbar taking advantage of them:

```
1 <nav class="navbar navbar-inverse navbar-static-top"
  role="navigation">
2   <div class="container">
3     <header class="navbar-header">
4       <button type="button" class="navbar-toggle"
5         data-toggle="collapse" data-target=".navbar-collapse">
6         <span class="sr-only">Toggle navigation</span>
7         <span class="icon-bar"></span>
8         <span class="icon-bar"></span>
9         <span class="icon-bar"></span>
10      </button>
11      <a class="navbar-brand" href="{% url 'home' %}">Mos Eisley's
        Cantina</a>
12    </header>
13    <div class="navbar-collapse collapse">
14      <ul class="nav navbar-nav">
15        <li class="active"><a href="{% url 'home' %}">Home</a></li>
16        <li><a href="/pages/about">About</a></li>
17        <li><a href="{% url 'contact' %}">Contact</a></li>
18        {% if user %}
19          <li><a href="{% url 'sign_out' %}">Logout</a></li>
20        {% else %}
```

```

20         <li><a href="{% url 'sign_in' %}">Login</a></li>
21         <li><a href="{% url 'register' %}">Register</a></li>
22     {% endif %}
23 </ul>
24 </div>
25 </div>
26 </nav>

```

Notice the difference? Basically we replaced two of the divs with more meaningful section elements.

1. The second line and the second to last line define the nav or navigation section, which lets a program know: **This is the main navigation section for the site.**
2. Inside the nav element there is a header element, which encompasses the “brand” part of the navbar, which is intended to show the name of the site with a link back to the main page.

Those two changes may not look like much, but depending on the scenario, they can have a huge impact. Let’s look at a few examples real quick.

1. *Web scraping*: Pretend you are trying to scrape the website. Before, you would have to follow every link on the page, make sure the destination is still in the same domain, detect circular links, and finally jump through a whole bunch of loops and special cases just to visit all the pages on the site. Now you can basically achieve the same thing in just a few steps. Take a look at the following pseudo-code:

```

1 access site
2
3 for <a> in <nav>:
4     click <a>
5     check for interesting sections
6     extract data
7
8 #n ow return to main landing page
9 <nav>.<header>.<a>.click()

```

2. *Specialized interfaces*: Imagine trying to read a web page on something tiny like google glass, or a “smart watch”. Lot’s of pinching and zooming and pretty tough to do. Now imagine if that smart watch could determine the content and respond to voice commands like “Show navigation”, “Show Main Content”, “Next Section”. This could happen across all sites today if they used Semantic markup. But there is no way to make that work if your site is just a bunch of unstructured divs.

3. *Accessibility Devices*: Several accessibility devices rely on semantic markup to make it possible for those with accessibility needs to access the web. There are a number of write ups about this on the web including [here](#) and [here](#) and [here](#)

But it's not just about making your site more accessible in a programatic way. It's also about making the code for your site clearer and thus easier to maintain.

As we have talked about before, writing maintainable code is a huge part of Software Craftsmanship. If you go back and look at our previous example with all the divs vs the semantic markup you should find it much easier to see whats going on at a glance by just looking at the section tags. This may not seem like much, but these tiny change add up quickly. Every little bit counts.

Try this: Use your browser to View Page Source of your favorite web site. Chances are if it's not using semantic markup you'll be drowning in divs, making it tough to dig through the HTML and see what corresponds to what. Adding semantic markup make this process much simpler and thus makes things more maintainable.

That's basically it. In our case, we now have a navigation structure that not only looks cool (hopefully), but is accessible programmatically - which is a really good thing. HTML5 section tags are probably the simplest thing you can do to make your page more semantic. To get started, simply replace div tags where appropriate with HTML5 sections tag that correspond to the type of data you're showing.

We will cover some of the other new tags in HTML5 that apply to semantics as we go further into the chapter. Covering the entirety of the Semantic Web is way beyond the scope of this course, but if you want to find out more information, start at the Wikipedia [page](#) and watch the TED [talk](#) by the creator of the web about it.

More Bootstrap Customizations

A Carousel of Star Wars' Awesomeness

Coming back to our new site design, the next thing we'll do is add in some cool Star Wars pictures that auto-rotate (like a slideshow) across the main page. This is done with the Bootstrap [carousel](#) control. Since we have everything we need at this point in the *base.html* file, let's put the carousel in our *index.html* file.

Here's some example code:

```
1 {% extends 'base.html' %}
2
3 {% block content %}
4
5     {% load static %}
6
7     <center>
8     <section id="myCarousel" class="carousel slide"
9         data-ride="carousel" style="max-width: 960px;">
10         <!-- Indicators -->
11         <ol class="carousel-indicators">
12             <li data-target="#myCarousel" data-slide-to="0"
13                 class="active"></li>
14             <li data-target="#myCarousel" data-slide-to="1"></li>
15             <li data-target="#myCarousel" data-slide-to="2"></li>
16         </ol>
17         <div class="carousel-inner">
18             <figure class="item active">
19                 
21                 <figcaption class="carousel-caption">
22                     <h1>Join the Dark Side</h1>
23                     <p>Or the light side. Doesn't matter. If your into Star
24                         Wars then this is the place for you.</p>
25                     <p><a class="btn btn-lg btn-primary" href="{% url
26                         'register' %}" role="button">Sign up today</a></p>
27                 </figcaption>
28             </figure>
29             <figure class="item">
```

```

25     
26     <figcaption class="carousel-caption">
27         <h1>Wage War</h1>
28         <p>Daily chats, Weekly Gaming sessions, Monthly
            real-life battle re-inactments.</p>
29         <p><a class="btn btn-lg btn-primary" href="#"
            role="button">I'm ready to fight</a></p>
30     </figcaption>
31 </figure>
32 <figure class="item">
33     
34     <figcaption class="carousel-caption">
35         <h1>Meet fellow Star Wars Fans</h1>
36         <p>Join forces with fellow padwans and Jedi who lives
            near you.</p>
37         <p><a class="btn btn-lg btn-primary" href="#"
            role="button">Check the Members Map</a></p>
38     </figcaption>
39 </figure>
40 </div>
41 <a class="left carousel-control" href="#myCarousel"
    data-slide="prev"><span class="glyphicon
    glyphicon-chevron-left"></span></a>
42 <a class="right carousel-control" href="#myCarousel"
    data-slide="next"><span class="glyphicon
    glyphicon-chevron-right"></span></a>
43 </section>
44 </center>
45
46 {% endblock %}

```

That's a fair amount of code there, so let's break it down into sections.

```

1 <section id="myCarousel" class="carousel slide"
    data-ride="carousel">
2     <!-- Indicators -->
3     <ol class="carousel-indicators">
4         <li data-target="#myCarousel" data-slide-to="0"
            class="active"></li>

```

```

5     <li data-target="#myCarousel" data-slide-to="1"></li>
6     <li data-target="#myCarousel" data-slide-to="2"></li>
7 </ol>

```

1. The first line is the HTML5 section tag, which is just separating the carousel as a separate section of the document. The attribute `data-ride="carousel"` starts the carousel animation on page load.
2. The ordered list `ol` displays the three dots near the bottom of the carousel that indicate which page of the carousel is being displayed. Clicking on the list will advance to the associated image in the carousel.

The next section has three items that each correspond to an item in the carousel. They all behave the same, so let's just describe the first one, and the same will apply to all three.

```

1 <figure class="item active">
2   
3   <figcaption class="carousel-caption">
4     <h1>Join the Dark Side</h1>
5     <p>Or the light side. Doesn't matter. If you're into Star
        Wars then this is the place for you.</p>
6     <p><a class="btn btn-lg btn-primary" href="{% url 'register'
            %}" role="button">Sign up today</a></p>
7   </figcaption>
8 </figure>

```

1. `figure` is another HTML5 element that we haven't talked about yet. Like the section elements, it is intended to provide some semantic meaning to the page:

NOTE: The `figure element` represents a unit of content, optionally with a caption, which is self-contained, that is typically referenced as a single unit from the main flow of the document, and that can be moved away from the main flow of the document without affecting the document's meaning.

2. Just like with the HTML5 section tags, we are replacing the overused `div` element with an element `figure` that provides some meaning.
3. Also as a sub-element of the figure we have the `<figcaption>` element which represents a caption for the figure. In our case we put the "join now" message as a link to the registration page in our caption.

The final part of the carousel are two links on the left and right of the carousel that look like > and <. These advance to the next or previous slide in the carousel, and the code for these links look like:

```
1 <a class="left carousel-control" href="#myCarousel"
  data-slide="prev"><span class="glyphicon
    glyphicon-chevron-left"></span></a>
2 <a class="right carousel-control" href="#myCarousel"
  data-slide="next"><span class="glyphicon
    glyphicon-chevron-right"></span></a>
```

Finally, create an “img” folder within the “static” folder, and then grab the *star-wars-battle.jpg*, *star-wars-cast.jpg*, and *darth.jpg* images from the “chpo8” folder in the exercise [repo](#) and them to that newly created “img” folder.

And that’s it. You now have a cool carousel telling visiting younglings about the benefits of joining the site.

Some additional content

We can add some additional content below the carousel to talk about some of the benefits and features of the membership site. Starting with the most straightforward way, we could just add a bunch of extra HTML after the carousel like so:

```
1 <br><br>
2
3 <section class="container marketing">
4   <!-- Three columns of text below the carousel -->
5   <div class="row center-text">
6     <div class="col-lg-4">
7       
8       <h2>Hone your Jedi Skills</h2>
9       <p>All members have access to our unique training and
          achievements ladders. Progress through the levels and
          show everyone who the top Jedi Master is! </p>
10      <p><a class="btn btn-default" href="#" role="button">Sign Up
          Now &raquo;</a></p>
11    </div><!-- /.col-lg-4 -->
12    <div class="col-lg-4">
13      
```

```

14     <h2>Build your Clan</h2>
15     <p>Engage in meaningful conversation, or bloodthirsty battle!
        If it's related to Star Wars, in any way, you better
        believe we do it here.</p>
16     <p><a class="btn btn-default" href="#" role="button">Sign Up
        Now &raquo;</a></p>
17 </div><!-- /.col-lg-4 -->
18 <div class="col-lg-4">
19     
20     <h2>Find Love</h2>
21     <p>Everybody knows Star Wars fans are the best mates for Star
        Wars fans. Find your Princess Leia or Han Solo and
        explore the stars together.</p>
22     <p><a class="btn btn-default" href="#" role="button">Sign Up
        Now &raquo;</a></p>
23 </div><!-- /.col-lg-4 -->
24 </div> <!-- /.row -->
25 </section>

```

Be sure to grab the images - *yoda.jpg*, *clone_army.jpg*, *leia.jpg* - from the repo again.

This gives us three sections each with an image in a circular border (because circles are hot these days... right?), some text and a view details button. This is all pretty straight-forward Bootstrap stuff; Bootstrap uses a [grid](#) system, which breaks up the page into columns and rows.

By putting content into the same row (that is to say, all elements that are a child of the same `<div class="row center-text">`), it will appear lined up side by side. And as you might expect, a second row will appear underneath the previous row. Likewise with columns, add content and a column will appear to the right of the previous column (i.e. `<div class="column">`) or to the left of the subsequent column.

For our marketing info, we create one row div with three child divs `<div class="col-lg-4">`. `col-lg-4` is interesting. With Bootstrap each row has a total of 12 columns. You can break up the page how you want as long as the sum of all the columns is 12. In other words, `<div class='col-6'>` will take up half of the width of the page, whereas `<div class='col-4'>` will take up a third of the width of the page. For a more in-depth discussion on this, take a look at [this](#) blog post.

Did you notice the other identifier? `lg`. This stands for “large”. Along with `lg`, there’s `xs`, `sm`, and `md` for extra-small, small, and medium, respectively.

Bootstrap 3, being responsive by default, has the notion of extra-small, small, medium, and large screen sizes. So by using the `lg` identifier, you can say, “I want columns only if the screen size is large,”. Likewise, you can do the same with other sizes.

You can see this by looking at our Marketing Items using a large screen size (where it will have three columns because we are using the `col-lg-4` class:

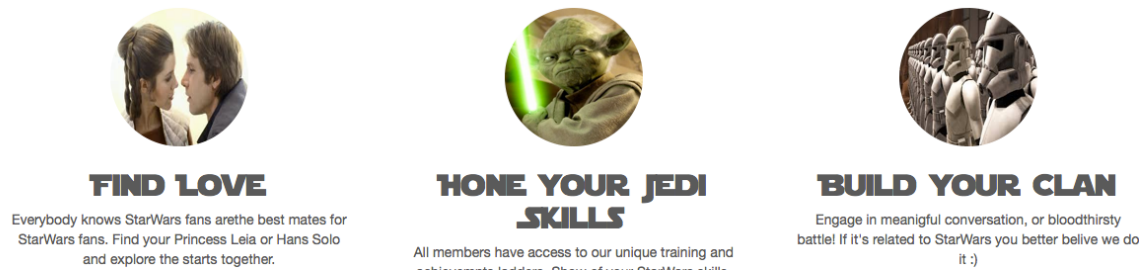


Figure 9.4: large screen size

And if we view the page on a medium or small screen size it will show only one column:

The grid system is really powerful and completely responsive. With the above code, try resizing your browser to fill your entire monitor and you will see three columns. Make the width of your browser smaller and you will see the columns stack on top of each other. Pretty cool, huh?

The best place to get all the information about the Bootstrap grid system is directly from the Bootstrap [docs](#). They are really good and worth a read.

So we have some basic marketing material there, but we had to type a lot of boilerplate HTML code. Plus we are not really taking advantage of Django's templating system. Let's do that.



FIND LOVE

Everybody knows StarWars fans are the best mates for StarWars fans. Find your Princess Leia or Hans Solo and explore the starts together.

[Sign Up Now »](#)



HONE YOUR JEDI SKILLS

All members have access to our unique training and achievements ladders. Show off your StarWars skills, progress through the levels and show everyone who the

Custom template tags

Built-in tags

Django provides you with a lot of built-in template [tags](#) like the `{% static %}` tag that can make dealing with dynamic data easier. Several of them you are already familiar with:

- `{% static %}` provides a way to reference the location of your static folder without hardcoding it in.
- `{% extend 'base.html' %}` allows for the extending of a parent template.
- `{% csrf %}` provides protection against Cross Site Request Forgeries.

Django also provides a number of built in filters which make working with data in your templates easier. Some examples include:

- `{{ value | capfirst }}` - if value = “jack” this will produce “Jack”
- `{{ value | add:"10" }}` - if value is “2” this will produce “12”
- `{{ value | default:"Not logged in" }}` - if value evaluates to False, use the given default - i.e., “Not logged in”

There are tons more built-in filters and template tags. Again, check out the Django [documentation](#) for more information.

Custom template tags

Can’t find what your looking for in the docs? You can also create your own custom template [tags](#). This is great if you have certain content/html structures that you’re repeating over and over.

For example, this section of code-

```
1 <div class="col-lg-4">
2   
3   <h2>Hone your Jedi Skills</h2>
4   <p>All members have access to our unique training and
      achievements ladders. Progress through the levels and show
      everyone who the top Jedi Master is! </p>
5   <p><a class="btn btn-default" href="#" role="button">View details
      &raquo;</a></p>
6 </div><!-- /.col-lg-4 -->
```

-could be easily re-factored into a template tag so you wouldn't have to type so much. Let's do it.

Directory structure

First, create a directory in your *main* app called “templatetags” and add the following files to it:

```
1 .
2 __init__.py
3 models.py
4 templatetags
5     __init__.py
6     marketing.py
7 views.py
```

Define the tag

In *marketing.py* add the following code to define the custom template tag:

```
1 from django import template
2 register = template.Library()
3
4
5 @register.inclusion_tag('circle_item.html')
6 def circle_header_item(img_name="yoda.jpg", heading="yoda",
7                        caption="yoda",
8                        button_link="register", button_title="View
9                        details"):
10
11     return {
12         'img': img_name,
13         'heading': heading,
14         'caption': caption,
15         'button_link': button_link,
16         'button_title': button_title
17     }
```

The first two lines just register `circle_header_item()` as a template tag so you can use it in a template using the following syntax:

```
1 {% circle_header_item img_name='img.jpg' heading='nice image' %}
```

Just as if you were calling a regular Python function, when calling the template tag you can use [keywords or positional arguments](#) - but not both. Arguments not specified will take on the default value.

The `@register` function simply creates a context for the template to use. It does this by creating a dictionary of variable names mapped to values. Each of the variable names in the dictionary will become available as template variables.

The remainder of the line-

```
1 @register.inclusion_tag('circle_item.html')
```

-declares an HTML fragment that is rendered by the template tag. Django will look for the HTML file everywhere that is specified in the `TEMPLATE_LOADERS` list, which is specified in the `settings.py` file. In our case, this is under the `template` subdirectory.

HTML Fragment

The `circle_item.html` file looks like this:

```
1 {% load staticfiles %}
2
3 <div class="col-lg-4">
4     
6     <h2>{{ heading }}</h2>
7     <p>{{ caption }}</p>
8     <p><a class="btn btn-default" href="{% url button_link %}"
9         role="button">{{button_title}}</a></p>
10 </div>
```

Add this to the “templates” directory.

We took our original HTML to define the structure and then used several template variables (the ones we returned from `main.templatetags.marketing.circle_header_item`), which enables us to dynamically populate the structure with data. This way we can repeat the structure several times without having to repeat the HTML. Everything is pretty standard here, but there is one template tag/filter you may not be familiar with:

```
1 src="{% static 'img/'|add:img %}"
```

Looking at the `src` attribute you can see it is using the standard `{% static %}` tag. However there is a funny looking bit - `'img/'|add:img` - which is used for concatenation.

You can use it for addition:

```
1 {{ num|add:'1' }}
```

If you passed 5 in for num, then this would render 6 in your HTML.

add can also concatenate two lists as well as strings (like in our case). It's a pretty handy filter to have.

Adding the tag

In terms of our inclusion tag, we call it directly from *index.html*:

```
1 <section class="container marketing">
2   <!-- Three columns of text below the carousel -->
3   <div class="row center-text">
4     {% circle_header_item img_name='yoda.jpg' heading='Hone your
5       Jedi Skills'
6       caption='All members have access to our unique training and
7         achievements ladders.
8         Progress through the levels and show everyone who the top
9         Jedi Master is!' %}
10
11     {% circle_header_item img_name='clone_army.jp' heading='Build
12       your Clan'
13       caption='Engage in meaningful conversation, or bloodthirsty
14         battle! If it's
15         related to Star Wars, in any way, you better believe we do it.
16         :)' %}
17
18     {% circle_header_item img_name="leia.jpg" heading="Find Love"
19       caption="Everybody knows Star Wars fans are the best mates
20         for Star Wars
21         fans. Find your Princess Leia or Han Solo and explore the stars
22         together." button_title="Sign Up Now" %}
23   </div>
24 </section>
```

NOTE: Django template tags are not allowed to span multiple lines, so you actually have to string everything together in one super long line. We broke it apart for readability in this course. Be sure to turn word-wrap on in your text editor so you can view all of the text without having to scroll.

There you have it!

Just three calls to our new template tag and nowhere near as much HTML all over the place. This is a bit of a cheat though since we hard coded the text directly in the HTML. You'd actually want to pass it in through the view to make it truly dynamic. We'll save this for an exercise at the end of the chapter.

A Note on Structure

Pairing Bootstrap with Django templates can not only make design a breeze, but also greatly reduce the amount of code you have to write.

If you do start to use a number of custom template tags and/or use multiple templates to render a page, it can make it more difficult to debug issues within the template. The hardest part of the process is often determining which file is responsible for the markup you see on the screen. This becomes more prevalent the larger an application gets. *In order to combat this issue you should choose a meaningful structure to organize your templates and stick to it religiously.* It will make debugging and maintenance much easier.

Currently, in our code, we are just throwing everything in the “templates” directory. This approach doesn’t scale; it makes it very difficult to know which templates belong to which views and how the templates relate to one other. Let’s fix that.

Restructure

First, let’s organize the templates by subfolder, where the name of each subfolder corresponds with the app that is using the template. Reorganizing our template folder as such produces:

```
1 .
2 base.html
3 contact
4     contact.html
5 flatpages
6     default.html
7 main
8     index.html
9     templatetags
10         circle_item.html
11     user.html
12 payments
13     cardform.html
14     edit.html
15     errors.html
16     field.html
17     register.html
18     sign_in.html
```

With this setup we can easily see which templates belong to which application. This should make it easier to find the appropriate templates. We do have to change the way in which we refer to templates in both our views and our templates by appending the app name before the template when calling it so that Django can find the template correctly. This is just a minor change and also adds readability.

For example, in a template you change this:

```
1 {% include "cardform.html" %}
```

To:

```
1 {% include "payments/_cardform.html" %}
```

The only thing we are doing differently here is including the folder where the template is stored. And since the folder is the same name as the app that uses the template, we also know where to find the corresponding *views.py* file, which in this case would be *payments/views.py*.

In a *views.py* file it's the same sort of update:

```
1 return render_to_response(  
2     'payments/sign_in.html',  
3     {  
4         'form': form,  
5         'user': user  
6     },  
7     context_instance=RequestContext(request)  
8 )
```

Again, just like in the template, we add the name of the directory so that Django can locate the template correctly.

Don't make these changes just yet.

This is a good first step to organizing our templates, but we can take it a bit further and communicate the purpose of the template by adhering to a naming convention.

Naming Convention

Basically you have three types of templates:

1. Templates that were meant to be extended, like our *base.html* template.
2. Templates that are meant to be included, like our *payments/cardform.html* template.

3. Template tags and filters, including inclusion tags.

If we further extend our organization structure to differentiate between those three types of templates, then we can quickly understand the purpose of the template with a simple `ls` command (in Unix), without even having to open up the template. You can use any naming convention you like as long as it works for you.

Case	Description	Example
1	start name with <code>__</code>	<code>__base.html</code>
2	start name with <code>_</code>	<code>payments/_cardform.html</code>
3	put in <code>templatetag</code> sub directory	<code>@register.inclusion_tag('main/templatetags/circle_item.html')</code>

Doing this will let us quickly identify what the purpose of each of our templates is. This will make things easier, especially when your application grows and you start including a number of different types of templates and referencing them from all over your application.

Go ahead and make the restructuring and renaming changes now. Once done, run the sever to make sure you caught everything.

Your templates directory should now look like this:

```
1 .
2 __base.html
3 contact
4     contact.html
5 flatpages
6     default.html
7 main
8     index.html
9     templatetags
10         circle_item.html
11     user.html
12 payments
13     _cardform.html
14     _field.html
15     edit.html
16     errors.html
17     register.html
18     sign_in.html
```

As you can see, the basic structure stays the same, but now if you look at the payments folder, for example, you can quickly tell that `*_cardform.html*` and `*_field.html*` are templates that are meant to be included in other templates, while the three other templates represent a page in the application. And we know all of that without even looking at their code.

Update template tag

The final thing we should change is the name of our template tag, which is currently `marketing`. Remember when we load the tags to be used in a template, our load looks like this:

```
{% load marketing %}
```

The first question you should have after reading that is, “Well where are the marketing template tags, and what’s in there?”

Let’s first rename the `marketing.py` to `main_marketing.py` so at a glance we can tell where the template tags are located (within the “`templates/main/templatetags`” folder). Also in the `main_marketing.py` file we have a tag called `circle_header_item`. While this may describe what it is, it doesn’t tell us where it came from. Larger Django projects could have templates that include ten other templatetag libraries. In such a case, it’s pretty difficult to tell which tags belong to which library. *The solution is to name the tag after the library.*

One convention is to use the `taglibname__tagname`. In this case `circle_header_item` becomes `marketing__circle_item`. This way, if we find it in a template, we know it comes from a library called `marketing`, and if we just jump to the top of the HTML file, we’ll see the `{% load main_marketing %}` tag and thus we’ll know to look for the code: `main.templatetags.main_marketing.marketing__circle_item`.

This may not seem like much but it’s a life saver when the application becomes large and/or you are revisiting the app six months after you wrote it. So take the time and add a little structure to your templates. You’ll thank yourself later.

Make all the changes and run your tests. You’ll probably pick up anything you missed just by running the test suite since moving around templates will cause your tests to fail. Still having problems? Check the Project in the “chpo9” folder in the exercise [repo](#).

Conclusion

We have talked about several important topics in this chapter with regards to Bootstrap:

1. First, we went through the basics of Bootstrap 3, its grid system and some of the cool tools it has for us to use.
2. Then, we talked about using custom fonts, putting in our own styling and imagery, and making Bootstrap and Django templates play nicely together.

This all represents the core of what you need to know to successfully work with Bootstrap and Django. That being said, Bootstrap is a giant library, and we barely scratched the surface. Read more about Bootstrap on the official [website](#). There are a number of things that you can do with Bootstrap, and the more you use it, the better your Bootstrap skills will become. With that in mind, the exercises will go through a number of examples designed to give you more practice with Bootstrap.

Finally, if you need additional help with Bootstrap, check out [this](#) blog post, which touches on a number of features and components of the framework, outside the context of Django - which may be easier to understand.

We also talked about two important concepts that go hand and hand with Bootstrap and front end work:

1. HTML5 Semantic Tags - these help make your web site more accessible programmatically, while also making your HTML easier to understand and maintain by fellow web developers. The web is about data and making the data of your website more accessible is generally a good thing.
2. Template Tags: We talked a lot about custom template tags and how they can reduce your typing, and make your templates much easier to use. We will explore custom tags further in the exercises with a look at how to “data drive” your website using custom template tags.

A lot of ground was covered so be sure to go through the exercises to help ensure you understand everything that we covered in this chapter.

Exercises

1. Bootstrap is a front-end framework, and although we didn't touch much on it in this chapter, it uses a number of CSS classes to insert things on the page, making it look nice and provide the responsive nature of the page. It does this by providing a large number of classes that can be attached to any HTML element to help with placement. All of these capabilities are described on the [Bootstraps CSS page](#). Have a look through it, and then let's put some of those classes to use.
 - In the main carousel, the text, "Join the Dark Side" on the Darth Vader image, blocks the image of Darth himself. Using the Bootstrap / carousel CSS, can you move the text and sign up button to the left of the image so as to not cover Lord Vader?
 - If we do the above change, everything looks fine until we view things on a phone (or make our browser really small). Once we do that, the text covers up Darth Vader completely. Can you make it so on small screens the text is in the "normal position" (centered / lower portion of the image) and for larger images it's on the left.
2. In this chapter, we updated the Home Page but we haven't done anything about the Contact Page, the Login Page, or the Register Page. Bootstrapify them. Try to make them look awesome. The Bootstrap examples [page](#) is a good place to go to get some simple ideas to implement. Remember: try to make the pages semantic, reuse the Django templates that you already wrote where possible, and most of all have fun.
3. Previously in the chapter we introduced the `marketing__circle_item` template tag. The one issue we had with it was that it required a whole lot of data to be passed into it. Let's see if we can fix that. Inclusion tags don't have to have data passed in. Instead, they can inherit context from the parent template. This is done by passing in `takes_context=True` to the inclusion tag decorator like so:

```
1 @register.inclusion_tag('main/templatetags/circle_item.html',  
    takes_context=True)
```

If we did this for our `marketing__circle_item` tag, we wouldn't have to pass in all that data; we could just read it from the context.

Go ahead and make that change, then you will need to update the `main.views.index` function to add the appropriate data to the context when you call `render_to_response`. Once that is all done, you can stop hard-coding all the data in the HTML template and instead pass it to the template from the view function.

For bonus points, create a `marketing_info` model. Read all the necessary data from the model in the index view function and pass it into the template.

Chapter 10

Building the Members Page

Now that we have finished getting the public facing aspects of our site looking “nice and purdy”, it’s time to turn our attention to our paying customers and give them something fun to use, that will keep them coming back for more (at least in theory).

User Story

As always, let's start with the user story we defined in Chapter 7...

US3: Members Home Page

The main page where returning “padwans” are greeted. This members page is a place for announcements and to list current happenings. It should be the single place a user needs to go to know all of the latest news at MEC.

And that is exactly what we are going to do. To give you an idea of where we should end up with by the end of this chapter, have a look at the screenshot below:

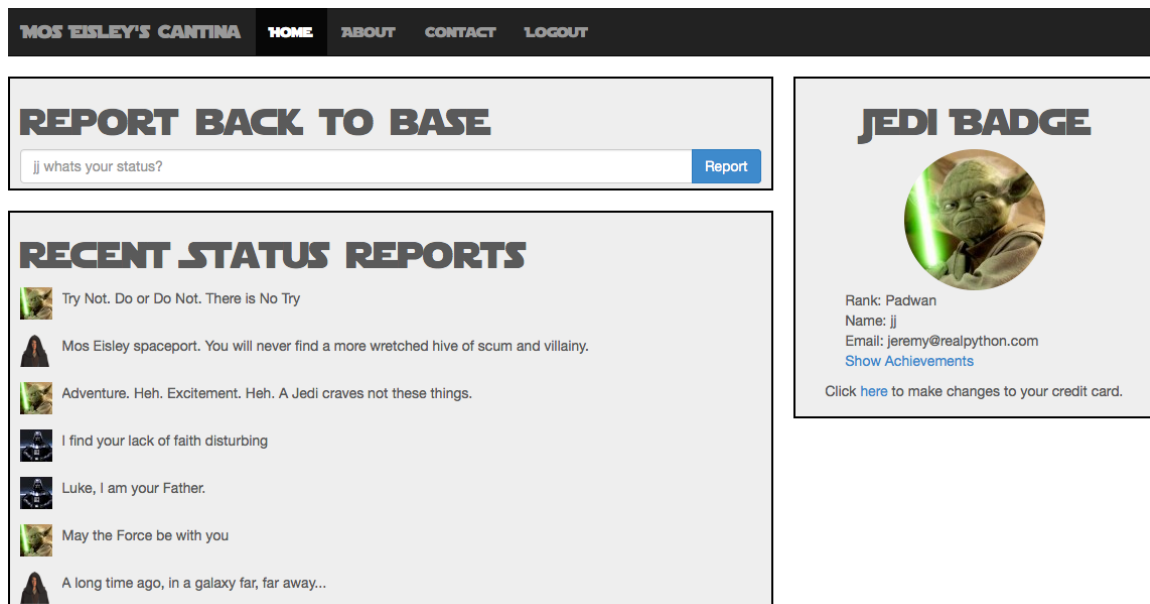


Figure 10.1: Members Page

Update Main Template

This is what a registered user will see after login. If you recall, the registered user page is in *templates/main/user.html*. If we add three boxes to the *user.html* page - ‘Report Back to Base’, ‘Jedi Badge’, and ‘Recent Status Reports’ boxes - then the template should look like this:

```
1 {% extends "__base.html" %}
2 {% load staticfiles %}
3 {% block content %}
4     <div class="row member-page">
5         <div class="col-sm-8">
6             <div class="row">
7                 {% include "main/_statusupdate.html" %}
8                 {% include "main/_lateststatus.html" %}
9             </div>
10        </div>
11        <div class="col-sm-4">
12            <div class="row">
13                {% include "main/_jedibadge.html" %}
14            </div>
15        </div>
16    </div>
17 {% endblock %}
```

Looks pretty simple, right? Here we added the basic Bootstrap scaffolding. Remember that Bootstrap uses a grid system, and we can access that grid system by creating CSS classes that use the row and col syntax. In this case, we have used special column classes `col-sm-8` and `col-sm-4`. Since the total columns available is 12, this tells Bootstrap that our first column should be 2/3 of the screen (8 of 12 columns) and our second column should be 1/3 of the screen width.

The `sm` part of the class denotes that these columns should appear on tablets and larger devices. On anything smaller than a tablet there will only be one column. You have four [options](#) for column size with Bootstrap:

class name	width in pixels	device type
<code>.col-xs-</code>	<code>< 768px</code>	Phones
<code>.col-sm-</code>	<code>>= 768px</code>	Tablets
<code>.col-md-</code>	<code>>= 992 px</code>	Desktops
<code>.col-lg-</code>	<code>>= 1200 px</code>	Large Desktops

class name	width in pixels	device type

Keep in mind that choosing a column size will ensure that the column is available for that size and all sizes larger (i.e., specifying `.col-md` will show the column for desktops and large desktops but not phones or tablets).

After setting up the grid, we used three includes:

```
1 {% include "main/_statusupdate.html" %}
2 {% include "main/_lateststatus.html" %}
3 {% include "main/_jedibadge.html" %}
```

After the last chapter you should be familiar with includes; they just let us include a separate template in our current template. Each of these represents an info box - e.g, the “Report Back to Base” box - which is a separate reusable piece of logic kept in a separate template file. This ensures that your templates stay relatively small and readable, and it makes it easier to understand things.

Let’s set up these sub-templates now.

Template 1: Showing User Info for the Current User

The `*main/_jedibadge.html*` template displays information about the current logged in user:

```
1 <!-- The jedi badge info box, shows user info -->
2 {% load staticfiles %}
3 <section class="info-box" id="user_info">
4     <h1>Jedi Badge</h1>
5     
7     <ul>
8         <li>Rank: {{user.rank}}</li>
9         <li>Name: {{user.name}}</li>
10        <li>Email: {{user.email}}</li>
11        <li><a id="show-achieve" href="#">Show Achievements</a></li>
12    </ul>
13    <p>Click <a href="{% url 'edit' %}">here</a> to make changes to
14        your credit card.</p>
15</section>
```

We start with a section tag that wraps the whole section and gets its styling from a class called `info-box`. Add the following CSS styles to `mec.css`:

```
1 /* Member Page */
2 .info-box {
3     border: 2px solid #000000;
4     margin-bottom: 20px;
5     padding-left: 10px;
6     padding-right: 10px;
7     padding-bottom: 5px;
8     background-color: #eee;
9 }
10 #user_info {
11     text-align: center;
12     margin-left: 20px;
13 }
14 #user_info ul {
15     list-style-type: none;
16     text-align: left;
17 }
18 .member-page {
```

```

19 padding-top: 20px;
20 padding-bottom: 40px;
21 background-color: white;
22 margin-left: 0px;
23 margin-right: 0px;
24 margin-top: -20px;
25 }

```

Mainly we are just setting some spacing and background colors. Nothing too fancy here.

Coming back to the jedi badge info box, there are two things we have changed about the user:

1. The user now has an avatar (which for the time being we are hard-coding as the yoda image) - ``
2. Users now have a rank attribute. If you remember from our user stories we talked about users having a rank of either a ‘youngling’ or ‘padwan’ - `<li>Rank: {{user.rank}}</li>`

Add Rank to Models

Adding the rank attribute to our models is simple: It’s just another column in `payments.models.User` object:

```

1 rank = models.CharField(max_length=50, default="Padwan")

```

We default the value to “Padwan” because all registered users start with that value, and un-registered users, who technically have a rank of “youngling”, won’t see the rank displayed anywhere on the system.

Since we are making changes to the `User` model we need to re-sync our database to add the new rank column to the `payments_user` table. However, if you run `./manage.py syncdb`, Django will essentially do nothing because it cannot add a column to a table that has users in it. It is worth noting that in Django versions prior to 1.7 you would see an error populate. To resolve this, you need to first remove all the users and then the `syncdb` can continue error free.

For right now that is okay: We can delete the users. But if you are updating a system where you need to keep the data (say, a system that is in production), this is not a feasible option. Thankfully, in Django 1.7 the concept of migrations has been introduced. Migrations actually come from a popular Django framework called [South](#), which allows you to update a

table/schema without losing data. An upcoming chapter will cover migrations in more detail. For now we will just drop the database, and then re-sync the tables.

Before you do that, let's talk quickly about users. Since we are developing the membership page (which requires a login), we will need registered users so we can log into the system. Having to delete all the users anytime we make a change to the User model can be a bit annoying, but we can get around this issue by using *fixtures*.

Fixtures

We talked about fixtures in the *Software Craftsmanship* chapter as a means of loading data for unit testing. Well, you can also use fixtures to load initial data into a database. This can make things harder to debug, but in this case since we are doing a lot of work on how the system responds to registered users, we can save ourselves a lot of time by “pre-registering” users.

Since users are in the payments application, we should put our fixture there. Let's create a fixture that runs each time `./manage.py syncdb` is run (which also runs every time our unit test are run):

1. Create a directory called *fixtures* in the *payments* directory.
2. Make sure you have some registered users in your database. Manually add some if necessary. Then run `./manage.py dumpdata payments.User > payments/fixtures/initial_data.json`.

`manage.py dumpdata` spits out JSON for all the data in the database, or in the case above for a particular table. Then we just redirect that output to the file `payments/fixtures/initial_data.json`. Now when you run `./manage.py syncdb`, it will search all of the applications registered in *settings.py* for a *fixtures/initial_data.json* file and load that data into the database that is stored in that file.

Here is an example of what the data might look like, formatted to make it more human-readable:

```
1 [
2   {
3     "pk": 1,
4     "fields": {
5       "last_login": "2014-03-11T08:58:20.136",
6       "rank": "Padwan",
7       "name": "jj",
```

```

8         "password":
9             "pbkdf2_sha256$12000$c8TnAstAXuo4$agxS589FflHZf+C14EHpZr5+EzFtS1V1t
10         "email": "jeremy@realpython.com",
11         "stripe_id": "cus_3e8fBA8rIUEg5X",
12         "last_4_digits": "4242",
13         "updated_at": "2014-03-11T08:58:20.239",
14         "created_at": "2014-03-11T08:58:20.235"
15     },
16     "model": "payments.user"
17 },
18 {
19     "pk": 2,
20     "fields": {
21         "last_login": "2014-03-11T08:59:19.464",
22         "rank": "Jedi Knight",
23         "name": "kk",
24         "password":
25             "pbkdf2_sha256$12000$bEnyOYJkIYWS$jqwLJ4iijmVgPHu9na/JncIi5nJnxb147
26         "email": "k@k.com",
27         "stripe_id": "cus_3e8gyBJlWAu8u6",
28         "last_4_digits": "4242",
29         "updated_at": "2014-03-11T08:59:19.579",
30         "created_at": "2014-03-11T08:59:19.577"
31     },
32     "model": "payments.user"
33 },
34 {
35     "pk": 3,
36     "fields": {
37         "last_login": "2014-03-11T09:12:09.802",
38         "rank": "Jedi Master",
39         "name": "ll",
40         "password":
41             "pbkdf2_sha256$12000$QE2hn0nj0IWm$Ea+IoZMzv6KYV2ycpe+g7afFWi2wPSSya
42         "email": "vader@softworks.com.my",
43         "stripe_id": "cus_3e8tB7Easpo0iJ",
44         "last_4_digits": "4242",
45         "updated_at": "2014-03-11T09:12:10.033",
46         "created_at": "2014-03-11T09:12:10.029"
47     },
48     "model": "payments.user"
49 }

```



```

45         "model": "payments.user"
46     }
47 ]

```

With that, you won't have to worry about re-registering users every time you run a unit test or you resync your database. But if you do use the above data exactly, it will break your unit tests, because our unit tests assume there is no data in the database. In particular, the test `test_get_by_id` in `tests.payments.testUserModel.UserModelTest` should now be failing (among others). Let's fix it really quick:

```

1 def test_get_by_id(self):
2     self.assertEqual(User.get_by_id(self.test_user.id),
                       self.test_user)

```

Before we hard-coded the id to 1, which is okay if you know what the state of the database is... but it's still hard-coding, and it has come back to bite us in the rear. Never again! Now we just use the id of the `test_user` (that we created in the `setUpClass` method), so it doesn't matter how much data we have in the database; this test should continue to pass, time after time.

Update Database

With the fixtures setup, let's update the database.

1. First drop the database from the Postgres Shell:

```

1 DROP DATABASE django_db;

```

2. Then recreate it:

```

1 CREATE DATABASE django_db;

```

3. Finally, run syncdb:

```

1 $ ./manage.py syncdb

```

Make sure to run your tests. Right now you should have three errors since we have not created the `*main/_statusupdate.html*` template yet.

Gravatar Support

Most users are going to want to be able to pick their own avatar as opposed to everybody being Yoda. We could give the user a way to upload an image and store a reference to it in the user table, then just look up the image and display it in the jedi badge info box. But somebody has already done that for us

[Gravatar](#) or Globally Recognized Avatars, is a site that stores an avatar for a user based upon their email address and provides APIs for all of us developers to access that Avatar so we can display it on our site. This is a nice way to go because it keeps you from having to reinvent the wheel and it keeps the user from having to upload yet another image to yet another service.

To use it, let's create a custom tag that will do the work of looking up the gravatar for us. Once the tag is created it will be trivial to insert the gravatar into our `*main/_jedibadge.html*` template.

Create `main/templatetags/main_gravatar.py` and fill it with the following code:

```
1 from django import template
2 from urllib.parse import urlencode
3 import hashlib
4
5 register = template.Library()
6
7
8 @register.simple_tag
9 def gravatar_img(email, size=140):
10     url = get_url(email, size)
11     return '''''' % (url, size, size)
14
15 def get_url(email, size=140):
16     default = ('http://upload.wikimedia.org/wikipedia/en/9/9b/'
17              'Yoda_Empire_Strikes_Back.png')
18
19     query_params = urlencode([('s', str(size)),
20                             ('d', default)])
21
22     return ('http://www.gravatar.com/avatar/' +
23            hashlib.md5(email.lower().encode('utf-8')).hexdigest() +
```

```
24         '?' + query_params)
```

What's happening here?

Let's go through this code a bit.

- Lines 1-8: provide the needed imports and register a tag called `gravatar_img`.
- Lines 9-12: get the gravatar url and then create a nice circular image tag and return it. The image is sized based upon the passed-in value.
- The rest: this is the code that actually constructs the url to use when calling gravatar to get the user's gravatar.

To construct the url we have a few steps to cover. Let's work backwards.

```
1 return ('http://www.gravatar.com/avatar/' +
2         hashlib.md5(email.lower().encode('utf-8')).hexdigest() +
3         '?' + query_params)
```

The base url is `http://www.gravatar.com/avatar/`. To that we add the user's email address hashed with md5 (which is required by gravatar) along with the `query_params`.

We pass in two query parameters:

- `s`: the size of the image to return
- `d`: a default image that is returned if the email address doesn't have a gravatar account

The code to do that is here:

```
1 default = ('http://upload.wikimedia.org/wikipedia/en/9/9b/'
2            'Yoda_Empire_Strikes_Back.png')
3
4 query_params = urlencode([('s', str(size)),
5                           ('d', default)])
```

`urlencode` is important as it provides any necessary escaping/character mangling to ensure you have a proper query string. For our default image, it must be available on the web somewhere, so we've just picked a random Star Wars image. There are several other query string parameters that gravatar accepts, and they are all explained in detail [here](#).

With that we should now have a functioning tag called `gravatar_img` that takes in an email address and an optional size and returns the appropriate `img` markup for us to use. Let's now use this in our `_jedibadge` template.

```

1 <!-- The jedi badge info box, shows user info -->
2 {% load staticfiles %}
3 {% load main_gravatar %}
4 <section class="info-box" id="user_info">
5   <h1>Jedi Badge</h1>
6   {% gravatar_img user.email %}
7   <ul>
8     <li>Rank: {{user.rank}}</li>
9     <li>Name: {{user.name }}</li>
10    <li>Email: {{user.email }}</li>
11    <li><a id="show-achieve" href="#">Show Achievements</a></li>
12  </ul>
13  <p>Click <a href="{% url 'edit' %}">here</a> to make changes to
    your credit card.</p>
14 </section>

```

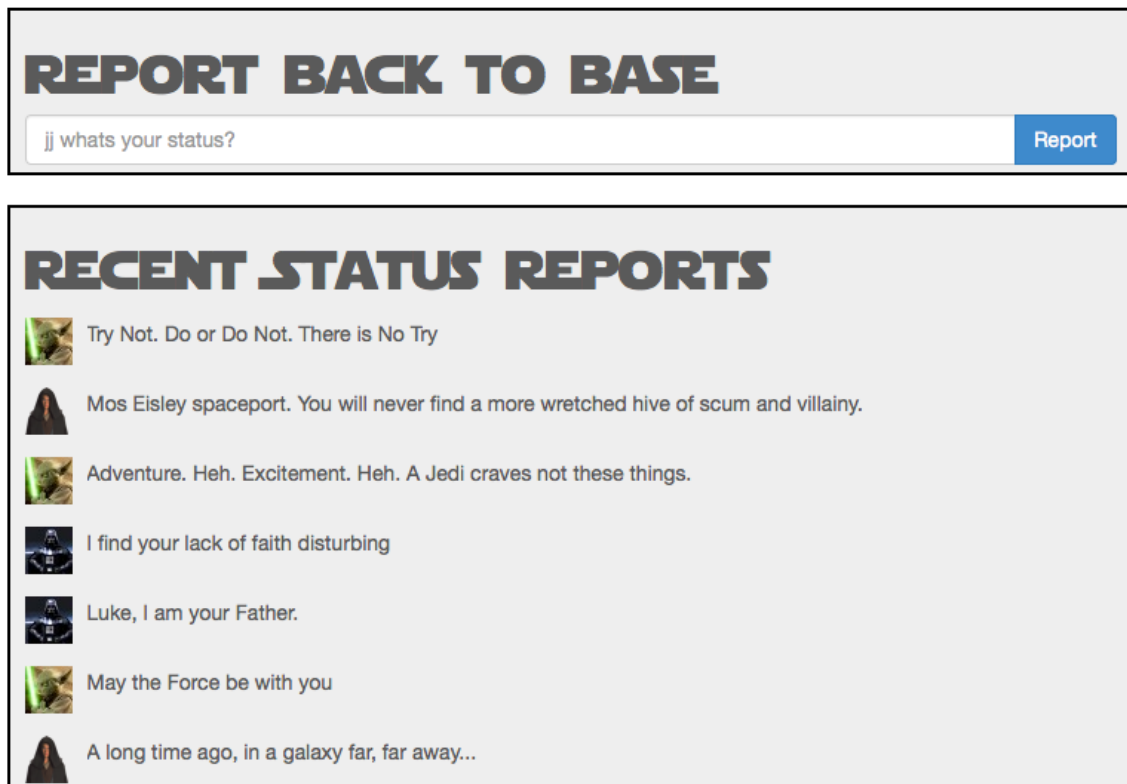
Notice we have changed line 3 and line 6 from our earlier template: Line 3 loads our custom tag library and line 6 calls it, passing in `user.email`. Now we have gravatar support - and it only took a handful of lines!

NOTE: There are a number of gravatar plugins available on GitHub, most of which are basically the same thing we just implemented. While you shouldn't re-invent the wheel, there's not much point in utilizing an external dependency for something that is this straight-forward. However if/when we need more than basic gravatar support, it may be worth looking into some of the pre-existing packages.

To finalize the gravatar support, we better re-run our tests and make sure nothing fails (aside for the three previous errors, of course), as well as add some new tests for the gravatar tag. Since this is review, add this on your own - it's always good to get some extra practice.

Template 2: Status Updating and Reporting

It's pretty common these days for membership sites to have some sort of status update functionality, where users can post their status or whatever is on their minds and others can see a list of the most recent status updates. The screenshot below shows the two info boxes that will participate in the status updating / reporting functionality.



REPORT BACK TO BASE

jj whats your status? [Report](#)

RECENT STATUS REPORTS

-  Try Not. Do or Do Not. There is No Try
-  Mos Eisley spaceport. You will never find a more wretched hive of scum and villainy.
-  Adventure. Heh. Excitement. Heh. A Jedi craves not these things.
-  I find your lack of faith disturbing
-  Luke, I am your Father.
-  May the Force be with you
-  A long time ago, in a galaxy far, far away...

Figure 10.2: Status Reporting

The top info box allows users to submit status updates and the bottom info box shows the history of status updates. Let's see how these are implemented.

First for the status updater. The HTML template `templates/main/_statusupdate.html` is shown below:

```
1 <!-- represents the status update info box -->
2 <section class="info-box" id="updates">
3   <h1>Report back to base</h1>
4   <form accept-charset="UTF-8" action="{% url 'report' %}"
```

```

5     role="form" method="post">{% csrf_token %}
6     <div class="input-group">
7         <input id="status" type="text" class="form-control"
8             name="status"
9             placeholder="{% user.name %} whats your status?">
10        <span class="input-group-btn">
11            <button class="btn btn-primary"
12                type="submit">Report</button>
13        </span>
14    </div>
15 </form>
16 </section>

```

It's just a simple form that POSTS the report URL; be sure to add the URL to `urls.py` as:

```

1 url(r'^report$', 'main.views.report', name="report"),

```

To fully understand the view function, `main.views.report`, we need to first have a look at the model, which the view function relies on. The listing for `user.models.StatusReport` is below:

```

1 class StatusReport(models.Model):
2     user = models.ForeignKey('payments.User')
3     when = models.DateTimeField(auto_now_add=True)
4     status = models.CharField(max_length=200)

```

Update the model, and then run `syncdb`.

Just three columns here:

1. user- a foreign key into the 'User' table in `payments.models`
2. when - a time stamp defaulted to now
3. status - the actual status message itself

For the view functionality we could have created a Django form like we did with the sign-in or registration function, but that's not strictly required, especially since we aren't doing any validation. So let's use a straight-forward view function which is implemented in `main.views.report`:

```

1 def report(request):
2     if request.method == "POST":
3         status = request.POST.get("status", "")

```

```

4         #update the database with the status
5         if status:
6             uid = request.session.get('user')
7             user = User.get_by_id(uid)
8             StatusReport(user=user, status=status).save()
9
10        #always return something
11        return index(request)

```

Update the imports:

```

1 from main.models import MarketingItem, StatusReport

```

A brief description of the important lines:

- Line 2: only respond to POST requests
- Line 3: pull the status out of the request, which corresponds to name of status on our form:

```

1 <input id="status" type="text" class="form-control" name="status"
2     placeholder="{{user.name}} what's your status?">

```

- Line 7-8: grab the current logged-in user from the request
- Line 9: update the StatusReport table with the update the user just submitted, then return the index page, which will in turn return user.html since we have a logged-in user.

The last line will in effect cause the ‘Recent Status Reports’ info box to update with the newly posted status, with the following updates to the main.views.index function:

```

1 def index(request):
2     uid = request.session.get('user')
3     if uid is None:
4         #main landing page
5         market_items = MarketingItem.objects.all()
6         return render_to_response(
7             'main/index.html',
8             {'marketing_items': market_items}
9         )
10    else:

```

```

11     #membership page
12     status = StatusReport.objects.all().order_by('-when')[:20]
13     return render_to_response(
14         'main/user.html',
15         {'user': User.get_by_id(uid), 'reports': status},
16         context_instance=RequestContext(request),
17     )

```

Be sure to update the imports as well:

```

1 from django.shortcuts import render_to_response, RequestContext
2 from payments.models import User
3 from main.models import MarketingItem, StatusReport

```

The main difference from our previous version of this function is:

```

1 status = StatusReport.objects.all().order_by('-when')[:20]

```

This line grabs a list of twenty status reports ordered by posted date in reverse order. Please keep in mind that even though we are calling `objects.all()` we are never actually retrieving all records from the database for this table. Django's ORM by default uses [Lazy Loading](#) for its query sets, meaning Django won't actually query the database until you access the data from the query set - which, in this case, is at the point of slicing, `[:20]`. Thus we will only pull at most 20 rows from the database.

Sanity Check - a brief aside

Don't believe me? Test it out.

We can prove that we are only pulling up to twenty results by turning on logging and running the query. One way to do this is to fire up the Django shell by typing `./manage.py shell`, then running the following code:

```

1 >>> from main.models import StatusReport
2 >>> q = StatusReport.objects.all().order_by('-when')[:20]
3 >>> print(q.query)
4 SELECT "main_statusreport"."id", "main_statusreport"."user_id",
   "main_statusreport"."when", "main_statusreport"."status" FROM
   "main_statusreport" ORDER BY "main_statusreport"."when" DESC
   LIMIT 20

```

As you can see from line 4, the exact SQL that Django executes is outputted for you to read and it does include a `LIMIT 20` at the end. *In general, dropping down into the shell and*

outputting the exact query is a good sanity check to quickly verify that the ORM is executing what you think it is executing.

Django 1.8 QuerySets - another brief aside

In Django you have always been able to create a custom QuerySet that can make it easier to understand the intent of your code. For example let's create a latest query for the StatusReports object. To do that in Django 1.8 we first create a StatusReportQuerySet class like so:

```
1 class StatusReportQuerySet(models.QuerySet):
2     def latest(self):
3         return self.all().order_by('-when')[:20]
```

Then we hook it up to our StatusReport model by adding one line to our StatusReport class:

```
1 objects = StatusReportQuerySet.as_manager()
```

Once that is setup we can use our new query set in place of our main.views.index function by changing this line:

```
1 status = StatusReport.objects.all().order_by('-when')[:20]
```

to:

```
1 status = StatusReport.objects.latest()
```

What's the difference?

Well, prior to Django 1.7 we had to write:

```
1 status = StatusReport.objects.get_query_set().latest()
```

At the very least Django 1.7 saves us a *few* keystrokes. In general though, there is some value in using custom query sets because it can make the intent of the code clearer. It's pretty easy to guess that latest() returns the latest StatusReports. This is especially advantageous when you have to write some custom SQL or you have a complex query that isn't easy to understand what it does. By wrapping it in a query_set you're creating a kind of self-documenting code that makes it easier to maintain, and now in 1.7 takes less key strokes as well.

Don't forget we can also verify that our new query is doing what we expect by using the same technique we used previously, e.g., run `./manage.py shell` import the model and then run:

```
1 print (StatusReport.objects.latest().query)
```

And this should give the same output as our original query.

Back to the task a hand

Coming back to our view function:

```
1 def index(request):
2     uid = request.session.get('user')
3     if uid is None:
4         #main landing page
5         market_items = MarketingItem.objects.all()
6         return render_to_response(
7             'main/index.html',
8             {'marketing_items': market_items}
9         )
10    else:
11        #membership page
12        status = StatusReport.objects.all().order_by('-when')[:20]
13        return render_to_response(
14            'main/user.html',
15            {'user': User.get_by_id(uid), 'reports': status},
16            context_instance=RequestContext(request),
17        )
```

Lines 10-14:

```
1 return render_to_response(
2     'main/user.html',
3     {'user': User.get_by_id(uid), 'reports': status},
4     context_instance=RequestContext(request),
5 )
```

The difference here is that we are now returning `context_instance=RequestContext(request)`. This is required because in our template we added the `{% csrf_token %}` to avoid cross-site scripting vulnerabilities. Thus we need the view to return the `RequestContext` which will be picked up by Django's `CsrfViewMiddleware` and used to prevent against cross-site scripting attacks.

With all that, we have the functionality to allow a user to submit a status update. Now we need to display the status updates.

Template 3: Displaying Status Updates

If you glance back up at the code listing for `main.views.index` you will notice that we already included the code to return the list of Status Reports. We just need to display it with a template: `templates/main/_lateststatus.html`:

```
1 <!-- list of latest status messages sent out by all users of the
   site -->
2 {% load staticfiles %}
3 {% load main_gravatar %}
4 <section class="info-box" id="latest_happenings">
5   <h1>Recent Status Reports</h1>
6   {% for report in reports %}
7     <div class="media">
8       <div class="media-object pull-left">
9         
11       </div>
12       <div class="media-body">
13         <p>{{ report.status }}</p>
14       </div>
15     </div>
16   {% endfor %}
17 </section>
```

Take note of:

- Line 6: loops through the list of status reports returned from the database (no more than twenty).
- Line 9: new template tag `gravatar_url` that we will explain in a minute.
- Line 13: displays the status update.
- Line 6, Line 7, and Line 11: the classes `media`, `media-object`, and `media-body` are all supplied by Bootstrap and designed to provide a list of items with images next to them.

That gives us the list of the most recent status updates. The only thing left to do is to explain the new `gravatar_url` template tag.

If you recall from our earlier `gravatar_img` tag found in `main/templatetags/main_gravatar.py`, internally it called a function `get_url` to get the gravatar URL. What we have done is simply

exposed that `get_url` function by adding the `@register.simple_tag` decorator, and changed the name of the function to `gravatar_url` to fit in with our template tag naming conventions discussed in the last chapter.

So the code now looks like this:

```
1 @register.simple_tag
2 def gravatar_url(email, size=140):
3     default = ('http://upload.wikimedia.org/wikipedia/en/9/9b/'
4               'Yoda_Empire_Strikes_Back.png')
5
6     #mainly for unit testing with a mock object
7     if not(isinstance(email, str)):
8         return default
9
10    query_params = urlencode([('s', str(size)),
11                             ('d', default)])
12
13    return ('http://www.gravatar.com/avatar/' +
14            hashlib.md5(email.lower().encode('utf-8')).hexdigest() +
15            '?' + query_params)
```

And the final templatetag:

```
1 from django import template
2 from urllib.parse import urlencode
3 import hashlib
4
5 register = template.Library()
6
7
8 @register.simple_tag
9 def gravatar_img(email, size=140):
10    url = gravatar_url(email, size)
11    return '''''' % (url, size, size)
14
15 @register.simple_tag
16 def gravatar_url(email, size=140):
17    default = ('http://upload.wikimedia.org/wikipedia/en/9/9b/'
```

```

18         'Yoda_Empire_Strikes_Back.png')
19
20     #mainly for unit testing with a mock object
21     if not(isinstance(email, str)):
22         return default
23
24     query_params = urlencode([('s', str(size)),
25                               ('d', default)])
26
27     return ('http://www.gravatar.com/avatar/' +
28            hashlib.md5(email.lower().encode('utf-8')).hexdigest() +
29            '?' + query_params)

```

That should give us the basic functionality for our members page. You'll be adding some more functionality in the exercises to give you a bit of practice, and then in the next chapter we'll look at switching to a REST-based architecture.

Run your automated tests:

```

1 $ ./manage.py test ../tests
2 Creating test database for alias 'default'...
3 .....
4
5 .....
6 .
7 .....
8 -----
9 Ran 31 tests in 1.434s
10
11 OK
12 Destroying test database for alias 'default'...

```

Then manually test:

1. Register a new user
2. Ensure that the user can update their status
3. Logout
4. Login as a different user
5. Ensure you can see the status update posted by the previous user

Exercises

1. Our User Story **US3 Main Page** says that the members page is a place for announcements and to list current happenings. We have implemented user announcements in the form of status reports, but we should also have a section for system announcements/current events. Using the architecture described in this chapter, create an `Announcements info_box` to display system-wide announcements?
2. You may have noticed that in the Jedi Badge box there is a list achievements link. What if the user could get achievements for posting status reports, attending events, and any other arbitrary action that we create in the future? This may be a nice way to increase participation, because everybody likes badges, right? Go ahead and implement this achievements feature. You'll need a model to represent the badges and a link between each user and the badges they own (maybe a `user_badges` table). Then you'll want your template to loop through and display all the badges that the given user has.

Chapter 11

REST

Remember the status updates features that we implemented in the last chapter? It works but we can do better.

The main issue is that when you submit a status, the entire page must reload before you see your updated status. This is so web 1.0. We can't have that, can we? The way to improve this and remove the screen refresh is by using [AJAX](#), which is a client-side technology for making asynchronous requests that don't cause an entire page refresh. Often AJAX is coupled with a server-side API to make it much easier to get the data you need from JavaScript.

One of the most popular server-side API styles in modern web programming is REST.

REST stands for *Representational State Transfer*, which to most people means absolutely nothing. Let's hazard a definition: *REST is a stateless "architectural style" generally run over HTTP that relies on consistent URL names and HTTP verbs (GET, POST, DELETE, etc.) to make it easy for various client programs to simply and consistently access and manipulate resources from a server in a standard way.*

REST doesn't actually specify what format should be used for data exchange, but most new REST APIs are implemented with JSON. This is great for us since JSON is extremely simple to work with in Python, as a Python dictionary is basically JSON out of the box. So we will also use JSON in our examples here.

When implementing a REST API, there are a number of ways you could choose to implement it, and a lot of debate about which is the best way. We'll focus on a standard method otherwise we might never finish this chapter!

Structuring a REST API

There are three key points one must adhere to when implementing a nice REST-based API...

Resources should be the main concern of your REST architecture and URLs should be structured accordingly.

In REST your URL structure defines your API and thus how clients interact with your API server. The URL structure should be based around the resources your server provides. Generally there are two ways to access a resources in REST, Through the **Collection** URI ([Uniform Resource Identifiers](#)) and through the **Member** URI (also commonly referred to as the element URI).

For example, to represent our status updates as a REST API, we would use the following URIs:

For a collection:

```
1 http://<site-name>/status_reports/
```

And for the member (i.e. an individual status_report):

```
1 http://<site-name>/status_reports/2
```

(where 2 is the id of the status_report object.)

It is also common to prefix the RESTful URLs with an api/ directory - i.e:

```
1 http://<site-name>/api/status_reports/
```

This helps to differentiate between the REST API and URLs that just return HTML.

Finally, to be good web citizens, it's a good idea to put the version of your API into your URL structure so you can change it in future versions without breaking everybody's code. Doing so would have your URLs looking like:

```
1 http://<site-name>/api/v1/status_reports/
```

REST is built on top of HTTP; thus, it should use the appropriate HTTP verbs.

The following tables describes the HTTP Verbs and how they map to a typical REST API:

For a Collection URI - i.e., `http://<site-name>/api/v1/status_reports/`

HTTP Verb	Typical Use
GET	Returns the entire collection as a list, potentially with related information
PUT	Replaces the entire collection with the passed-in collection
POST	Creates a new entry in the collection, generally with an auto-assigned ID
DELETE	Blows away the entire collection... Bye Bye

For a Member URI - i.e., `http://<site-name>/api/v1/status_reports/2`

HTTP Verb	Typical Use
GET	Returns the member and any related data - status report with an id of 2
PUT	Replaces the addressed member with the one passed in
POST	<i>USUALLY NOT USED</i> : POST to the Collections URI instead
DELETE	Deletes the member with corresponding id

A couple of notes are worth mentioning. PUT is meant to be idempotent, which means you can expect the same result every time you call it. That is why it implements an insert. It either updates an existing member or inserts a new one if the member does not exist; the end result is that the appropriate member will exist and have data equal to the data passed in.

POST on the other hand is not idempotent and is thus used to create things.

There is also an HTTP verb called PATCH which allows for partial updates; there is a lot of debate about if it should be used and how to use it. Many (err most) developers ignore it since you can create a new item with POST and update with PUT. We'll ignore it as well since it does over-complicate things.

Use HTTP return codes appropriately

HTTP has a [rich set of status codes](#), and they should be used to help convey the result of an API call. For example, successful calls can return a status code of 200 and errors/failures can use the 4xx or the 5xx error codes. If you want to provide more specific information, you can include it in the details portion of your response.

Those are three main concerns with regards to designing a RESTful API. Authentication is a fourth concern, but let's come back to authentication a bit later. *Let's start by first designing what our API should look like, and then implementing it.*

It is important to note that we don't necessarily need to expose all the resources of our system to the REST API; just the ones we care to share. Put another way, we should *only* expose the resources that other developers care about, who would use the data in some meaningful way.

Conversely, oftentimes when thinking through how to design a REST API, developers are stuck with the idea that they need more verbs to provide the access they want to provide. While this is sometimes true, it can usually be solved by exposing more resources. The canonical example of this is login. Rather than implementing something like:

```
1 GET api/v1/users/1/login
```

Consider login as a separate resource. Or better yet, call it a session:

```
1 POST api/v1/session - login and return a session ID  
2 DELETE api/v1/session/sessionID - log out
```

This sticks more strictly to the REST definition. Of course with REST there are only suggestions/conventions, and nothing will stop you from implementing whatever URL structure you wish. This can get ugly if you don't stick with the conventions outlined. Only deviate from them if you have a truly compelling reason to do so.

REST for MEC

For Mos Eisley's Cantina, let's start off with the simplest REST framework we can get away with and add to it as we go. The one issue we are trying to fix now is the posting of status updates without having to reload the page, so let's just create the status API, and then we will expose other resources as necessary.

Following the design considerations discussed above, we will use the following URL structure:

Collection

- GET - `api/v1/status_reports` - returns ALL status reports
- POST - `api/v1/status_reports` - creates a status update and returns the id

Member

- GET - `api/v1/status_reports/<id>` - returns a particular report status by id

That's all we need for now. You can see that we could simply add a number of URL query strings - i.e., user, date, etc. - to provide further query functionality, and maybe a DELETE as well if you wanted to add additional functionality, but we don't need those yet.

Django REST Framework (DRF)

Before jumping in, it's always worth weighing the cost of learning a new framework versus the cost of implementing things yourself. If you had a very simple REST API that you needed to implement, you could develop it in a few lines of code:

```
1 from django.http import HttpResponse
2 from django.core import serializers
3
4 def report(request):
5     if request.method == "GET":
6         status = StatusReport.objects.all().order_by('-when')[:20]
7
8         return HttpResponse(
9             serializers.serialize("json", status),
10            content_type='application/json',
11            context_instance=RequestContext(request)
12        )
```

There you go: You now have a simple and extremely naive REST API with one method.

Of course that isn't going to get you very far. We could abstract the JSON functionality into a mixin class, and then by using class-based views make it simple to use JSON on all of our views. This technique is actually described in the [Django documentation](#).

```
1 from django.core import serializers
2 from django.http import HttpResponse
3
4 class JSONResponseMixin(object):
5     """
6     A mixin that can be used to render a JSON response.
7     """
8     def render_to_json_response(self, context, **response_kwargs):
9         """
10         Returns a JSON response, transforming 'context' to make the
11         payload.
12         """
13         return HttpResponse(
14             serializers.serialize("json", context),
15             content_type='application/json',
16             **response_kwargs
17         )
```

While this is a bit better, it still isn't going to help a whole lot with authentication, API discovery, handling POST parameters, etc. To that end, we are going to use a framework to help us implement our API. The two most popular ones as of writing are [Django REST Framework](#) and [Tastypie](#). Which should you use? They really are about the same in terms of functionality, so it's up to you.

It's worth nothing (or maybe not) that we chose Django REST Framework since it has a "cooler" logo.

If you're unsure (and not sold by the logo argument), you can look at the popularity of the package, how active it is, and when it's last release date to help you decide. There are two great places to find this information: [Django Packages](#) and [Github](#). Look for the number of watchers, stars, forks, and contributors. Check out the issues page as well to see if there are any reported show stopper issues.

Using this information, especially when you are unfamiliar with the project can greatly aid in the decision making process of which project to use.

That said, let's jump right into the meat and potatoes.

Installation

The first thing to do is install DRF:

```
1 $ pip install djangorestframework
```

Also, let's update our *requirements.txt* file to include our new dependency:

```
1 $ pip freeze > requirements.txt
```

Your file should look like this:

```
1 Django==1.8.2
2 django-embed-video==0.11
3 djangorestframework==3.1.1
4 mock==1.0.1
5 psycpg2==2.5.3
6 requests==2.3.0
7 stripe==1.9.2
```

Serializers

DRF provides a number of tools you can use. Let's start with the most fundamental: the [serializer](#).

Serializers provide the capability to translate a Model or QuerySet to/from JSON. This is what we just saw with the `django.core.serializers` above. DRF serializers do more-or-less the same thing, but they also hook directly into all the other DRF goodness.

Let's create a serializer for the StatusReport object. Create a new file called `main/serializers.py` with the following content:

```
1 from rest_framework import serializers
2 from main.models import StatusReport
3
4
5 class StatusReportSerializer(serializers.Serializer):
6     id = serializers.ReadOnlyField()
7     user = serializers.StringRelatedField()
8     when = serializers.DateTimeField()
9     status = serializers.CharField(max_length=200)
10
11     def create(self, validated_data):
12         return StatusReport(**validated_data)
13
14     def update(self, instance, validated_data):
15         instance.user = validated_data.get('user', instance.user)
16         instance.when = validated_data.get('when', instance.when)
17         instance.status = validated_data.get('status',
18                                             instance.status)
19         instance.save()
20         return instance
```

If you've worked with Django Forms before, the StatusReportSerializer should look somewhat familiar. It functions like a Django Form. You first declare the fields to include in the serializer, just like you do in a form (or a model, for that matter). The fields you declare here are the fields that will be included when serializing/deserializing the object.

Two quick notes about the fields we declared for our serializer:

1. We declared an `id` (primary key) field as type `serializers.ReadOnlyField()`. This is a read-only field that cannot be changed, but it needs to be there since it maps to our `id` field, which we will need on the client side for updates.
2. Our `user` field is of type `serializers.StringRelatedField()`. This represents a many-to-one relationship and says that we should serialize the user object by using its `__str__` function. In our case this is the user's email address.

There are two functions, `create()` and `update()`. They do pretty much exactly what you would think: Create a new model instance from serialized data, or update an existing model instance. Any custom creation logic you may need can be put in these functions. By “creation logic” we’re not referring to the code that would normally be put into your `__init__` function, because that is already going to be called. Creation logic is simply any logic you may need to include when creating the object from a deserialized JSON string (i.e., a python dictionary).

For a single object serializer like the one shown above, there isn’t likely to be much extra logic, but there is nothing preventing you from writing a serializer that works on an entire object chain. This is sometimes helpful when dealing with nested or related objects.

Tests

Let’s write some tests for the serializer to ensure it does what we want it to do. You do remember the Test Driven Development chapter, right?

Implementing a new framework/reusable app is another great example of where Test Driven Development shines. It gives you an easy way to get at the underpinnings of a new framework, find out how it works, and ensure that it does what you need it to. It also gives you an easy way to try out different techniques in the framework and quickly see the results (as we will see throughout the rest of this chapter).

Serialization

The DRF serializers actually work in two steps: It first converts the object into a Python dictionary and then converts the dictionary into JSON. Let’s test that those two steps work as we expect.

Create a file called `../tests/main/testSerializers.py`:

```
1 from django.test import TestCase
2 from main.models import StatusReport
3 from payments.models import User
4 from main.serializers import StatusReportSerializer
5 from rest_framework.renderers import JSONRenderer
6 from collections import OrderedDict
7 from rest_framework.parsers import JSONParser
8 from django.utils.six import BytesIO
9
10
11 class StatusReportSerializer_Tests(TestCase):
```

```

12
13     @classmethod
14     def setUpTestData(cls):
15         cls.u = User(name="test", email="test@test.com")
16         cls.u.save()
17
18         cls.new_status = StatusReport(user=cls.u, status="hello
19             world")
20         cls.new_status.save()
21
22         cls.expected_dict = OrderedDict([
23             ('id', cls.new_status.id),
24             ('user', cls.u.email),
25             ('when', cls.new_status.when.isoformat()),
26             ('status', 'hello world'),
27         ])
28
29     def test_model_to_dictionary(self):
30         serializer = StatusReportSerializer(self.new_status)
31         self.assertEqual(self.expected_dict, serializer.data)

```

There is a fair amount of code here, so let's take it a piece at a time. The first part of this file-

```

1 from django.test import TestCase
2 from main.models import StatusReport
3 from payments.models import User
4 from main.serializers import StatusReportSerializer
5 from rest_framework.renderers import JSONRenderer
6 from collections import OrderedDict
7 from rest_framework.parsers import JSONParser
8
9
10 class StatusReportSerializer_Tests(TestCase):
11
12     @classmethod
13     def setUpTestData(cls):
14         cls.u = User(name="test", email="test@test.com")
15         cls.u.save()
16
17         cls.new_status = StatusReport(user=cls.u, status="hello
18             world")

```



```

18         cls.new_status.save()
19
20         cls.expected_dict = OrderedDict([
21             ('id', cls.new_status.id),
22             ('user', cls.u.email),
23             ('when', cls.new_status.when.isoformat()),
24             ('status', 'hello world'),
25         ])

```

-is responsible for all the necessary imports and for setting up the user/status report that we will be working with in our tests.

The first test-

```

1 def test_model_to_dictionary(self):
2     serializer = StatusReportSerializer(self.new_status)
3     self.assertEqual(self.expected_dict, serializer.data)

```

-verifies that we can take our newly created object `self.new_status` and serialize it to a dictionary. This is what our serializer class does. We just create our serializer by passing in our object to serialize and then call `serialize.data`, and out comes the dictionary we want.

Run the test... boom! Hashtag FAIL. If you look at the results you should see an error about the two dictionaries not being equal. You should also see a warning about naive datetime.

A Brief Aside about Timezone support

JSON, by convention, should use a datetime format called `iso-8601`. This is a universal datetime format that includes timezone information. In python you can output this format by calling `isoformat` on your string. The specification format will output the date then the time then the timezone but if the timezone is UTC it will output `Z` instead of the timezone so it's something like: `2013-01-29T12:34:56.123Z` however python's `isoformat` will output `2013-01-29T12:34:56.123+00:00` for the same date. This makes our test fail if we are using UTC, which is the default timzone. For a quick fix, let's just change our `setUpClass` method as such, to use the properly formatted timezone in the `expected_dict` so the function now will look like:

```

1 @classmethod
2 def setUpTestData(cls):
3     cls.u = User(name="test", email="test@test.com")
4     cls.u.save()
5
6     cls.new_status = StatusReport(user=cls.u, status="hello world")
7     cls.new_status.save()

```

```

8
9     when = cls.new_status.when.isoformat()
10    if when.endswith('+00:00'):
11        when = when[:-6] + 'Z'
12
13    cls.expected_dict = OrderedDict([
14        ('id', cls.new_status.id),
15        ('user', cls.u.email),
16        ('when', when),
17        ('status', 'hello world'),
18    ])

```

Notice in lines 9-11 that we grab the time (when) and convert it to the format that DRF expects. Then we use that value to populate our `expected_dict`. After making this change our test should pass.

Dictionary to JSON

The next step in the object to JSON conversion process is converting the dictionary to JSON:

```

1 def test_dictionary_to_json(self):
2     serializer = StatusReportSerializer(self.new_status)
3     content = JSONRenderer().render(serializer.data)
4     expected_json = JSONRenderer().render(self.expected_dict)
5     self.assertEqual(expected_json, content)

```

To convert to JSON you must first call the serializer to convert to the dictionary, and then call `JSONRenderer().render(serializer.data)`. This instantiates the `JSONRenderer` object and passes it a dictionary to render as JSON. The render function calls `json.dumps` and ensures the output is in the proper unicode format. Now we have an option of how we want to verify the results. We could build the expected JSON string and compare the two strings.

One drawback here is that you often have to play around with formatting the string exactly right, especially when dealing with date formats that get converted to the JavaScript date format.

Another option (which we used) is to create the dict that we should get from the serializer (and we know what the dict is because we just ran that test), then convert that dict to JSON and ensure the results are the same as converting our `serializer.data` to JSON. This also has its issues, as the order in which the attributes are placed in the results JSON string is important, and dictionaries don't guarantee order. So we have to use `OrderedDict`, which

will ensure our dictionary preserves the order of which the keys were inserted. After all that, we can verify that we are indeed converting to JSON correctly.

Run the tests:

```
1 $ ./manage.py test ../tests
```

So serialization seems to work correctly.

How about deserializaiton?

Deserialization

We need to run the opposite test:

```
1 def test_json_to_StatusReport(self):
2
3     json = JSONRenderer().render(self.expected_dict)
4     stream = BytesIO(json)
5     data = JSONParser().parse(stream)
6
7     #where calling update to pass in existing object, plus data to
8     #update
9     serializer = StatusReportSerializer(self.new_status, data=data)
10    self.assertTrue(serializer.is_valid())
11
12    status = serializer.save()
13    self.assertEqual(self.new_status.id, status.id)
14    self.assertEqual(self.new_status.status, status.status)
15    self.assertEqual(self.new_status.when, status.when)
16    self.assertEqual(self.new_status.user, status.user)
```

All should pass.

Of course we could replace all the asserts at the end with:

```
1 self.asesrtEqual(self.new_status, status)
```

But I just wanted to be explicit and show that each field was infact being deserialized correctly.

Now what about creating a new serializer instance from json? Let's write another test:

```
1 def test_json_to_new_StatusReport(self):
2     json = JSONRenderer().render(self.expected_dict)
3     stream = BytesIO(json)
```

```

4     data = JSONParser().parse(stream)
5
6     serializer = StatusReportSerializer(data=data)
7     self.assertTrue(serializer.is_valid())
8
9     status = serializer.save()
10    self.assertEqual(self.new_status.status, status.status)
11    self.assertIsNotNone(status.when)
12    self.assertEqual(self.new_status.user, status.user)

```

You probably already guessed it, but this test is going to fail.

```

1  Traceback (most recent call last):
2    File "/testSerializers.py", line 61, in test_json_to_StatusReport
3      self.assertEqual(self.new_status.user, status.user)
4    File "/site-packages/django/db/models/fields/related.py", line
5      6088, in __get__
6      "%s has no %s." % (self.field.model.__name__, self.field.name)
django.db.models.fields.related.RelatedObjectDoesNotExist:
    StatusReport has no user.

```

The important part is `RelatedObjectDoesNotExist`; that's the error you get when you try to look up a model object from the db and it doesn't exist. Why don't we have a user object associated with our `StatusReport`? Remember in our serializer, we used this line:

```

1  user = serializers.StringRelatedField()

```

This means that we serialize the user field by calling its `__str__` function, which just returns an email. Then when we deserialize the object, our create function is called, but since `StringRelatedField` is by read only then the user won't get passed in. Nor will the id for that matter (because it's set to be a `ReadOnlyField`). We'll come back to the solution for this in just a second.

First, let's talk about `ModelSerializers`.

ModelSerializers

Our initial `StatusReportSerializer` contains a ton of boilerplate code. We're really just copying the field from our model. Fortunately, there is a better way. Enter [ModelSerializers](#). If we rewrite our `StatusReportSerializer` using DRF's `ModelSerializer`, it looks like this:

```

1 from rest_framework import serializers
2 from main.models import StatusReport
3
4
5 class StatusReportSerializer(serializers.ModelSerializer):
6
7     class Meta:
8         model = StatusReport
9         fields = ('id', 'user', 'when', 'status')

```

Wow! That's *a lot* less code. Just like Django gives you a Form and a ModelForm, DRF gives you a Serializer and a ModelSerializer. And just like Django's ModelForm, the ModelSerializer will get all the information it needs from the model. You just have to point it to the model and tell it what fields you want to use.

The only difference between these four lines of code in the ModelSerializer and the twelve lines of code in our Serializer is that the Serializer serialized our user field using the id instead of the email address. This is not exactly what we want, but it does mean that when we deserialize the object from JSON, we get our user relationship back! To verify that, and to update our tests to account for the user being serialized by id instead of email, we only have to change our `cls.expected_dict` to look like this in *testSerializers.py*:

```

1 cls.expected_dict = OrderedDict([
2     ('id', cls.new_status.id),
3     ('user', cls.u.id),
4     ('when', cls.new_status.when),
5     ('status', 'hello world'),
6 ])

```

Almost there. If you recall, our `main.models.index` uses the user's email address so we can do the gravatar lookup. How do we get that email address?

We create a custom relationship field in *serializers.py*:

```

1 from payments.models import User
2
3 class RelatedUserField(serializers.RelatedField):
4
5     read_only = False
6
7     def to_representation(self, value):
8         return value.email
9

```

```

10     def to_internal_value(self, data):
11         return User.objects.get(email=data)

```

- **Line 3:** We inherit from the `serializers.RelatedField` which is the base relationship field for DRF. To fully implement this field type we need two function `to_representation` - for serialization, and `to_internal_value` - for deserialization
- **Line 5:** If you want deserialization to occur, you have to set the `read_only = False` attribute, in effect saying this field should be read / write
- **Line 7-8:** The function `to_representation(self, value)` will receive the target of the field (in our case the `User`), and it's our job to return the serialized representation we want (in our case the email address).
- **Line 10-11:** The function `to_internal_value(self, data)` will receive the JSON value from the field (as the `data` parameter), and it's our job to return the object we want. We do this by looking it up in the users table.

Full update:

```

1  from rest_framework import serializers
2  from main.models import StatusReport
3  from payments.models import User
4
5  class RelatedUserField(serializers.RelatedField):
6
7      read_only = False
8
9      def to_representation(self, value):
10         return value.email
11
12     def to_internal_value(self, data):
13         return User.objects.get(email=data)
14
15 class StatusReportSerializer(serializers.ModelSerializer):
16     user = RelatedUserField(queryset=User.objects.all())
17
18     class Meta:
19         model = StatusReport
20         fields = ('id', 'user', 'when', 'status')

```

Take note of line 16. When declaring a `RelatedField` in our serializer queryset is a required parameter. The intent is to make it explicit where the data is coming from for this field.

Update the `'cls.expected_dict'` again, back to using the user's email:

```
1 cls.expected_dict = OrderedDict([
2     ('id', cls.new_status.id),
3     ('user', cls.u.email),
4     ('when', cls.new_status.when),
5     ('status', 'hello world'),
6 ])
```

Now run the tests. They should pass. Perfect.

There are several other ways to manage/display relationships in DRF; for more information on these, check out [the docs](#).

Also have a look at the `SlugRelatedField` as it basically does the same thing we just implemented.

Now Serving JSON

Since we spent all that time getting our serializer to work just right, let's make a view for it and get our REST API up and running. Start by creating a separate file for our REST API views, `main/json_views.py`. This isn't required, but it's useful to keep them separate.

The distinction is not just that one set of views returns JSON and other returns HTML, but that the API views (which return JSON) are defining the REST framework for our application - e.g., how we want other clients or programs to be able to access our information. On the other hand, the "standard" views are involved with the web view of our application. We are in effect splitting our application into two distinct parts here:

1. a "core application" which exposes our resources in a RESTful way.
2. our web app which focuses on displaying web content and is also a client of our "core application".

This is a key design principle of REST, as building the application this way will allow us to more easily build other clients (say, mobile apps) without having to re-implement the back-end. *In fact, taken to its natural conclusion we could eventually have one Django app that is the "core application" that exposes the REST API to our "web app", Mos Eisley's Cantina.* We aren't quite there yet, but it's a good metaphor to keep in your head when thinking about the separation of the REST API from your web app.

And A View

Our new view function in `main/json_views.py` should look like:

```
1 from rest_framework import status
2 from rest_framework.decorators import api_view
3 from rest_framework.response import Response
4 from main.serializers import StatusReportSerializer
5 from main.models import StatusReport
6
7
8 @api_view(['GET', 'POST'])
9 def status_collection(request):
10     """Get the collection of all status_reports
11     or create a new one"""
12
13     if request.method == 'GET':
```



```

14     status_report = StatusReport.objects.all()
15     serializer = StatusReportSerializer(status_report,
16                                         many=True)
17     return Response(serializer.data)
18 elif request.method == 'POST':
19     serializer = StatusReportSerializer(data=request.DATA)
20     if serializer.is_valid():
21         serializer.save()
22         return Response(serializer.data,
23                         status=status.HTTP_201_CREATED)
24     return Response(serializer.errors,
25                     status=status.HTTP_400_BAD_REQUEST)

```

- **Line 1-5:** Import what we need
- **Line 8:** The `@api_view` is a decorator provided by DRF, which:
 - checks that the appropriate request is passed into the view function
 - adds context to the Response so we can deal with stuff like CSRF tokens
 - provides authentication functionality (which we will discuss later)
 - handles ParseErrors
- **Line 8:** The arguments to the `@api_view` are a list of the HTTP verbs to support.
- **Line 13-16:** A GET request on the collection view should return the who list. Grab the list, then serialize to JSON and return it.
- **Line 16:** DRF also provides the 'Response' object which inherits `django.template.response.SimpleTemplateResponse`. It takes in unrendered content (for example, JSON) and renders it based upon a Content-Type specified in the request header.
- **Line 17-20:** For POST requests just create a new object based upon passed-in data.
- **Line 17:** Notice the use of `request.DATA` DRF provides the `Request` class that extends Django's `HttpRequest` and provides a few enhancements: `request.DATA`, which works similar to `HttpRequest.POST` but handles POST, PUT and PATCH methods.
- **Line 21:** On successfully saving, return a response with HTTP return code of 201 (created). Notice the use of `status.HTTP_201_CREATED`. You could simply put in 201,

but using the DRF status identifiers makes it more explicit as to what code you're returning so that people reading your code don't have to remember all the HTTP return codes.

- **Line 22:** If the deserialization process didn't work (i.e., `serializer.is_valid()` returns `False`) then return `HTTP_400_BAD_REQUEST`. This basically means don't call me again with the same data because it doesn't work.

That's a lot of functionality and not very much code. Also if you recall from the section on *Structuring a REST API*, this produces a REST API that uses the correct HTTP Verbs and returns the appropriate response codes. If you further recall from the discussion on Structuring a REST API, resources have a collection URL and a member URL. To finish the example we need to flesh out the member URL below by updating *main/json_views.py*:

```
1 @api_view(['GET', 'PUT', 'DELETE'])
2 def status_member(request, id):
3     """Get, update or delete a status_report instance"""
4
5     try:
6         status_report = StatusReport.objects.get(id=id)
7     except StatusReport.DoesNotExist:
8         return Response(status=status.HTTP_404_NOT_FOUND)
9
10    if request.method == 'GET':
11        serializer = StatusReportSerializer(status_report)
12        return Response(serializer.data)
13    elif request.method == 'PUT':
14        serializer = StatusReportSerializer(status_report,
15                                           data=request.DATA)
16        if serializer.is_valid():
17            serializer.save()
18            return Response(serializer.data)
19        return Response(serializer.errors,
20                        status=status.HTTP_400_BAD_REQUEST)
21    elif request.method == 'DELETE':
22        status_report.delete()
23        return Response(status=status.HTTP_204_NO_CONTENT)
```

This is nearly the same as the collection class, but we support different HTTP verbs and are dealing with one object instead of an entire collection of objects. With that, we now have the entire API for our `StatusReport`

NOTE: In the code above, the PUT request is not idempotent. Do you know why? What happens if we call a PUT request with an id that is not in the database? For extra credit go ahead and implement the fix now ... or just read on; we will fix it later.

Now were getting somewhere.

Let's not forget to test it.

Test

Create a new file called `../tests/main/testJSONViews.py`:

First the GET functionality:

```
1 from django.test import TestCase
2 from main.json_views import status_collection
3 from main.models import StatusReport
4 from main.serializers import StatusReportSerializer
5
6
7 class dummyRequest(object):
8
9     def __init__(self, method):
10         self.method = method
11         self.encoding = 'utf8'
12         self.user = "root"
13         self.QUERY_PARAMS = {}
14         self.META = {}
15
16
17 class JsonViewTests(TestCase):
18
19     def test_get_collection(self):
20         status = StatusReport.objects.all()
21         expected_json = StatusReportSerializer(status,
22             many=True).data
23         response = status_collection(dummyRequest('GET'))
24
25         self.assertEqual(expected_json, response.data)
```

Above we create a dummyRequest that has the information that DRF expects.

NOTE: We can't use the `RequestFactory` yet because we haven't setup the URLs.

Then in our `JsonViewTests` we call our `status_collection` function, passing in the GET parameter.

This should return all the `StatusReport` objects as JSON. We manually query all the `StatusReport`, convert them to JSON, and then compare that to the return from our view call. Notice the returned response we call `response.data` as opposed to `response.content` which we are used to, because this response hasn't actually been rendered yet.

Otherwise the test is the same as any other view test. To be complete, we should check the case where we have data and where there is no data to return as well, and we should also test the POST with and without valid data. We'll leave that as an exercise for you, dear reader.

Don't forget to run the test:

```
1 $ ./manage.py test ../tests
```

Now that we have tested our view, let's go ahead and wire up the URLs. We are going to create a separate `urls` file specifically for the JSON URLs in our main application as opposed to using our default `django_ecommerce/urls.py` file. This creates better separation of concerns and allows our REST API to be more "independent".

Let's create a `main/urls.py` file that contains:

```
1 from django.conf.urls import patterns, url
2
3 urlpatterns = patterns(
4     'main.json_views',
5     url(r'^status_reports/$', 'status_collection'),
6     url(r'^status_reports/(?P<id>[0-9]+)$', 'status_member'),
7 )
```

We need our `django_ecommerce/urls.py` to point to this new `urls.py`. So add the following URL entry to the end of the list:

```
1 url(r'^api/v1/', include('main.urls')),
```

Don't forget to actually add `rest_framework` to the list of `INSTALLED_APPS` in your `settings.py`. This should now look like:

```
1 INSTALLED_APPS = (
2     'django.contrib.auth',
```

```

3     'django.contrib.contenttypes',
4     'django.contrib.sessions',
5     'django.contrib.sites',
6     'django.contrib.messages',
7     'django.contrib.staticfiles',
8     'main',
9     'django.contrib.admin',
10    'django.contrib.flatpages',
11    'contact',
12    'payments',
13    'embed_video',
14    'rest_framework',
15 )

```

Now if you start the development server and navigate to http://127.0.0.1:8000/api/v1/status_reports/ you should see something like:

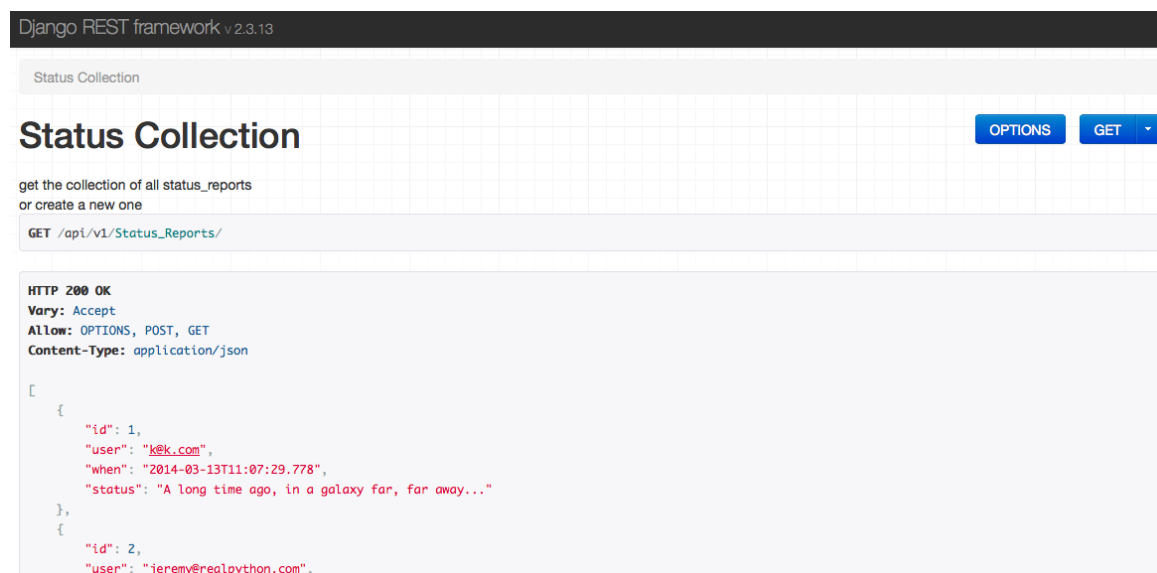


Figure 11.1: DRF browsable API for Status Collection

Wow! That's DRF's "Browsable API" and it's a real usability win. With no extra work you automatically get this nice browsable API, which shows you, in human-readable format, what function is being called and its return value.

Also if you scroll down on the page a little bit, you will see that it gives you a way to easily call the API. Perfect for a quick manual test/sanity check to make sure things are working correctly. (But obviously not a replacement for unit testing, so don't get any ideas.)

Of course, when you call your API from your program you don't want to see that page; we just want the JSON. Don't worry: DRF has you covered. By default the `@api_view` wrapper, which gives us the cool browsable API amongst other things, listens to the `Accept` header to determine how to render the template (remember the `rest_framework.response.Response` is just a `TemplateResponse` object).

Try this from the command line (with the development server running):

```
$ curl http://127.0.0.1:8000/api/v1/status_reports/
```

NOTE: Windows Users Sorry. You most likely don't have curl installed by default like the rest of us do. You can download it [here](#). Just scroll ALL the way down to the bottom and select the appropriate download for you system.

Or, to be more explicit:

```
$ curl http://127.0.0.1:8000/api/v1/status_reports/ -H 'Accept: application/json'
```

And you will get back raw JSON (as long as you have a status update in the table, of course).

```
[{"id": 1, "user": "test@testusr.com", "when": "2014-08-22T01:35:03.965Z", "status": "test"}]
```

Thus returning JSON is the default action the DRF Response will take. However, by default your browser will set the `Accept` header to `text/html`, which you can also do from curl like this:

```
$ curl http://127.0.0.1:8000/api/v1/status_reports -H 'Accept: text/html'
```

And then you'll get back a whole mess of HTML. Hats off to the DRF folks. Very nicely done.

Using Class-Based Views

Up until now we have been using function-based views. Functions are cool and easy, but there are some things that become easier if you use a class-based view - mainly, reusing functionality. DRF provides a number of mixins that can be used with your class-based views to make your life easier.

Let's refactor our function-based views to class-based views.

By using some of the mixins that DRF provides, we can do more with *a lot* less code. Update `json_views.py`:

```
1 from rest_framework import mixins, generics
2 from main.serializers import StatusReportSerializer
3 from main.models import StatusReport
4
5
6 class StatusCollection(mixins.ListModelMixin,
7                       mixins.CreateModelMixin,
8                       generics.GenericAPIView):
9
10     queryset = StatusReport.objects.all()
11     serializer_class = StatusReportSerializer
12
13     def get(self, request):
14         return self.list(request)
15
16     def post(self, request):
17         return self.create(request)
```

Where did all the code go? That's exactly what the mixins are for:

- **Line 7:** `mixins.ListModelMixin` provides the `list(request)` function that allows you to serialize a collection to JSON and return it.
- **Line 7:** `mixins.CreateModelMixin` provides the `create(request)` function that allows for the POST method call - e.g., creating a new object of the collection type.
- **Line 7:** `generics.GenericAPIView` - this mixin provides the “core” functionality plus the Browsable API we talked about in the previous section.
- **Line 10:** defining a class-level `queryset` member is required so the `ListModelMixin` can work its magic.

- **Line 11:** defining a class-level `serializer_class` member is also required for all the Mixins to work.
- **Remaining Lines:** we implement GET and POST by passing the call to the respective Mixin.

Using the class-based view in this way with the DRF mixins saves a lot of boilerplate code while still keeping things pretty easy to understand. Also, we can clearly see what happens with a GET vs a POST request without having a number of if statements, so there is better separation of concerns.

NOTE: It would help even more if the mixin would have been called `generics.GetPostCollectionAPIView` so that you know it's for GET and POST on a collection as opposed to having to learn DRF. `ListCreateAPIView` doesn't really tell us anything about the REST API that this view function is creating unless we are already familiar with DRF. In general, the folks at Real Python like to be a bit more explicit even if it means just a bit more code. Fortunately, there is nothing preventing you from putting in a nice docstring to explain what the function does - which is a good compromise. Ultimately it's up to you to decide which one you prefer.

To complete the example, here is the `status_member` function after being refactored into a class view:

```

1 class StatusMember(mixins.RetrieveModelMixin,
2                     mixins.UpdateModelMixin,
3                     mixins.DestroyModelMixin,
4                     generics.GenericAPIView):
5
6     queryset = StatusReport.objects.all()
7     serializer_class = StatusReportSerializer
8
9     def get(self, request, *args, **kwargs):
10         return self.retrieve(request, *args, **kwargs)
11
12     def put(self, request, *args, **kwargs):
13         return self.update(request, *args, **kwargs)
14
15     def delete(self, request, *args, **kwargs):
16         return self.destroy(request, *args, **kwargs)

```


In fact we can even simplify the class inheritance further using:

```
1 class StatusMember(mixins.RetrieveUpdateDestroyAPIView):
```

This is just a combination of the four mixins we inherited from above. The choice is yours.

We also need to change our *main/urls.py* function slightly to account for the class-based views:

```
1 from django.conf.urls import patterns, url
2 from main import json_views
3
4 urlpatterns = patterns(
5     'main.json_views',
6     url(r'^status_reports/$',
7         json_views.StatusCollection.as_view()),
8     url(r'^status_reports/(?P<pk>[0-9]+)/$',
9         json_views.StatusMember.as_view())
10 )
```

There are two things to take note of here:

1. We are using the class method `as_view()` that provides a function-like interface into the class.
2. The second URL for the `StatusMember` class must use the `pk` variable (before we were using `id`), as this is a requirement of DRF.

And finally, we need to modify our test a little bit to account for the class-based views:

```
1 from main.json_views import StatusCollection
2
3 def test_get_collection(self):
4     status = StatusReport.objects.all()
5     expected_json = StatusReportSerializer(status, many=True).data
6     response = StatusCollection.as_view()(dummyRequest("GET"))
7
8     self.assertEqual(expected_json, response.data)
```

Notice in line 5 that we need to call the `as_view()` function of our `StatusCollection` class just like we do in *main/urls.py*. We can't just call `StatusCollection().get(dummyRequest("GET"))` directly. Why? Because `as_view()` is magic. It sets up several instance variables such as `self.request`, `self.args`, and `self.kwargs`; without these member variables set up, your test will fail.

Make sure to run your tests before moving on.

Authentication

There is one final topic that needs to be covered so that a complete REST API can be implemented: [Authentication](#).

Since we are charging a membership fee for MEC, we don't want unpaid users to have access to our members-only data. In this section we will look at how to use authentication so that only authorized users can access your REST API. In particular, we want to enforce the following constraints:

- Unauthenticated requests should be denied access
- Only authenticated users can post status reports
- Only the creator of a status report can update and/or delete that status report

Unauthenticated requests should be denied access

We can use DRF's built-in permissions - `rest_framework.permissions.IsAuthenticated` - to take care of this. All we need to do is add that class as a property of our class views (`StatusCollection` and `StatusMember`) by adding the property like so:

```
1 permission_classes = (permissions.IsAuthenticated,)
```

Pay attention to that comma at the end of the line! We need to pass a tuple, not a single item. Also, don't forget the proper import:

```
1 from rest_framework import permissions
```

Testing Authentication

We can verify that this is working by running our unit tests which should now fail, as they try to check if the user is authenticated:

```
1 AttributeError: 'str' object has no attribute 'is_authenticated'
```

We need an authenticated user. Again, DRF comes through with some helpful test functions. Previously we were using a `dummyRequest` class to provide the functionality that we needed. Let's drop that and use DRF's `APIRequestFactory`:

```
1 from django.test import TestCase
2 from rest_framework.test import APIRequestFactory,
  force_authenticate
```

```

3 from payments.models import User
4
5 class JsonViewTests(TestCase):
6
7     @classmethod
8     def setUpClass(cls):
9         super().setUpClass()
10        cls.factory = APIRequestFactory()
11
12    @classmethod
13    def setUpTestData(cls):
14        cls.test_user = User(id=2222, email="test@user.com")
15
16    def get_request(self, method='GET', authed=True):
17        request_method = getattr(self.factory, method.lower())
18        request = request_method("")
19        if authed:
20            force_authenticate(request, self.test_user)
21
22    return request

```

OK so what did we do in the above section of code?

- **Line 1:** - import all the DRF testing goodies that we need.
- **Lines 7-10:** - using the setUpClass function that will only run once for the test suite, create the APIRequestFactory that we will use in our tests.
- **Lines 12-14** - create a dummy user that we will use to authenticate. Note: we do not need to actually save the user to the database, we just need a user object.
- **Line 16:** - get_request is a factory that will create a request using the HTTP verb passed in as 'method' (the default value of which is 'GET').
- **Line 17:** - the actual factory pattern that will get a method such as APIRequestFactory.get or APIRequestFactory.post depending on the value we pass in for the method parameter.
- **Line 18:** - create the request that we got from the factory in line 11 - e.g., create a get or post or delete or whatever request.
- **Lines 19-20:** - if we passed in a truthy value for the method parameter 'auth' ensure the request is authenticated with our self.test_user otherwise the request will be unauthenticated.
- **Line 22:** - return it and we are done.

In a nutshell we have coded a way to create a request for any type of HTTP verb. Further, we can decide if that request is authenticated or not. This gives us the flexibility to test as many permutations as we might need to so we can properly exercise our REST API. For now we put the code in the `JSONViewTests` because that's all we need. However you might consider creating your own `DRFTestCase`, perhaps putting it in `../test/base_test_case.py`, for example. Then you could easily share it amongst whatever tests you create that need the functionality.

Using Our New Request Factory

Now that we have the new fangled `get_request` factory function lets update our `test_get_collection` so it will pass.

```
1 def test_get_collection(self):
2     status = StatusReport.objects.all()
3     expected_json = StatusReportSerializer(status, many=True).data
4
5     response = StatusCollection.as_view()(self.get_request())
6     self.assertEqual(expected_json, response.data)
```

Line 5 is the only line that changed, as it now calls our newly created `self.get_request()` factory method.

Let's add one more test, `test_get_collection_requires_logged_in_user`, to verify that our authentication is working correctly:

```
1 from rest_framework import status
2
3 def test_get_collection_requires_logged_in_user(self):
4
5     anon_request = self.get_request(method='GET', authenticated=False)
6     response = StatusCollection.as_view()(anon_request)
7
8     self.assertEqual(response.status_code,
9                       status.HTTP_403_FORBIDDEN)
```

- **Line 1:** we use DRF's statuses in our test as they are more descriptive, so we will need to add this import line to the top of the module
- **Line 5:** we pass in `authenticated=False` to our `get_request` factory method, which sets the user to be unauthorized

- **Line 8:** we verify that we return `status.HTTP_403_FORBIDDEN`, which is the correct HTTP status code to return in the case of unauthorized access

Now we have our permission setup to ensure unauthorized users get a good ole [Heisman](#).

This also takes care of our first authorization requirement: *Only authenticated users can post status reports*. Now for our final requirement: *Only the creator of a status report can update or delete that status report*.

To implement this type of permission, we need to create our own custom permission class. Create a `main/permissions.py` file and then add the following code:

```
1 from rest_framework import permissions
2
3
4 class IsOwnerOrReadOnly(permissions.BasePermission):
5
6     def has_object_permission(self, request, view, obj):
7
8         #Allow all read type requests
9         if request.method in ('GET', 'HEAD', 'OPTIONS'):
10             return True
11
12         #this leaves us with write requests (i.e. POST / PUT /
13         DELETE)
14         return obj.user == request.user
```

Once the class is created, we need to update our `StatusMember` class in `json_views.py` as well:

```
1 from main.permissions import IsOwnerOrReadOnly
2
3 class StatusMember(mixins.RetrieveModelMixin,
4                   mixins.UpdateModelMixin,
5                   mixins.DestroyModelMixin,
6                   generics.GenericAPIView):
7
8     queryset = StatusReport.objects.all()
9     serializer_class = StatusReportSerializer
10    permission_classes = (permissions.IsAuthenticated,
11                        IsOwnerOrReadOnly)
```

```
12     def get(self, request, *args, **kwargs):
13         return self.retrieve(request, *args, **kwargs)
14
15     def put(self, request, *args, **kwargs):
16         return self.update(request, *args, **kwargs)
17
18     def delete(self, request, *args, **kwargs):
19         return self.destroy(request, *args, **kwargs)
```

And that's it. Now only owners can update / delete their status reports.

It is important to note that all of this authentication is using the default Authentication classes which are `SessionAuthentication` and `BasicAuthentication`. If your main client is going to be an AJAX web-based application then the default authentication classes will work fine, but DRF does provide several other authentication classes if you need something like `OAuthAuthentication`, `TokenAuthentication` or something custom.

The official DRF [documentation](#) does a pretty good job of going over these if you want more info.

Conclusion

We started this chapter talking about the desire to not refresh the page when a user submitted a status report. And, well, we didn't even get to that solution yet. Think of it as a cliff hanger... to be continued in the next chapter.

We did however go over one of the key ingredients to making that no-refresh happen: a good REST API. REST is increasingly popular because of its simplicity to consume resources and because it can be accessed from any client that can access the web.

Django REST Framework makes implementing the REST API relatively straight-forward and helps to ensure that we follow good conventions. We learned how to serialize and deserialize our data and structure our views appropriately along with the browsable API and some of the important features of DRF. There are even more features in DRF that are worth exploring such as `ViewSet` and `Routers`. While these powerful classes can greatly reduce the code you have to write, you sacrifice readability. But that doesn't mean you shouldn't [check them out](#) and use them if you like.

In fact, it's worth going through the [DRF site](#) and browsing through the API Guide. We've covered the most common uses when creating REST APIs, but with all the different use-cases out there, some readers will surely need some other part of the framework that isn't covered here.

Either way: **REST is everywhere on the web today. If you're going to do much web development, you will surely have to work with REST APIs - so make sure you understand the concepts presented in this chapter.**

Exercises

1. Flesh out the unit tests. In the `JsonViewTests`, check the case where there is no data to return at all, and test a POST request with and without valid data.
2. Extend the REST API to cover the `user.models.Badge`.
3. Did you know that the browsable API uses Bootstrap for the look and feel? Since we just learned Bootstrap, update the browsable API Template to fit with our overall site template.
4. We don't have permissions on the browsable API. Add them in.

Chapter 12

Django Migrations

What's new in Django 1.7? Basically migrations. While there are some other nice features, the new migrations system is the big one.

In the past you probably used [South](#) to handle database changes. However, in Django 1.7, migrations are now integrated into the Django Core thanks to Andrew Godwin, who ran [this Kickstarter](#). He is also the original creator of South.

Let's begin...

The problems that Migrations Solve

Migrations:

1. Speed up the notoriously slow process of changing a database schema.
2. Make it easy to use git to track your database schema and its associated changes. Databases simply aren't aware of git or other version control systems. Git is awesome for code, but not so much for database schemas.
3. Provide an easy way to maintain fixture data linked to the appropriate schema.
4. Keep the code and schema in sync.

Have you ever had to make a change to an existing table (like re-naming a field) and you didn't want to mess with dropping the table and re-adding it?

Migrations solve that.

Or perhaps you needed to update the schema on a live application with millions of rows of data that simply cannot be lost.

Migrations make this much easier.

In general, migrations allow you to manage and work with your database schema in the same way you would with your Django code. You can store versions of it in git, you can update it from the command line, and you don't have to worry about creating large complex SQL queries to keep everything up to date - although you still can if you love pain... I mean SQL.

Getting Started with Migrations

The basic process for using migrations is simple:

1. Create and/or update a model.
2. Run `./manage.py makemigrations <app_name>`.
3. Run `./manage.py migrate` to migrate everything or `./manage.py migrate <app_name>` to migrate an individual app.
4. Repeat as necessary.

That's it! Pretty straight-forward for the basic use-case, and this will work the *majority* of the time. But when it doesn't work it can be hard to figure out why, so let's dig a bit deeper to get a better understanding of how migrations work.

The Migration File

Start by running the following migration

```
1 $ ./manage.py makemigrations contact
```

You should see something like:

```
1 Migrations for 'contact':
2   0001_initial.py:
3     - Create model ContactForm
```

What exactly happened?

Django migrations actually created a migration file that describes how to create (or update) the appropriate tables in the database. In fact, you can look at the migration file that was created. Don't be afraid: It's just python :). The file is located within the “migration” directory in the *contact* app. Open `contact/migrations/0001_initial.py`.

It should look something like this:

```
1 # -*- coding: utf-8 -*-
2 from __future__ import unicode_literals
3
4 from django.db import models, migrations
5 import datetime
6
7
8 class Migration(migrations.Migration):
9
10     dependencies = [
11     ]
12
13     operations = [
14         migrations.CreateModel(
15             name='ContactForm',
16             fields=[
17                 ('id', models.AutoField(
18                     verbose_name='ID', primary_key=True,
19                     serialize=False, auto_created=True)),
20                 ('name', models.CharField(max_length=150)),
21                 ('email', models.EmailField(max_length=250)),
22                 ('topic', models.CharField(max_length=200)),
```

```

23         ('message', models.CharField(max_length=1000)),
24         ('timestamp',
           models.DateTimeField(auto_now_add=True)),
25     ],
26     options={
27         'ordering': ['-timestamp'],
28     },
29     bases=(models.Model,),
30 ),
31 ]

```

For a migration to work, the migration file must contain a class called `Migration()` that inherits from `django.db.migrations.Migration`. This is the class that the migration framework looks for and executes when you ask it to run migrations - which we will do later.

The `Migration()` class contains two main lists, dependencies and operations.

Migration dependencies

Dependencies is a list of migrations that must be run prior to the current migration being run. In the case above, nothing has to run prior so there are no dependencies. But if you have foreign key relationships then you will have to ensure a model is created before you can add a foreign key to it.

To see that let's create migrations for our main app:

```

1 $ ./manage.py makemigrations main

```

You should see something like this:

```

1 Migrations for 'main':
2   0001_initial.py:
3     - Create model Announcement
4     - Create model Badge
5     - Create model MarketingItem
6     - Create model StatusReport

```

After running that have a look at `main/migrations/0001_initial.py`. In the dependency list you will see:

```

1 dependencies = [
2     ('payments', '__first__'),
3 ]

```

The dependency above says that the migrations for the payments app must be run before the current migration. You might be wondering, “How does Django know I have a dependency on ‘payments’ when I only ran makemigrations for main?”:

- Short Answer: Magic.
- Slightly Longer Answer: makmigrations look at things like ForeignKey Fields to determine dependencies (more on this later).

You can also have a dependency on a specific file.

To see an example, let’s initialize another migration:

```
1 $ ./manage.py makemigrations payments
```

You should see something like:

```
1 Migrations for 'payments':  
2   0001_initial.py:  
3     - Create model UnpaidUsers  
4     - Create model User
```

Check out the dependency in the payments/migrations/0001_initial.py migration file.

```
1 dependencies = [  
2     ('main', '0001_initial'),  
3 ]
```

This essentially means that it depends on the *0001_initial.py* file in the main app to run first. This functionality provides a lot of flexibility, as you can accommodate foreign keys that depend upon models from different apps.

Dependencies can also be combined (it’s just a list after all) so you can have multiple dependencies - which means that the numbering of the migrations (usually 0001, 0002, 0003, ...) doesn’t strictly have to be in the order they are applied. You can add any dependency you want and thus control the order without having to re-number all the migrations.

Migration operations

The second list in the Migration() class is the operations list. This is a list of operations to be applied as part of the migration. Generally the operations fall under one of the following types:

- **CreateModel:** You guessed it: This creates a new model. See the migration above for an example.
- **DeleteModel:** removes a table from the database; just pass in the name of the model.
- **RenameModel:** Given the `old_name` and `new_name`, this renames the model.
- **AlterModelTable:** changes the name of the table associated with a model. Same as the `db_table` option.
- **AlterUniqueTogether:** changes unique constraints.
- **AlteIndexTogether:** changes the set of custom indexes for the model.
- **AddField:** Just like it sounds. Here is an example (and a preview of things to come... dun dun dun dun):

```

1 migrations.AddField(
2     model_name='user',
3     name='badges',
4     field=models.ManyToManyField(to='main.Badge')
5 ),

```

- **RemoveField:** We don't want that field anymore... so just drop it.
- **RenameField:** Given a `model_name`, an `old_name` and a `new_name`, this changes the field with `old_name` to `new_name`.

There are also a few “special” operations:

- **RunSQL:** This allows you to pass in raw SQL and execute it as part of your model.
- **RunPython:** passes in a callable to be executed; useful for things like data loading as part of the migration.

You can even write your own operations. Generally when you run `makemigrations`, Django will create the necessary migrations with the appropriate dependencies and operations that you need. However, understanding the migration files themselves and how they work give you more flexibility.

When Migrations Don't Work

One of the most frequent causes for migrations not working correctly is a circular dependency error.

Let's now try applying the migrations:

```
1 $ ./manage.py migrate
```

You should see the follow error:

```
1 raise CircularDependencyError(path[path.index(start):] + [start])
2 django.db.migrations.graph.CircularDependencyError: [('main',
  '0001_initial'), ('payments', '0001_initial'), ('main',
  '0001_initial')]
```

And, yes: This is a circular dependency error.

Let's look at what's happening.

Models

Our `payments.models.User` looks like this:

```
1 class User(AbstractBaseUser):
2     name = models.CharField(max_length=255)
3     email = models.CharField(max_length=255, unique=True)
4     #password field defined in base class
5     last_4_digits = models.CharField(max_length=4, blank=True,
        null=True)
6     stripe_id = models.CharField(max_length=255)
7     created_at = models.DateTimeField(auto_now_add=True)
8     updated_at = models.DateTimeField(auto_now=True)
9     rank = models.CharField(max_length=50, default="Padwan")
10    badges = models.ManyToManyField(Badge)
```

Notice the `ManyToManyField` called `badges` at the end, which references `main.models.Badge`:

```
1 class Badge(models.Model):
2     img = models.CharField(max_length=255)
3     name = models.CharField(max_length=100)
4     desc = models.TextField()
```

Okay. So far there are no issues, but we have another model to deal with:


```

1 class StatusReport(models.Model):
2     user = models.ForeignKey('payments.User')
3     when = models.DateTimeField(blank=True)
4     status = models.CharField(max_length=200)

```

Oops! We now have `payments.models` depending on `main.models` and `main.models` depending on `payments.models`. That's a problem. In the code, we solved this already by not importing `payments.models` and instead using the line:

```

1 user = models.ForeignKey('payments.User')

```

While that trick works at the application-level, it doesn't work when we try to apply migrations to the database.

Migrations

How about the migration files? Again, take a look at the dependencies:

`main.migrations.001_initial:`

```

1 dependencies = [
2     ('payments', '__first__'),
3 ]

```

`payments.migrations.001_initial:`

```

1 dependencies = [
2     ('main', '0001_initial'),
3 ]

```

So, the latter migration depends upon the main migration running first, and thus - we have a circular reference. Remember how we talked about `makemigrations` looking at `ForeignKey` fields to create dependencies? That's exactly what happened to us here.

The fix

When I was an intern in college (my first real development job) my dev manager said something to me which I will never forget: *You can't really understand code unless you can write it yourself.*

This was after a copy and paste job I did crashed the system.

So let's write a migration from scratch.

We are going to remove the `StatusReport` model from `main/migrations/0001_initial.py` and add it to a new migration.

Create a new file called `main/migrations/0002_statusreport.py`:

```
1 # -*- coding: utf-8 -*-
2 from __future__ import unicode_literals
3
4 from django.db import models, migrations
5
6
7 class Migration(migrations.Migration):
8
9     dependencies = [
10         ('payments', '__first__'),
11         ('main', '0001_initial'),
12     ]
13
14     operations = [
15         migrations.CreateModel(
16             name='StatusReport',
17             fields=[
18                 ('id', models.AutoField(
19                     primary_key=True, auto_created=True,
20                     verbose_name='ID', serialize=False)),
21                 ('when', models.DateTimeField(blank=True)),
22                 ('status', models.CharField(max_length=200)),
23                 ('user', models.ForeignKey(to='payments.User')),
24             ],
25             options={},
26             bases=(models.Model,)),
27     ],
28
29 ]
```

Notice how this migration depends upon both `main.0001_initial` and `payments.__first__`. This means that the `payments.user` model will already be created before this migration runs and thus the user foreign key will be created successfully.

Don't forget to:

1. Modify `main/migrations/0001_initial.py` to remove the `payments` dependency

- on payments (so it has no dependencies):
2. Remove the `CreateModel()` for `StatusReports`.

And now we have resolved the circular dependency by splitting up the migration into two parts. This is a common pattern to use when fixing circular dependencies, so keep it in mind if you run into similar issues.

Go ahead and run the migrations now to ensure everything is working correctly:

```
$ ./manage.py migrate
```

You should now have your database in sync with your migration files!

Timezone support

If you have timezone support enabled in your `settings.py` file, you will likely get an error, when running migrations that says something to the effect of: native datetime while time zone support is active. The fix is to use `django.utils.timezone` instead of `datetime`. You'll have to update `contact.models.ContactForm` and `payments.models.UnpaidUsers`. Just replace `datetime.datetime` with `django.utils.timezone`. You should update both the models and the migrations files.

More info on timezone support can be found in the Django [docs](#).

Warning Message on Migrate

There is another datetime related issue that you may run into with migrations. You may see the following warning message when you run migrate:

```
Your models have changes that are not yet reflected in a migration,
and so won't be applied.
```

This has to do with the same datetime fields we just discussed. Since we are defaulting the value of the datetime field to `now()` Django will see the default value as always being different and thus it will give you the warning above. While this is supposed to be a helpful warning message to remind you to keep your migrations up-to-date with your `models.py` in this case it is a minor annoyance. There is nothing to be done here, just ignore the warning.

Data Migrations

Migrations are mainly for keeping the data model of your database up-to-date, but a database is more than just a data model. Most notably, it's also a large collection of data. So any discussion of database migrations wouldn't be complete without also talking about data migrations.

Data migrations are used in a number of scenarios. Two very popular ones are:

1. **Loading system data:** When you would like to load “system data” that your application depends upon being present to operate successfully.
2. **Migrating existing data:** When a change to a data model forces the need to change the existing data.

Do note that loading dummy data for testing is not in the above list. You could use migrations to do that, but migrations are often run on production servers, so you probably don't want to be creating a bunch of dummy test data on your production server. (More on this later)

Let's look at examples of each...

Loading System Data

As an example of creating some “system data”, let's define a system user that always needs to exist. In `payments/migrations`, add a new file called `0003_initial_data.py`.

Start by adding the following code:

```
1  # -*- coding: utf-8 -*-
2  from __future__ import unicode_literals
3
4  from django.db import migrations
5
6
7  class Migration(migrations.Migration):
8
9      dependencies = [
10         ('payments', '0001_initial'),
11     ]
12
13     operations = [
```

```

14         migrations.RunPython(create_default_user)
15     ]

```

Like any other migration, we create a class called `Migration()`, set its dependency, and then define the operation.

For Data Migrations we can use the `RunPython()` operation, which just accepts a callable and calls that function. So, we need to add in the `create_default_user()` function:

```

1  # -*- coding: utf-8 -*-
2  from __future__ import unicode_literals
3  from django.db import migrations
4  from payments.models import User
5  from django.contrib.auth.hashers import make_password
6
7
8  def create_default_user(apps, schema_editor):
9      new_user = apps.get_model("payments", "User")
10     try:
11         vader = new_user.objects.get(email="darth@mec.com")
12         vader.delete()
13     except new_user.DoesNotExist:
14         pass
15
16     u = new_user(
17         name='vader', email="darth@mec.com",
18         last_4_digits="1234", password= make_password("darkside")
19     ).save()
20
21
22 class Migration(migrations.Migration):
23
24     dependencies = [
25         ('payments', '0001_initial'),
26     ]
27
28     operations = [
29         migrations.RunPython(create_default_user)
30     ]

```

This function just adds the new user. A couple of things are worth noting:

1. We don't use the User object but rather grab the model from the App Repository. By doing this, migrations will return us the historic version of the model. This is important as the fields in the model may have changed. Grabbing the model from the App Repository will ensure we get the correct version of the model.
2. Since we are grabbing from a historic app repository it is likely we don't have access to our custom defined functions such as `user.create`. So we saved the user without using the `user.create` function.
3. There are cases where you may want to rerun all your migrations or perhaps there is existing data in the database before you run migrations, so we've added a check to clear out any conflicting data before we create the new user:

```
1 try:
2     vader = new_user.objects.get(email="darth@mec.com")
3     vader.delete()
4 except new_user.DoesNotExist:
5     pass
```

This will prevent the annoying duplicate primary key error that you would get if you somehow ran this migration twice.

Go ahead and apply the migration:

```
1 $ ./manage.py migrate
```

You should see something like:

```
1 Operations to perform:
2   Synchronize unmigrated apps: contact, rest_framework
3   Apply all migrations: admin, sessions, payments, sites,
      flatpages, contenttypes, auth, main
4 Synchronizing apps without migrations:
5   Creating tables...
6   Installing custom SQL...
7   Installing indexes...
8 Running migrations:
9   Applying payments.0003_initial_data... OK
```

So we have two main use cases for loading data:

1. To load system data, that is data that needs to be in the database for the application to work correctly.
2. To load data that is necessary / helpful for our tests to run.

While both use cases can be accomplished with migrations the second use case, loading test data, should be thought of as something separate from migrations. You can continue to use fixtures for loading test data, or better yet just create the data you need in the testcase itself.

Aside: How do migrations know what to migrate?

Let's digress for a bit and look at what happens when you run migrations multiple times. The standard behavior is that Django will never run a migration more than once on the same database. This is managed by a table called `django_migrations` that is created in your database the first time you apply migrations. For each migration that is ran or faked, a new row is inserted into the table.

Let's look at our table. Open the Postgres Shell, connect to the database, and then run:

```
1 SELECT * FROM django_migrations;
```

You should see something similar to:

1	id	app	name	date applied
2				
3	1	main	0001_initial	2014-09-20 23:51:38.499414-05
4	2	payments	0001_initial	2014-09-20 23:51:38.600185-05
5	4	main	0002_statusreport	2014-09-20 23:52:33.808006-05
6	5	payments	0003_initial_data	2014-09-21 11:36:12.702975-05

The next time migrations are run, it will skip the migration files listed in the database here. This means that even if you change the migration file manually, it will be skipped if there is an entry for it in the database. This makes sense as you generally don't want to run migrations twice. But if for whatever reason you do, one way to get it to run again is to first delete the corresponding row from the database. In the case of schema migrations, though, Django will first check the database structure, and if it is the same as the migration (i.e. the migration doesn't apply any new changes) then the migration will be "faked" meaning not really ran.

Conversely, if you want to "undo" all the migrations for a particular app, you can migrate to a special migration called zero. For example if you type:

```
1 $ ./manage.py migrate payments zero
```

It will undo (reverse) all the migrations for the `payments` app. You don't have to use zero; you can use any arbitrary migration (like `./manage.py migrate payments 0001_initial`), and if that migration is in the past then the database will be rolled back to the state of that migration, or rolled forward if the migration hasn't yet been run. Pretty powerful stuff!

Note: This doesn't apply to data migrations.

Migrating existing data

Coming back to data migrations, the other reason why you might use data migrations is when you actually need to migrate the data - e.g., change how the data is stored.

The [Django docs](#) have a good example of this called `combine_names`, which uses a data migration to combine the first and last name into one column, `name`. Most likely this migration would come just after the migration that created the new name column, but before the migration that deleted the `first_name` and `last_name` columns. The migration is called the same way as in our previous example using `create_default_user`.

Let's look at the actual `combine_names` function that is demonstrated in the documentation:

```
1 # -*- coding: utf-8 -*-
2 from django.db import models, migrations
3
4 def combine_names(apps, schema_editor):
5     # We can't import the Person model directly as it may be a newer
6     # version than this migration expects. We use the historical
7     # version.
8     Person = apps.get_model("yourappname", "Person")
9     for person in Person.objects.all():
10         person.name = "%s %s" % (person.first_name,
11                                 person.last_name)
12         person.save()
13
14 class Migration(migrations.Migration):
15
16     dependencies = [
17         ('yourappname', '0001_initial'),
18     ]
19
20     operations = [
21         migrations.RunPython(combine_names),
22     ]
```

When you create a Python function to be called by the `RunPython` migration, it must accept two arguments. The first is `apps`, which is of type `django.apps.registry.Apps` and gives you access to the historical models/migrations. In other words, this is a model that has the state as defined in the previous migration (which could be vastly different to the current state of the model). By state we are mainly referring to the fields associated with the model.

The second argument is the `schema_editor` for changing the schema, which should not be

necessary very often when migrating data, because you're not changing the schema, just the data.

In the example we call `apps.get_model`, which gives us that historical model. Then we loop through all rows in the model and combine the `first_name` and `last_name` into a single name and save the row. That's it. our migration is done. It's actually pretty straight-forward to write the migration once you get the hang of it.

The hardest part is remembering the structure of a migrations file, but Django Migrations has got that covered!. From the command line, if you run-

```
1 $ ./manage.py makemigrations --empty yourappname
```

-this will create an empty migration file in the appropriate app. Django will also suggest a name for the migration (which you are free to change), and it will add your dependencies automatically, so you can just start writing your operations.

Migrations and Fixtures

If you recall from earlier chapters we created a few fixtures to load some initial data:

- `main/fixtures/initial_data.json`
- `payments/fixtures/initial_data.json`

If you need to load system data you should load it in a migration. This is probably a good use case for the `MarketingItems` data (from the exercises at the end of the Bootstrap chapter) because we want that to display our web page.

That said, do not use any fixtures named `initial_data.json`, as it will cause problems. So, let's rename the fixtures to:

- `main/fixtures/system_data.json`
- `payments/fixtures/system_data.json`

To load them you need to use the django command `loaddata`:

```
1 $ ./manage.py loaddata main/fixtures/system_data.json
2 $ ./manage.py loaddata payments/fixtures/system_data.json
```

Or:

```
1 $ ./manage.py loaddata dummy_data
```

This will load the data into your database, so you can call it as needed.

Conclusion

We've covered the most common scenarios you'll encounter when using migrations. There are plenty more, and if you're curious and really want to dive into migrations, the best place to go (other than the code itself) is the [official docs](#). It's the most up-to-date and does a pretty good job of explaining how things work.

Remember that in the general case, you are dealing with either:

1. **Schema Migrations** - a change to the structure of the database or tables with no change to the data. This is the most common type, and Django can generally create these migrations for you automatically.
2. **Data Migrations** - a change to the data, or loading new data. Django cannot generate these for you. They must be created manually using the `RunPython` migration.

So pick the migration that is correct for you, run `makemigrations` and then just be sure to update your migration files every time you update your model - and that's more or less it. That will allow you to keep your migrations stored with your code in git and ensure that you can update your database structure without having to lose data.

Happy migrating!

NOTE: We will be building on the migrations that we created in this chapter, and problems will arise if your migration files do not match exactly (including the name of the files) with the migration files from the [repo](#). Compare your code/migration files with code/migration files from the repo. Fix any discrepancies. Thanks!

Exercises

1. At this point if you drop your database, run migrations and then run the tests you will have a failing test because there are no `MarketingItems` in the database. For testing you have two options:
 - Load the data in the test (or use a fixture).
 - Load the data by using a datamigration.

The preferred option for this case is to create a data migration to load the `MarketingItems`. Can you explain why?. Create the migration.

NOTE: For some fun (and a somewhat ugly hack) we can add a line to create the data to `test.main.testMainPageView.setUpClass`. See if you can figure out what this line does and why adding it will fix your test:

```
1 from main.models import MarketingItems
2 [MarketingItem(**m.__dict__).save() for m in market_items]
```

2. We have a new requirement for **two-factor authentication**. Add a new field to the user model called `second_factor`. Run `./manage.py makemigration payments`. What did it create? Can you explain what is going on in each line of the migration? Now run `./manage.py migrate` and check the database to see the change that has been made. What do you see in the database? Now assume management comes back and says two-factor is too complex for users; we don't want to add it after all. List two different ways you can remove the newly added field using migrations.
3. Let's pretend that MEC has been bought by a big corporation - we'll call it BIGCO. BIGCO loves making things complicated. They say that all users must have a `bigCoID`, and that ID has to follow a certain formula. The ID should look like this: `<first_two_digits_in_name><1-digit-Rank_code><sign-up-date><runningNumber>`
 - `1-digit-Rank_code` = 'Y' for youngling, 'P' for padwan, 'J' for Jedi
 - `sign-up-date` is in the format 'mmddyyyy'

Now create the new field and a migration for the field, then manually write a data migration to populate the new field with the data from the pre-existing users.

Chapter 13

AngularJS Primer



Figure 13.1: Angular

In this chapter, we will dive into client-side programming and specifically talk about the popular client-side framework [AngularJS](#). Referred to as a “Superheroic JavaScript MVW Framework” and as “what HTML would have been, had it been designed for building web-apps”, Angular is perfect for creating dynamic web apps that function like desktop apps. It also plays well with Django and the Django templating system, and it even shares some design principles with Django. Thus, of all the JavaScript frameworks out there, Angular is probably the best suited to use in conjunction with Django.

Django strongly encourages separation of concerns by following an MVC (Model-View-Controller) architecture. Angular follows a similar MVC pattern, although the Angular folks call it MVW (Model View [whatever works for you](#)), which means that you have some freedom to use it in a way that makes sense to you. Since you’re already familiar with Django’s MVC pattern, you can keep the same pattern in mind when working with Angular since MVW is a subset of MVC.

This is one of the main reasons we have chosen to use Angular as the JavaScript framework in this course: Since its overall design is very similar to Django, it allows developers to keep the

same conceptual architecture in mind when working on either the front-end with Angular or the back-end with Django.

What we are covering

Angular is a large framework. It has many many capabilities, and we cannot cover all of it in a chapter. The focus of this chapter is to expose you to enough of Angular that you can build some useful front-end ‘magic’ for your application. We’ll also detail how best to integrate Angular and Django and the trade-offs that should be considered when doing so.

In order to achieve this, let’s start first with Angular and discuss some of the basic components of the framework before adding Django into the mix.

In particular, we’ll cover:

1. Directives
2. Angular Models
3. Data/Expression Bindings
4. Angular Controllers

Let’s start off with user story 4 from the Membership Site Chapter:

US4: User Polls

Determining the truth of the galaxy and balancing the force are both very important goals at MEC. As such, MEC should provide the functionality to poll “padwans” and correlate the results in order to best determine or predict the answers to important topics of the Star Wars galaxy. This includes questions like Kit Fisto vs Aayla Secura, who has the best dreadlocks? Or who would win in a fight, C3PO or R2-D2? Results should also be displayed to the “padwans” so that all shall know the truth.

Again, let’s implement this first in Angular before integrating it into our Django app.

SEE ALSO: Since we cannot possibly cover all of the fundamentals in this chapter, please review [this](#) introductory Angular tutorial before moving on.

Angular for Pony Fliers (aka Django Devs)

“Hello World” just doesn’t make much sense in the Star Wars Galaxy, so let’s start with “Hello Universe” (didn’t see that one coming, did you?). So a minimal Angular app would look like this:

index.html

```
1 <!doctype html>
2 <html lang="en" ng-app=''>
3 <head>
4   <meta charset="UTF-8">
5   <title>{{ msg }} Universe</title>
6   <!-- styles -->
7   <link href="http://netdna.bootstrapcdn.com/bootswatch/3.1.1/
8     yeti/bootstrap.min.css" rel="stylesheet" media="screen">
9 </head>
10 <body>
11   <div class="container">
12     <br><br>
13     <p>What say you, padwan: <input type="text" ng-model="msg"
14       ng-init="msg='Hello'"></p>
15     <p>{{ msg }} Universe!</p>
16   </div>
17   <!-- scripts -->
18   <script
19     src="http://code.jquery.com/jquery-1.11.0.min.js"></script>
20   <script src="http://netdna.bootstrapcdn.com/bootstrap/3.1.1/js/
21     bootstrap.min.js"></script>
22   <script
23     src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.16/
24     angular.min.js" type="text/javascript"></script>
25 </body>
26 </html>
```

Save this file as *index.html* in a new directory called “angular_test” outside of the Django Project.

If you open the above web page in your browser then you will see that whatever you type in the text box is displayed in the paragraph below it. This is an example of data binding in

Angular.

Before we look at the code in greater detail, let's look at Angular [directives](#), which power Angular.

Angular Directives

When a HTML page is rendered, Angular searches through the DOM looking for directives and then executes the associated JavaScript. In general, all of these directives start with `ng` (which is supposed to stand for Angular). You can also prefix them with `data-` if you are concerned about strict w3c validation. Use `data-ng-` in place of `ng-`, in other words.

There are a number of built-in directives, like:

- `ng-click` handles user actions, specifying a Javascript function to call when a user clicks something.
- `ng-hide` controls when an HTML element is visible or hidden, based on a conditional.

These are just a few examples. As we go through this chapter we'll see many uses of directives.

Back to the code

Let's look at the example code:

- **Line 18:** You must include the AngularJS file. This is step one for any Angular app and is of course required. You can either download the file directly from [Angularjs.org](http://angularjs.org) or use a CDN.
- **Line 2** `ng-app` is a directive that is generally added to the main HTML tag or any `div` tag and tells Angular to startup (also known as bootstrapping, but to avoid confusion with the CSS framework Bootstrap we'll refer to it as Angular initialization). Only Angular code that is a child of the element containing `ng-app` will be evaluated by the framework. In our case, since `ng-app` is on the HTML tag, all elements on the page are evaluated by Angular.
- **Line 13:** `ng-model` and `ng-init` are both directives. In this case the directives create a model called `msg` and initialize the initial value (or state) of `msg` to `Hello`. Thus, when the page initially loads, the `msg` model will have the value of `Hello`. So what exactly is a model? Let's digress...

Angular Models

A [model](#) in Angular plays a similar role to a context variable in a Django template. If you think about a simple Django template-

```
1 <p> {{ msg }} Universe! </p>
```

-Django allows you to pass in the value of `msg` from your view function. Assuming the above template was called *index.html*, your view function might look like this:

```
1 def index():  
2     return render_to_response(index.html, {"msg": Hello})
```

This will create the *index* template with a context variable `msg` that has the value of `Hello`, which is exactly what Angular is doing directly in the “template”, on the client-side. The difference is that with Django (in most cases) you can *only* change the value of your context variable by requesting a new value from the server, whereas with Angular you can change the value of the model anytime on the client-side by executing some JavaScript code. This is ultimately how we are going to be able to update the page without a refresh (our main goal), because we can just update the model directly on the client-side.

Return of the code

- **Line 5 and 14:** An Angular model isn’t much use if you can’t update the HTML code as well. Just like Django, Angular uses templates. The Angular template consists of Angular directives and data/expression bindings (as well as filters and form controls, which we haven’t covered yet). Again, when you initialize Angular with the `ng-app` directive, the Angular template is evaluated. We know that directives are evaluated during Angular initialization. Expression bindings are also handled at that time. Expression Bindings use the same markup. In the case of line 5 and 13, the expression bindings `{{ msg }}` are asking Angular to substitute the value of the `msg` model in place of the binding. Again this conceptually works the same way as templates do in Django - it just happens on the client-side. That said, Angular expression bindings are not limited to models (i.e., data binding); you can also use actual expressions:
 - `{{ 1 + 2 }}`: any mathematical operation will work here
 - `{{ model.property }}`: you can access properties of the model
 - `{{ 2 * 4 | currency }}`: an example of an Angular filter, which outputs the results of the expression as a currency

Angular vs Django - re: template tags

The following table below summarizes the similarities between Angular and Django with regards to the topics we have covered thus far:

Features	Angular	Django
Expression Binding	{{ 1+2 }}	Doesn't evaluate expressions
Data Binding	{{ msg }}	{{ msg }}
Filtering	{{ 1+2 currency }}	{{ val currency }}
Directives	ng-app	{% marketing__circle_item %}
Models	ng-model="msg"	{'msg': 'hello'}

So -

1. Filters in Angular can be applied to either expressions (1 + 2) or data (variable_name), while in Django they can only be applied to context variables (variable_name).
2. Directives are defined with HTML attributes in Angular and with template tags in Django.
3. Models are passed into the template from view functions: `render_to_response(index.html {'msg': 'hello'})`

Hopefully this table will serve as a quick guide to remember what the various functionality in Angular does.

There are some rather sizable differences between the two frameworks as well. The main difference is the client-side nature of Angular versus the server-side nature of Django. The client-side nature of Angular leads to one of its most prominent features, which is marketed as two-way data binding.

Angular Data Binding Explained

To understand Angular's two way data-binding, it's helpful to first look at one-way data binding. We'll use a Django specific example but most one-way data binding systems work in a similar way.

One-way data binding

Above is an image depicting one-way data binding as it happens in Django. The process begins with an HTTP request. In the case of a GET request, sent from the end user to the server, Django handles the request by first routing it to the appropriate controller with *urls.py* and **views.py*, respectively. The view then handles the request, possibly grabbing some data from the database (and perhaps manipulating it), and sends a response that generally looks something like this:

```
return render_to_response('index.html',  
    {'marketing_items':market_items})
```

Here, we pass in two arguments to the `render_to_response` function - the first being the *template name*. Meanwhile, the second is technically the *template context*; however, since it generally consists of a model or set of models, we'll refer to it as a *model* for this discussion.

Django's template processor parses the template substituting the data binding expressions and template tags with the appropriate data, then produces a view, which is finally returned to the user to see.

At that point, the data binding process is complete. If the user changes data on the web page (e.g., the view), those changes won't be reflected in the model unless the change is sent back to the server (usually with a POST request) and the data binding process starts all over again.

Since data is bound only once (in the response), this technique is referred to as one-way data binding, which works well for a number of use cases.

Two-way data binding

Working within the Django paradigm, to keep the data on the view in sync with the data in the model (when you have an interactive page) can be difficult and would require a number of requests back and forth to and from the server.

However, if we remove the combining of the model and template from the server-side and place it on the client-side, then we can significantly decrease the number of required calls to

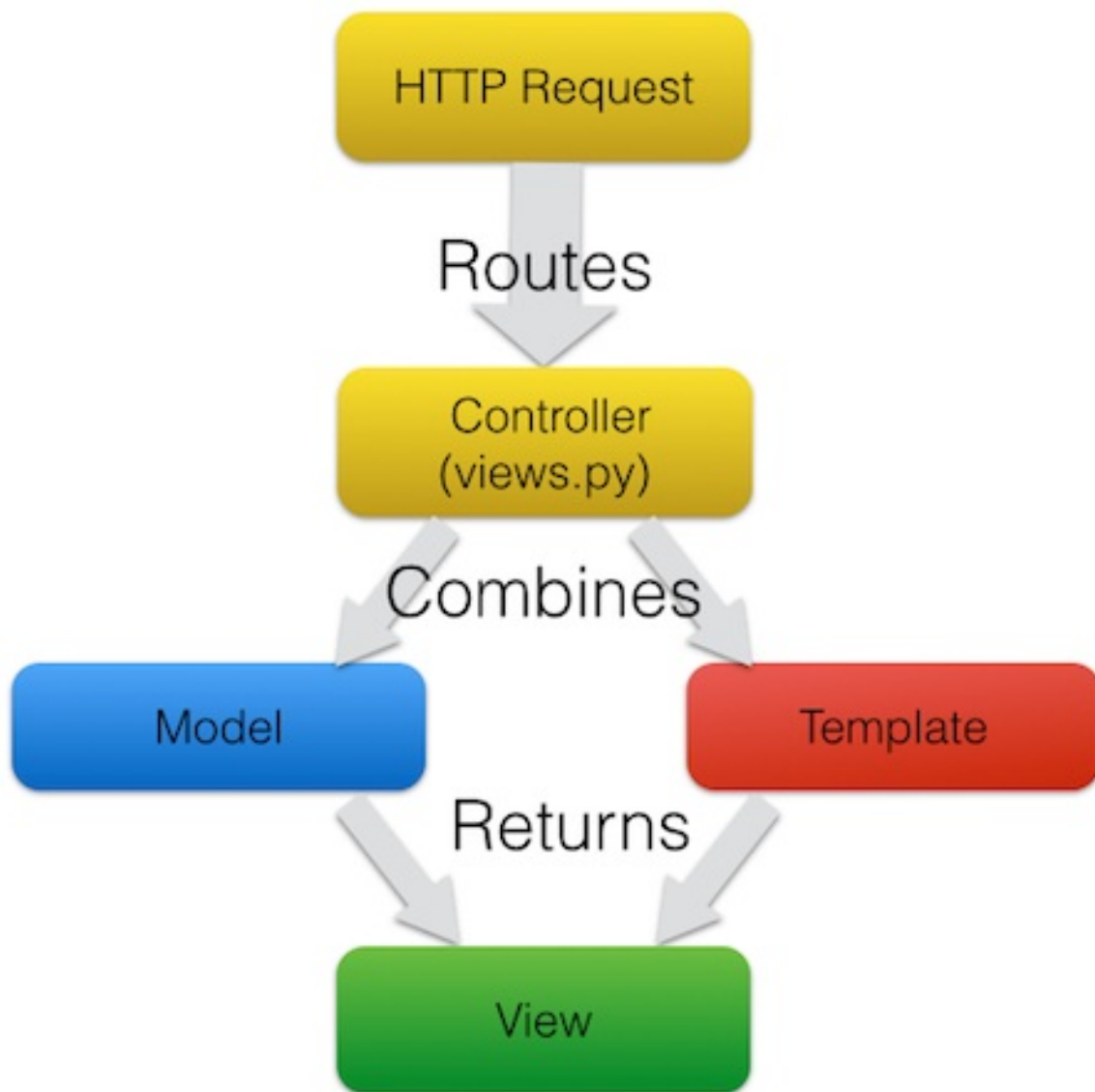


Figure 13.2: one way data binding

and from the server in order to keep the model and view in sync. This results in a site that appears much more responsive and functional to the end user.

Angular's two way data-binding is depicted in the image above. The process starts with a user action (rather than a request) - i.e., typing a value into a text box - then the JavaScript event handler updates the associated model and triggers Angular's "template compiler" to rebuild the view with the newly updated data from the model.

In a similar fashion, if you write JavaScript code that changes the model directly, Angular will again fire off the "template compiler" to ensure the view stays up-to-date with the model.

Regardless of whether you make a change to the view or the model, Angular ensures the two are kept in sync.

This is two-way data binding in action, which results in keeping everything (model and view) in sync. Since it's all done on the client-side, it's also done very quickly, without requiring multiple trips to the server or a page refresh.

To be completely fair, you don't need to use Angular to keep the model and view up-to-date. Since Angular uses vanilla JavaScript behind the scenes, you could use AJAX to achieve similar results. Angular makes the process much, *much* easier, allowing you to do a lot with little code.

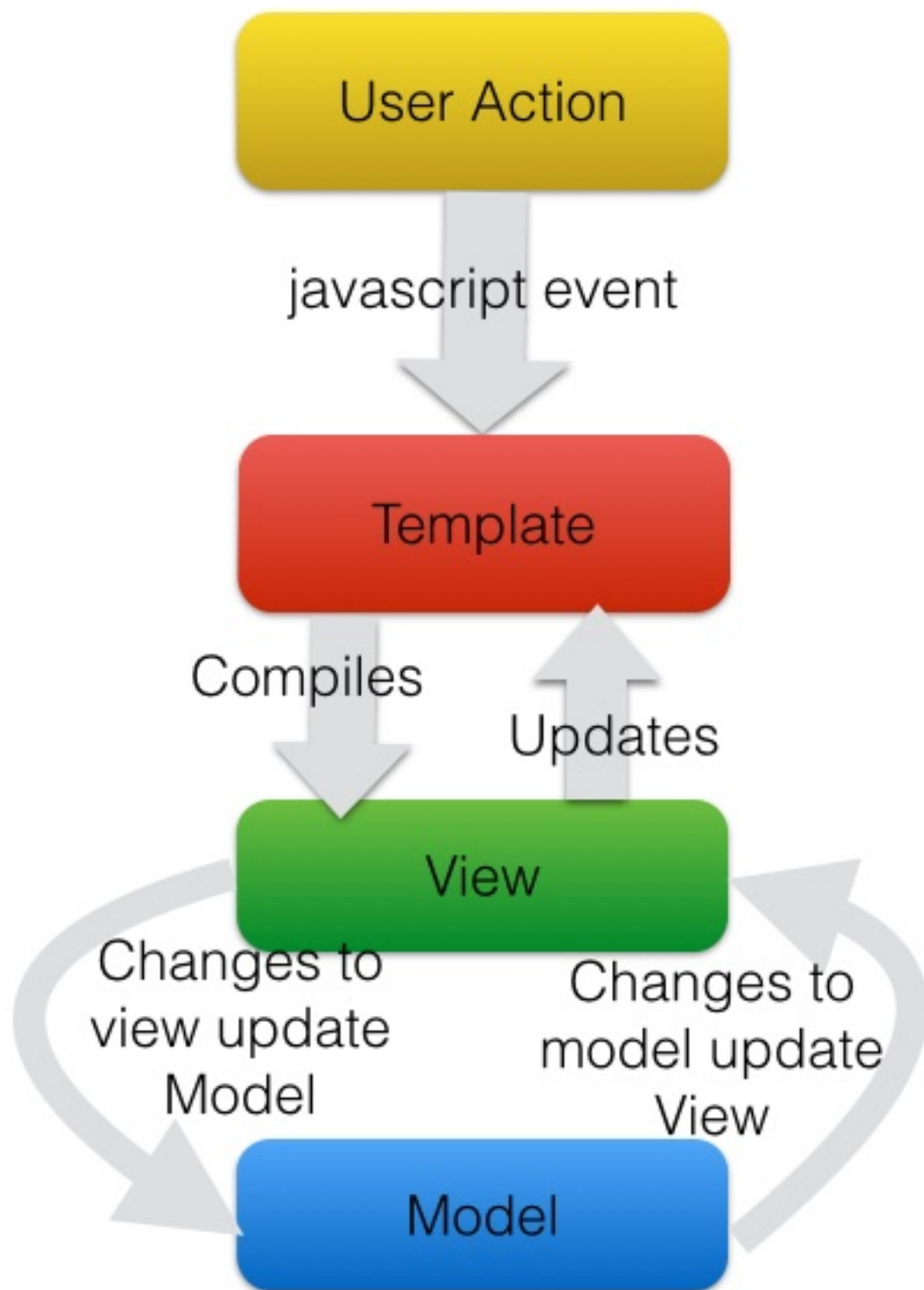


Figure 13.3: two way data binding

Building User Polls

With that brief introduction done, there's quite a lot you can already accomplish in Angular. Let's take it one step further and build out the User Polls functionality. Polls? That should sound familiar, as you either have gone through the official Django tutorial or at least heard of it. Let's do the same type of poll application in Angular.

Quick start

We'll start with just the poll in a standalone HTML file. Again, save this file in the "angular_test" directory as *polls.html*.

First, let's create some basic markup for the poll:

```
1 <!doctype html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <title>Poll: Who's the most powerful Jedi?</title>
6   <!-- styles -->
7   <link href="http://netdna.bootstrapcdn.com/bootswatch/3.1.1/yeti/
8     bootstrap.min.css" rel="stylesheet" media="screen">
9 </head>
10 <body>
11   <div class="container">
12     <div class="row">
13       <div class="col-md-8">
14         <h1>Poll: Who's the most powerful Jedi?</h1>
15         <br>
16         <span class="glyphicon glyphicon-plus"></span>
17         <strong>Yoda</strong>
18         <span class="pull-right">40%</span>
19         <div class="progress">
20           <div class="progress-bar progress-bar-danger"
21             role="progressbar" aria-value="40"
22             aria-valuemin="0" aria-valuemax="100" style="width:
23               40%;">
24           </div>
25         </div>
26         <span class="glyphicon glyphicon-plus"></span>
```

```

25 <strong>Qui-Gon Jinn</strong>
26 <span class="pull-right">30%</span>
27 <div class="progress">
28     <div class="progress-bar progress-bar-info"
29         role="progressbar" aria-value="30"
30         aria-valuemin="0" aria-valuemax="100" style="width:
31             30%;">
32     </div>
33 </div>
34 <span class="glyphicon glyphicon-plus"></span>
35 <strong>Obi-Wan Kenobi</strong>
36 <span class="pull-right">10%</span>
37 <div class="progress">
38     <div class="progress-bar progress-bar-warning"
39         role="progressbar" aria-value="10"
40         aria-valuemin="0" aria-valuemax="100" style="width:
41             10%;">
42     </div>
43 </div>
44 <span class="glyphicon glyphicon-plus"></span>
45 <strong>Luke Skywalker</strong>
46 <span class="pull-right">5%</span>
47 <div class="progress">
48     <div class="progress-bar progress-bar-success"
49         role="progressbar" aria-value="5"
50         aria-valuemin="0" aria-valuemax="100" style="width:
51             5%;">
52     </div>
53 </div>
54 <span class="glyphicon glyphicon-plus"></span>
55 <strong>Me... of course</strong>
56 <span class="pull-right">15%</span>
57 <div class="progress progress-striped active">
58     <div class="progress-bar" role="progressbar"
59         aria-valuenow="15"
60         aria-valuemin="0" aria-valuemax="100" style="width:
61             15%;">
62     <span class="sr-only">15% Complete</span>
63     </div>
64 </div>

```



```

57     </div>
58   </div>
59 </div>
60 <!-- scripts -->
61 <script
    src="http://code.jquery.com/jquery-1.11.0.min.js"></script>
62 <script src="http://netdna.bootstrapcdn.com/bootstrap/3.1.1/js/
    bootstrap.min.js"></script>
63 <script
    src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.16/
    angular.min.js" type="text/javascript"></script>
64 </body>
65 </html>

```

This snippet above relies on Bootstrap to create a simple list of choices each with a + next to them and a progress bar below. We've put in some default values just so you can see what it might look like after people have voted. The screenshot is below.

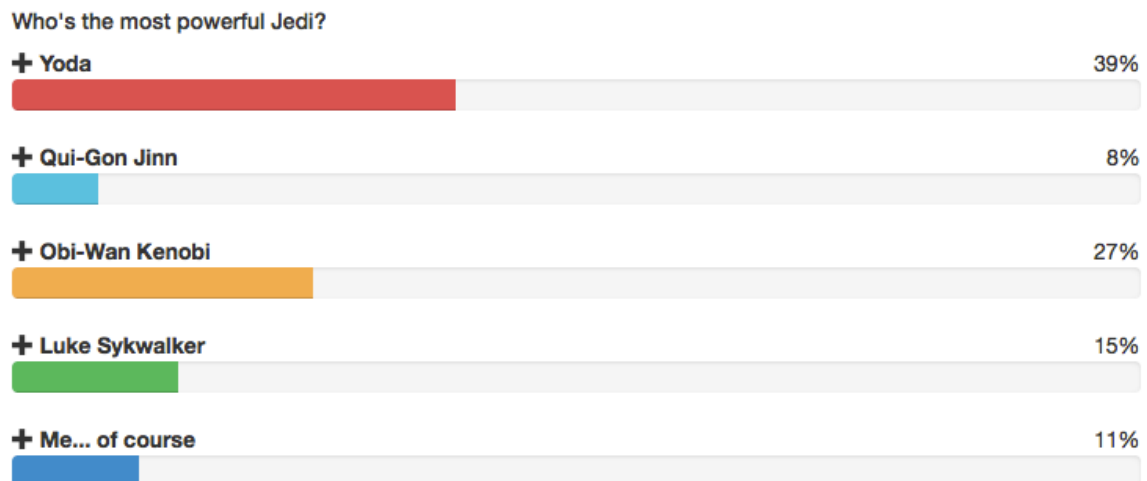


Figure 13.4: User Poll

As for the user functionality, when the + is clicked, the progress bar below the Jedi should increment. Let's add that now. In other words, let's add Angular!

Bootstrap Angular

How about we start with Yoda. Update the top of the file like so:

```

1 <!doctype html>
2 <html lang="en" ng-app=''>
3 <head>
4   <meta charset="UTF-8">
5   <title>Poll: Who's the most powerful Jedi?</title>
6   <!-- styles -->
7   <link href="http://netdna.bootstrapcdn.com/bootswatch/3.1.1/yeti/
8     bootstrap.min.css" rel="stylesheet" media="screen">
9 </head>
10 <body>
11   <div class="container">
12     <div class="row">
13       <div class="col-md-8">
14         <h1>Poll: Who's the most powerful Jedi?</h1>
15         <br>
16         <span ng-click='votes_for_yoda = votes_for_yoda + 1'
17           ng-init="votes_for_yoda=0" class="glyphicon
18             glyphicon-plus"></span>
19         <strong>Yoda</strong>
20         <span class="pull-right">{{ votes_for_yoda }}</span>
21         <div class="progress">
22           <div class="progress-bar progress-bar-danger"
23             role="progressbar" aria-value="{{ votes_for_yoda }}"
24             aria-valuemin="0" aria-valuemax="100" style="width:
25               {{ votes_for_yoda }}%;">
26           </div>
27         </div>
28       </div>
29     </div>
30   </div>

```

- **Line 2:** ng-app will bootstrap in Angular and get everything working.
- **Line 16:** ng-click='votes_for_yoda = votes_for_yoda + 1' - this Angular directive creates a click handler that increments the model votes_for_yoda by 1. It will be called each time the user clicks on the + span.
- **Line 16:** ng-init="votes_for_yoda=0" - this Angular directive initializes the value of votes_for_yoda to 0.
- **Line 18, 20, and 21:** the expression {{ votes_for_yoda }} uses two-way data binding to keep the values in sync with the votes_for_yoda model. Since these values control the progress bar, the bar will now grow each time we click on the plus.

Once this is working, update the code for all the remaining Jedi's.

```

1 <!doctype html>
2 <html lang="en" ng-app=''>
3 <head>
4   <meta charset="UTF-8">
5   <title>Poll: Who's the most powerful Jedi?</title>
6   <!-- styles -->
7   <link href="http://netdna.bootstrapcdn.com/bootswatch/3.1.1/yeti/
8     bootstrap.min.css" rel="stylesheet" media="screen">
9 </head>
10 <body>
11   <div class="container">
12     <div class="row">
13       <div class="col-md-8">
14         <h1>Poll: Who's the most powerful Jedi?</h1>
15         <br>
16         <span ng-click='votes_for_yoda = votes_for_yoda + 1'
17           ng-init="votes_for_yoda=0" class="glyphicon
18             glyphicon-plus"></span>
19         <strong>Yoda</strong>
20         <span class="pull-right">{{ votes_for_yoda }}</span>
21         <div class="progress">
22           <div class="progress-bar progress-bar-danger"
23             role="progressbar" aria-value="{{ votes_for_yoda }}"
24             aria-valuemin="0" aria-valuemax="100" style="width:
25               {{ votes_for_yoda }}%;">
26           </div>
27         </div>
28         <span ng-click='votes_for_qui = votes_for_qui + 1'
29           ng-init="votes_for_qui=0" class="glyphicon
30             glyphicon-plus"></span>
31         <strong>Qui-Gon Jinn</strong>
32         <span class="pull-right">{{ votes_for_qui }}</span>
33         <div class="progress">
34           <div class="progress-bar progress-bar-info"
35             role="progressbar" aria-value="{{ votes_for_qui }}"
36             aria-valuemin="0" aria-valuemax="100" style="width:
37               {{ votes_for_qui }}%;">
38           </div>
39         </div>
40       </div>
41     </div>
42   </div>

```

```

32     <span ng-click='votes_for_obi = votes_for_obi + 1'
      ng-init="votes_for_obi=0" class="glyphicon
        glyphicon-plus"></span>
33     <strong>Obi-Wan Kenobi</strong>
34     <span class="pull-right">{{ votes_for_obi }}</span>
35     <div class="progress">
36         <div class="progress-bar progress-bar-warning"
          role="progressbar" aria-value="{{ votes_for_obi }}"
37         aria-valuemin="0" aria-valuemax="100" style="width:
          {{ votes_for_obi }}%;">
38         </div>
39     </div>
40     <span ng-click='votes_for_luke = votes_for_luke + 1'
      ng-init="votes_for_luke=0" class="glyphicon
        glyphicon-plus"></span>
41     <strong>Luke Skywalker</strong>
42     <span class="pull-right">{{ votes_for_luke }}</span>
43     <div class="progress">
44         <div class="progress-bar progress-bar-success"
          role="progressbar" aria-value="{{ votes_for_luke }}"
45         aria-valuemin="0" aria-valuemax="100" style="width:
          {{ votes_for_luke }}%;">
46         </div>
47     </div>
48     <span ng-click='votes_for_me = votes_for_me + 1'
      ng-init="votes_for_me=0" class="glyphicon
        glyphicon-plus"></span>
49     <strong>Me... of course</strong>
50     <span class="pull-right">{{ votes_for_me }}</span>
51     <div class="progress progress-striped active">
52         <div class="progress-bar" role="progressbar"
          aria-value="{{ votes_for_me }}"
53         aria-valuemin="0" aria-valuemax="100" style="width:
          {{ votes_for_me }}%;">
54         </div>
55     </div>
56 </div>
57 </div>
58 </div>
59 <!-- scripts -->

```

```

60     <script
        src="http://code.jquery.com/jquery-1.11.0.min.js"></script>
61     <script src="http://netdna.bootstrapcdn.com/bootstrap/3.1.1/js/
62         bootstrap.min.js"></script>
63     <script
        src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.16/
64         angular.min.js" type="text/javascript"></script>
65     </body>
66 </html>

```

With that, you have a simple user poll that uses progress bars to show which option has the most votes. This will update automatically without making any calls to the back-end. This however puts a lot of logic into our actual HTML and can thus make maintenance a bit difficult. Ideally we would like a separate place to keep our logic.

We can tackle this problem by using an Angular controller.

Angular Controller

We haven't talked about controllers in Angular yet, but for now you can think of an Angular controller as your `views.py` file that handles the HTTP requests from the browser. Of course with Angular the requests are coming from individual client actions, but the two more or less serve the same purpose.

If we modify the above code to use a controller, it might look like this:

```

1  <!doctype html>
2  <html lang="en" ng-app="mecApp">
3  <head>
4      <meta charset="UTF-8">
5      <title>Poll: Who's the most powerful Jedi?</title>
6      <!-- styles -->
7      <link href="http://netdna.bootstrapcdn.com/bootswatch/3.1.1/yeti/
8          bootstrap.min.css" rel="stylesheet" media="screen">
9  </head>
10 <body>
11     <div class="container" ng-controller="UserPollCtrl">
12         <div class="row">
13             <div class="col-md-8">
14                 <h1>Poll: Who's the most powerful Jedi?</h1>
15                 <br>

```

```

16 <span ng-click="vote('votes_for_yoda')" class="glyphicon
    glyphicon-plus"></span>
17 <strong>Yoda</strong>
18 <span class="pull-right">{{ votes_for_yoda }}</span>
19 <div class="progress">
20   <div class="progress-bar progress-bar-danger"
        role="progressbar" aria-value="{{ votes_for_yoda }}"
21     aria-valuemin="0" aria-valuemax="100" style="width:
        {{ votes_for_yoda }}%;>
22   </div>
23 </div>
24 <span ng-click="vote('votes_for_qui')" class="glyphicon
    glyphicon-plus"></span>
25 <strong>Qui-Gon Jinn</strong>
26 <span class="pull-right">{{ votes_for_qui }}</span>
27 <div class="progress">
28   <div class="progress-bar progress-bar-info"
        role="progressbar" aria-value="{{ votes_for_qui }}"
29     aria-valuemin="0" aria-valuemax="100" style="width:
        {{ votes_for_qui }}%;>
30   </div>
31 </div>
32 <span ng-click="vote('votes_for_obi')" class="glyphicon
    glyphicon-plus"></span>
33 <strong>Obi-Wan Kenobi</strong>
34 <span class="pull-right">{{ votes_for_obi }}</span>
35 <div class="progress">
36   <div class="progress-bar progress-bar-warning"
        role="progressbar" aria-value="{{ votes_for_obi }}"
37     aria-valuemin="0" aria-valuemax="100" style="width:
        {{ votes_for_obi }}%;>
38   </div>
39 </div>
40 <span ng-click="vote('votes_for_luke')" class="glyphicon
    glyphicon-plus"></span>
41 <strong>Luke Skywalker</strong>
42 <span class="pull-right">{{ votes_for_luke }}</span>
43 <div class="progress">
44   <div class="progress-bar progress-bar-success"
        role="progressbar" aria-value="{{ votes_for_luke }}"

```

```

45         aria-valuemin="0" aria-valuemax="100" style="width:
           {{ votes_for_luke }}%";>
46     </div>
47 </div>
48 <span ng-click="vote('votes_for_me')" class="glyphicon
   glyphicon-plus"></span>
49 <strong>Me... of course</strong>
50 <span class="pull-right">{{ votes_for_me }}</span>
51 <div class="progress progress-striped active">
52     <div class="progress-bar" role="progressbar"
       aria-value="{{ votes_for_me }}"
53       aria-valuemin="0" aria-valuemax="100" style="width:
         {{ votes_for_me }}%";>
54     </div>
55 </div>
56 </div>
57 </div>
58 </div>
59 <!-- scripts -->
60 <script
   src="http://code.jquery.com/jquery-1.11.0.min.js"></script>
61 <script src="http://netdna.bootstrapcdn.com/bootstrap/3.1.1/js/
   bootstrap.min.js"></script>
62 <script
   src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.16/
   angular.min.js" type="text/javascript"></script>
63 <script type="text/javascript">
64     var mecApp = angular.module('mecApp', []);
65
66     mecApp.controller('UserPollCtrl', function($scope) {
67         $scope.votes_for_yoda = 80;
68         $scope.votes_for_qui = 30;
69         $scope.votes_for_obi = 20;
70         $scope.votes_for_luke = 10;
71         $scope.votes_for_me = 30;
72
73         $scope.vote = function(voteModel) {
74             $scope[voteModel] = $scope[voteModel] + 1;
75         };
76     });
77
78

```

```

79     </script>
80   </body>
81 </html>

```

Let's go through this one line at a time:

- **Line 2:** `<html lang="en" ng-app="mecApp">`. Notice on this line we passed a name into `ng-app`. Normally when we pass no name we are using the default Angular module. When we pass a name we can create a module with that name. This is important because you can declare multiple `ng-apps` on the same page and have a different module for each. Also, this helps to prevent name classes in large applications accidentally overriding each other.
- **Line 11:** An Angular module, which in our case is called “mecApp”, can contain multiple controllers. When you define a controller on a div, it will apply to all Angular directives / bindings declared inside that div. By using the directive: `ng-controller="UserPollCtrl"` we are declaring a controller with the name “UserPollCtrl” and assigning it to handle everything inside the div.
- **Line 16:** `ng-click="vote('votes_for_yoda')"`. Here we are using the `ng-click` directive to set up a click handler like we did in the previous example. The difference here is instead of putting the code to execute in-line, we are telling the click handler to call a function called `vote()` and passing it the string value `votes_for_yoda`. As you might have guessed, the `vote()` function will be defined in the `UserPollCtrl` controller. Also note that we are passing in a string to `vote` and not the actual `votes_for_yoda` model (this will become important later).
- **Lines 16, 18, 20:** Just like in the previous example we are using the model `votes_for_yoda` to update the values in the progress bar so that the bar length increases as we click on the +.
- **Lines 65-79:** Here we are inserting the actual JavaScript that defines the controller and its behavior. Oftentimes in Angular apps, controllers are put in a separate file called `controllers.js`, but to keep things simple we will just put everything in one file.

Let's look at the controller in more detail. Here is the code again:

```

1 <script type="text/javascript">
2   var mecApp = angular.module('mecApp', []);
3
4   mecApp.controller('UserPollCtrl', function($scope) {
5     $scope.votes_for_yoda = 80;
6     $scope.votes_for_qui = 30;

```



```

7     $scope.votes_for_obi = 20;
8     $scope.votes_for_luke = 10;
9     $scope.votes_for_me = 30;
10
11     $scope.vote = function(voteModel) {
12         $scope[voteModel] = $scope[voteModel] + 1;
13     };
14 });
15 </script>

```

- **Line 2:** We declare an Angular module called `mecApp`, which is what we named the module in our `ng-app` directive in the earlier example. If we don't declare a module with the name used in `ng-app` then things won't work. We created our module with two arguments - the first is the name of the module and the second is an array of the names of all the modules this module depends on. Since we don't depend on any other modules we just pass an empty array.
- **Line 4:** Here we create our controller by calling the controller method on our module. We pass in two arguments - the first is the name of the controller and the second is a constructor function which will be called when Angular parses the DOM and finds the `ng-controller` directive. The constructor method is always passed the `$scope` variable which Angular uses as context when evaluating expressions - i.e., (`{{ votes_for_yoda }}`) - or for propagating events.

What is Scope in Angular?

In Angular you can think of Scope or more specifically `$scope` as a data structure used to share information between the controller and the view. So any value you set on `$scope` in the controller will be reflected in the view during expression evaluation. An example may help understanding here.

Given the following HTML-

```

1 <div ng-controller='DaysCtrl'>
2   <p>Today is {{ day_of_week }} </p>
3 </div>

```

-and the following JS-

```

1 app.controller('DaysCtrl', function($scope) {
2     $scope.day_of_week = "Monday";
3 });

```

-will cause `<p>Today is {{ day_of_week }}</p>` to be rendered as `<p>Today is Monday</p>` because `day_of_week` has its value set to Monday in the controller. In some ways this is similar to a template context that is created in a Django view. The difference is that scope works two ways (two-way binding). So with the following code-

```
1 <div ng-controller='DaysCtrl'>
2   <input id="day-of-week" ng-model="day_of_week" placeholder="what
   day is it?">
3   <p>Today is {{ day_of_week }} </p>
4 </div>
```

When the user types in a value in the “day-of-week” input then that value is stored in `$scope.day_of_week` and can be accessed from the controller as well as in the next line of the html - `<p>Today is {{ day_of_week }} </p>`.

For our purposes that is really all you need to know about scope.

Back to the code

- **Lines 5-9:** Here we define five variables in the `$scope` (`votes_for_yoda`, `votes_for_qui`, and so forth) and set their initial values. By defining the variables in the `$scope` data structure, we make them available to the view so that they can be used in expressions. In other words, after line 5 is run, `{{ votes_for_yoda }}` in the view will evaluate to 80.
- **Lines 11-13:** In the HTML part of our example we used the following `ng-click` directive: `ng-click="vote('votes_for_yoda')"`. Lines 11-13 define the `vote()` function that is called by the `ng-click` directive. If no such function were defined, Angular would silently fail and give no indication the function wasn't defined.
- **Line 12:** When `voteModel` is a string (in our case, something like the string `votes_for_yoda`), then this expression is just looking up the value by name in the `$scope` data structure. This works much like a dictionary in Python. By using this syntax, we can use the same `vote()` function for whatever we want to vote for; we just pass in the name of the model we want to increment, and it will be incremented.

That's it. We probably want to set the default values to 0 for everybody to be fair; we set non-zero values in the example above to show that the default values would actually be reflected in the view.

Conclusion

That certainly isn't all there is to Angular. Again, Angular is a massive framework with a lot to offer, but this is enough to get us started building some features in Angular. As we go through the process of developing these user stories, we will most surely stumble upon some more features of Angular, but for now this should get your feet wet. The key here is to understand the similarities between Angular and Django (e.g., a templating language that is very similar to Django's templating language) and its differences (e.g., client-side vs server-side, two-way data binding vs one-way data binding). If you can grasp these core Angular concepts, you will be well on your way to developing some cool features with Angular.

In the next chapter we will look at integrating Angular with Django, talk about how to structure things so the two frameworks play nice together, and also talk about how to use Angular to update the back-end.

Exercises

1. Our User Poll example uses progress bars, which are showing a percentage of votes. But our vote function just shows the raw number of votes, so the two don't actually match up. Can you update the `vote()` function to return the current percentage of votes. (HINT: You will also need to keep track of the total number of votes.)
2. Spend a bit more time familiarizing yourself with Angular. Here are a few excellent resources to go through:
 - The Angular PhoneCat Tutorial - <https://docs.angularjs.org/tutorial>.
 - The Angular team is very good at releasing educational talks and videos about Angular; check those out on the [Angular JS YouTube Channel](#).
 - There is a really good [beginner to expert series](#) from the ng-newsletter, which is highly recommended.

Chapter 14

Djangular: Integrating Django and Angular

In the last chapter we went through a brief introduction to Angular and how to use it to deliver the User Polls functionality from our User Stories. However, there are a few issues with that implementation:

1. We implemented it stand-alone in a separate HTML page, so we still need to integrate it into our Django Project.
2. We hard coded a number of values, so it will only work for a single type of poll; we need a more general solution.
3. As implemented, the user poll will not share poll data between users (because the model only exists on the client side); we need to send the votes back to the server so all user votes can be counted and shared.

This chapter will focus on how to accomplish those three things. We will look at the Django backend as well as the Django and Angular templates. Let's start first with the Django back-end.

The User Polls Backend Data Model

In order to store the voting data and thus share it amongst multiple users, we first need to create the appropriate models. Let's create this as a completely separate Django App, as we may want to use it in other projects.

To do that, start with the command line:

```
1 $ ./manage.py startapp djangular_polls
```

NOTE: We'll skip talking about unit testing for this chapter to focus on the task at hand - Angular. That's no excuse for you to not continue writing tests on your own, though.

With the new app created, we can update the `djangular_polls/models.py` file and add our new model:

```
1 from django.db import models
2
3
4 class Poll(models.Model):
5     title = models.CharField(max_length=255)
6     publish_date = models.DateTimeField(auto_now=True)
7
8     def poll_items(self):
9         return self.pollitem_set.all()
10
11
12 class PollItem(models.Model):
13     poll = models.ForeignKey(Poll, related_name='items')
14     name = models.CharField(max_length=30)
15     text = models.CharField(max_length=300)
16     votes = models.IntegerField(default=0)
17     percentage = models.DecimalField(
18         max_digits=5, decimal_places=2, default=0.0)
19
20     class Meta:
21         ordering = ['-text']
```

We just created two models:

1. `Poll()`: represents the overall poll - i.e., “who is the best Jedi”.
2. `PollItem()`: the individual items to vote on in the poll.

Notice within `PollItem()` we also added a default ordering so that the returned JSON will always be in the same order. (This will become important later.) Don't forget to add the app to `INSTALLED_APPS` in the *settings.py* file and migrate the changes to your database:

```
1 $ ./manage.py makemigrations djangular_polls
2 $ ./manage.py migrate djangular_polls
```

Now you have a nicely updated database and are ready to go.

A REST Interface for User Polls

The next step is to make a nice RESTful interface where we can consume the `Poll()` and the associated `PollItem()`. We can, of course, use the Django REST Framework for this.

Serializer

First create the serializers:

```
1 from rest_framework import serializers
2 from django_polls.models import Poll, PollItem
3
4
5 class PollItemSerializer(serializers.ModelSerializer):
6     class Meta:
7         model = PollItem
8         fields = ('id', 'name', 'text', 'votes', 'percentage')
9
10
11 class PollSerializer(serializers.ModelSerializer):
12     items = PollItemSerializer(many=True)
13
14     class Meta:
15         model = Poll
16         fields = ('title', 'publish_date', 'items')
```

Save this code in a file called *serializers.py*.

Since we covered DRF in a previous chapter, we won't go into details on the serializers. That said, the intent is to create the following JSON for each `Poll`:

```
1 {
2     'title': 'Who is the best jedi',
3     'publish_date': datetime.datetime(2014, 5, 20, 5, 47, 56, 630638),
4     'items': [
5         {'id': 4, 'name': 'Yoda', 'text': 'Yoda', 'percentage':
6           Decimal('0.00'), 'votes': 0},
7         {'id': 5, 'name': 'Vader', 'text': 'Vader', 'percentage':
8           Decimal('0.00'), 'votes': 0},
9         {'id': 6, 'name': 'Luke', 'text': 'Luke', 'percentage':
10          Decimal('0.00'), 'votes': 0}
```



```

8     ]
9 }

```

This makes it easy to get a `Poll()` and its associated `PollItem()` in one JSON call. To do this, we gave the relationship, between a poll and its items, a name in the `PollItem()` model:

```

1 poll = models.ForeignKey(Poll, related_name='items')

```

The `related_name` provides a name to use for the reverse relationship from `Poll`.

Next, in the `PollSerializer()` class we included the `items` property:

```

1 items = PollItemSerializer(many=True)

```

Why? Well, without it the serializer will just return the primary keys.

Then we also have to include the reverse relationship in the fields list:

```

1 fields = ('title', 'publish_date', 'items')

```

Those three things will give us the JSON that we are looking for. This would be a good thing to write some unit tests for to make sure it's working appropriately. *Yes, try it on your own.*

Endpoints

Next, we need to create the REST endpoints in `djangular_polls/json_views.py`.

```

1 from djangular_polls.serializers import PollSerializer,
   PollItemSerializer
2 from djangular_polls.models import Poll, PollItem
3 from rest_framework import mixins
4 from rest_framework import generics
5
6
7 class PollCollection(mixins.ListModelMixin,
8                     mixins.CreateModelMixin,
9                     generics.GenericAPIView):
10
11     queryset = Poll.objects.all()
12     serializer_class = PollSerializer
13
14     def get(self, request):
15         return self.list(request)

```

```

16
17     def post(self, request):
18         return self.create(request)
19
20
21 class PollMember(mixins.RetrieveModelMixin,
22                  mixins.UpdateModelMixin,
23                  mixins.DestroyModelMixin,
24                  generics.GenericAPIView):
25
26     queryset = Poll.objects.all()
27     serializer_class = PollSerializer
28
29     def get(self, request, *args, **kwargs):
30         return self.retrieve(request, *args, **kwargs)
31
32     def put(self, request, *args, **kwargs):
33         return self.update(request, *args, **kwargs)
34
35     def delete(self, request, *args, **kwargs):
36         return self.destroy(request, *args, **kwargs)
37
38
39 class PollItemCollection(mixins.ListModelMixin,
40                          mixins.CreateModelMixin,
41                          generics.GenericAPIView):
42
43     queryset = PollItem.objects.all()
44     serializer_class = PollItemSerializer
45
46     def get(self, request):
47         return self.list(request)
48
49     def post(self, request):
50         return self.create(request)
51
52
53 class PollItemMember(mixins.RetrieveModelMixin,
54                     mixins.UpdateModelMixin,
55                     mixins.DestroyModelMixin,

```

```

56         generics.GenericAPIView):
57
58     queryset = PollItem.objects.all()
59     serializer_class = PollItemSerializer
60
61     def get(self, request, *args, **kwargs):
62         return self.retrieve(request, *args, **kwargs)
63
64     def put(self, request, *args, **kwargs):
65         return self.update(request, *args, **kwargs)
66
67     def delete(self, request, *args, **kwargs):
68         return self.destroy(request, *args, **kwargs)

```

Again, this should all be review from the Django REST Framework chapter. Basically we are just creating the endpoints for `pPoll()` and the associated `PollItem()` with basic CRUD functionality.

Routing

Don't forget to update our Django routing in `djangular_polls/urls.py`:

```

1 from django.conf.urls import patterns, url
2 from djangular_polls import json_views
3
4 urlpatterns = patterns(
5     'djangular_polls.main_views',
6     url(r'^polls/$', json_views.PollCollection.as_view(),
7         name='polls_collection'),
8     url(r'^polls/(?P<pk>[0-9]+)$', json_views.PollMember.as_view()),
9     url(r'^poll_items/$', json_views.PollItemCollection.as_view(),
10        name="poll_items_collection"),
11     url(r'^poll_items/(?P<pk>[0-9]+)$',
12        json_views.PollItemMember.as_view()),

```

Here, we are creating the standard endpoint addresses.

NOTE: Notice in the URL patterns listed above that there are no trailing `/`. This is important because some parts of Angular don't like trailing slashes. To

complicate things, Django has a default setting of `APPEND_SLASH=True`, which means if you (or Angular) request `polls/1` and your routing includes trailing `/`, then Django will automatically redirect by returning HTTP status code 301 to `polls/1/`. It works, but it is not ideal since (a) there is an extra request and (b) for PUT and POST requests, your browser will generally pop up an annoying message about the page being redirected. This isn't what we want. So the best answer when using Angular with Django is to remove the trailing `/` from your routes.

Now add the following code to `django_ecommerce/urls.py`:

```
1 from main.urls import urlpatterns as main_json_urls
2 from djangular_polls.urls import urlpatterns as
  djangular_polls_json_urls
3
4 main_json_urls.extend(djangular_polls_json_urls)
```

Above, we are combining the `json_urls` from our main application with the `json_urls` from our `djangular_polls` application. Then they are added to the overall `django_ecommerce.urls` with the same line we had before:

```
1 url(r'^api/v1/', include(main_json_urls)),
```

This way our entire REST API stays in one place, even if it spans multiple Django apps. Now we should have two new URLs in our REST API:

```
1 /api/v1/polls/
2 /api/v1/poll_items/
```

Test! Fire up the server and navigate to the browseable API:

1. <http://localhost:8000/api/v1/polls/>
2. http://localhost:8000/api/v1/poll_items/

These endpoints are what we will use to get our Polls functionality working.

Structural Concerns

Now that we have the backend all sorted, we have some choices to make about how to best integrate our frontend framework, Angular, with our backend framework, Django. There are many ways you can do this, and it really depends on what your design goals are and, as always, personal preference. Let's discuss a couple of integration options.

The Single Page Application

The much touted Single Page Application (or SPA) is a web page that acts like a desktop app, which basically means it doesn't ever do a full page refresh. You land on the main page, and from there everything is handled via AJAX and REST calls, and the user basically forgets that s/he is using a web page. To achieve this, you would have only one single "standard" Django View that sends back your entire page (something similar to our `main.views.index`), and from there you let Angular take over and do all the view work, client-side. You would then build a number of JSON views (using DRF) so that Angular could asynchronously fetch all the data it needs and never have to refresh the page again.

The Multi-Single Page App

This is really just a funny way of saying that you want multiple Single Page Apps working together. For example, with our MEC app, we might have one "App" for the non-logged in users that shows the marketing info and allows a user to register and/or access the sign-in page. Then we'd have another "App" for the logged in user, where there would be a screen refresh after registration/sign-in as the user would be "switching apps". Everything after the user logged in (which would translate to our `main.views.user`) would be asynchronous - so no more page refreshes.

In reality, you can choose to have a page refresh whenever you like. Each page refresh could correlate to calling a standard Django view function. When you don't want a page refresh and you need to get or update data, then use Angular to make a REST call to one of your JSON views. This is a common practice when you want to logically split up a large web application into multiple "apps" or "sections".

The "I Just Need a Bit of Angular Love Here" App

Perhaps you have a Django application all ready to go, but you just want to use Angular for a single piece of the application - like a poll or form validation - then we can make a view,

or even an inclusion tag, that uses Angular to accomplish that task and use standard Django views for the remainder of the application.

Those three choices should basically cover every possibility, and the choice really just comes down to how much Angular you want to include. Thus, there is no wrong or right choice; just do what best suits your needs.

Building the Template

For our purposes, we are going to follow the “I Just Need a Bit of Angular Love Here” idea, since we already have an application built. But how much love do we need? Let’s start off with the simplest case. If we just wanted to take the Angular code that we did in the previous chapter and cram it into our existing app, we could do the following...

Let’s add the poll along the right hand side of our users page just below where the status updates are:

user.html template

First, update our *main/user.html* template to include our *djangoangular_polls/_polls.html* template in a new info box:

```
1 {% extends "__base.html" %}
2 {% load staticfiles %}
3 {% block content %}
4 <div id="achievements" class="row member-page hide">
5     {% include "main/_badges.html" %}
6 </div>
7 <div class="row member-page">
8     <div class="col-sm-8">
9         <div class="row">
10             {% include "main/_announcements.html" %}
11             {% include "main/_lateststatus.html" %}
12         </div>
13     </div>
14     <div class="col-sm-4">
15         <div class="row">
16             {% include "main/_jedibadge.html" %}
17             {% include "main/_statusupdate.html" %}
18             {% include "djangoangular_polls/_polls.html" %}
19         </div>
20     </div>
21 </div>
22 {% endblock %}
23 {% block extrajs %}
24 <script src="{% static "js/angular.min.js" %}"
    type="text/javascript"></script>
```

[MOS EISLEY'S CANTINA](#)
[HOME](#)
[ABOUT](#)
[CONTACT](#)
[LOGOUT](#)

ORDERS FROM THE COUNCIL

Star Wars Cosplay from Celebration VI

MEC Costume competition!

Check the video above for some example outfits. We expect all MEC padwans to be there!

JEDI BADGE

Rank: Padwan
Name: jj
Email: jeremy@realpython.com
[Show Achievements](#)

[Click here](#) to make changes to your credit card.

REPORT BACK TO BASE

POLL: WHO'S THE MOST POWERFUL JEDI?

+ Yoda	0%
+ Qui-Gon Jinn	0%
+ Obi-Wan Kenobi	0%
+ Luke Skywalker	0%
+ Me... of course	0%

RECENT STATUS REPORTS

i'm awesome

fdfdsa

Mos Eisley spaceport. You will never find a more wretched hive of scum and villainy.

Adventure. Heh. Excitement. Heh. A Jedi craves not these things.

I find your lack of faith disturbing

Luke, I am your Father.

May the Force be with you

A long time ago, in a galaxy far, far away...

© 2014 [The Jedi Council](#)

Powered by about 37 AT-AT Walkers, Python 3 and Django 1.7

[Back to top](#)

Figure 14.1: Users Page


```

25 <script src="{% static 'js/userPollCtrl.js' %}"
    type="text/javascript"></script>
26 {% endblock %}

```

- Line 18 - This is the line that imports the new `_polls.html` template that will display the Polls info box shown in the above picture.
- Line 23-26 - We're using a new block called `extrajs` which we're using here to load both Angular and the code for our `userPollCtrl.js` - which will include the Angular module.

Now, let's make a few updates to wire up our Polls...

First, let's add `angular.min.js` to our static files. Just save the following file to the "static/js" directory - <https://code.angularjs.org/1.2.16/angular.min.js>. Make sure to add <https://code.angularjs.org/1.2.16/angular.min.js.map> as well.

Next, since we added a new block, `extrajs`, we need to include this in the `__base.html` template as well. Add the following line just before the closing body tag:

```

1 {% block extrajs %}{% endblock %}

```

Then add a new file to the same directory ("static/js") called `userPollCtrl.js` and add the following code:

```

1 var pollsApp = angular.module('pollsApp', []);
2
3 pollsApp.controller('UserPollCtrl', function($scope) {
4     $scope.total_votes = 0;
5     $scope.vote_data = {}
6
7     $scope.vote = function(voteModel) {
8         if (!$scope.vote_data.hasOwnProperty(voteModel)) {
9             $scope.vote_data[voteModel] = {"votes": 0, "percent": 0};
10            $scope[voteModel]=$scope.vote_data[voteModel];
11        }
12        $scope.vote_data[voteModel]["votes"] =
13            $scope.vote_data[voteModel]["votes"] + 1;
14        $scope.total_votes = $scope.total_votes + 1;
15        for (var key in $scope.vote_data) {
16            var item = $scope.vote_data[key];
17            item["percent"] = item["votes"] / $scope.total_votes * 100;

```

```

17     }
18   };
19
20 });

```

_polls.html template

Now let's add the `_polls.html` template within "templates/djangular_polls":

```

1 <section class="info-box" id="polls" ng-app="pollsApp">
2   <div ng-controller='UserPollCtrl'>
3     <h1>Poll: Who's the most powerful Jedi?</h1>
4     <span ng-click='vote("votes_for_yoda")' class="glyphicon
5       glyphicon-plus"></span>
6     <strong>Yoda</strong>
7     <span class="pull-right">[[ votes_for_yoda.percent | number:0
8       ]]%</span>
9     <div class="progress">
10      <div class="progress-bar progress-bar-danger"
11        role="progressbar" aria-value="[[ votes_for_yoda.percent
12          ]]"
13        aria-valuemin="0" aria-valuemax="100" style="width: [[
14          votes_for_yoda.percent ]]%;">
15      </div>
16    </div>
17    <span class="glyphicon glyphicon-plus"
18      ng-click="vote('votes_for_qui')" ></span>
19    <strong>Qui-Gon Jinn</strong>
20    <span class="pull-right">[[ votes_for_qui.percent | number:0
21      ]]%</span>
22    <div class="progress">
23      <div class="progress-bar progress-bar-info"
24        role="progressbar" aria-value="[[ votes_for_qui.percent
25          ]]"
26        aria-valuemin="0" aria-valuemax="100" style="width: [[
27          votes_for_qui.percent ]]%;">
28      </div>
29    </div>
30    <span class="glyphicon glyphicon-plus"
31      ng-click="vote('votes_for_obi')"></span>

```

```

21 <strong>Obi-Wan Kenobi</strong>
22 <span class="pull-right">[[ votes_for_obi.percent | number:0
    ]]%</span>
23 <div class="progress">
24   <div class="progress-bar progress-bar-warning"
25     role="progressbar" aria-value="[[ vote_for_obi.percent ]]"
26     aria-valuemin="0" aria-valuemax="100" style="width: [[
27       votes_for_obi.percent ]]%;">
28   </div>
29 </div>
30 <span class="glyphicon glyphicon-plus" ng-click="vote('votes_for_luke')"></span>
31 <strong>Luke Skywalker</strong>
32 <span class="pull-right">[[ votes_for_luke.percent | number:0
33   ]]%</span>
34 <div class="progress">
35   <div class="progress-bar progress-bar-success"
36     role="progressbar" aria-value="[[ vote_for_luke.percent
37       ]]"
38     aria-valuemin="0" aria-valuemax="100" style="width: [[
39       votes_for_luke.percent ]]%;">
40   </div>
41 </div>
42 <span class="glyphicon glyphicon-plus"
43   ng-click="vote('votes_for_me')"></span>
44 <strong>Me... of course</strong>
45 <span class="pull-right">[[ votes_for_me.percent | number:0
    ]]%</span>
46 <div class="progress">
47   <div class="progress-bar" role="progressbar" aria-value="[[
48     vote_for_me.percent ]]"
49     aria-valuemin="0" aria-valuemax="100" style="width: [[
50       votes_for_me.percent ]]%;">
51   </div>
52 </div>
53 </div>
54 </section>

```

This should look familiar, as it's pretty much the same code we had in the last chapter, except now we are shoving it all into one of our info-boxes, which we introduced way back in the

Bootstrap chapter.

There is one important difference here though, can you spot it?

Notice that we are using different template tags - e.g., `[[ ]]` instead of `{{ }}`. Why? Because Django and Angular both use `{{ }}` by default. Therefore we will instruct angular to use the delimiters `[[ ]]` and let Django continue to use the delimiters `{{ }}` - so we can be clear who is doing the interpolation. Luckily in Angular this is very easy to do. Just add the following lines to the top of the *userPollCtrl.js* file:

```
1 pollsApp.config(function($interpolateProvider){
2   $interpolateProvider.startSymbol('[[')
3   .endSymbol(']]');
4 });
```

The Angular Module should now look like this:

```
1 var pollsApp = angular.module('pollsApp', []);
2
3 pollsApp.config(function($interpolateProvider){
4   $interpolateProvider.startSymbol('[[')
5   .endSymbol(']]');
6 });
7
8 pollsApp.controller('UserPollCtrl', function($scope) {
9   $scope.total_votes = 0;
10  $scope.vote_data = {}
11
12  $scope.vote = function(voteModel) {
13    if (!$scope.vote_data.hasOwnProperty(voteModel)) {
14      $scope.vote_data[voteModel] = {"votes": 0, "percent": 0};
15      $scope[voteModel]=$scope.vote_data[voteModel];
16    }
17    $scope.vote_data[voteModel]["votes"] =
18      $scope.vote_data[voteModel]["votes"] + 1;
19    $scope.total_votes = $scope.total_votes + 1;
20    for (var key in $scope.vote_data) {
21      var item = $scope.vote_data[key];
22      item["percent"] = item["votes"] / $scope.total_votes * 100;
23    }
24  };
25 });
```

So, the config function of an Angular Module allows you to take control of how Angular will behave. In our case, we are using the config function to tell angular to use `[[ ]]` for demarcation, which Django ignores.

With that, you *should* have a functioning User Poll info-box on the main users page. Make sure to manually test this out. Once it's working, then add unit tests.

This app still isn't talking to the backend. Let's rectify that.

Loading Poll Items Dynamically

We have two options here:

1. Return the poll item as a template context variable from our `main.views.user` function and let the Django templates do their magic.
2. Use Angular to call the `api/v1/polls` REST API and have Angular build the HTML template for User Polls.

Since this is a chapter on Angular and we have the proper endpoints setup, let's go with option 2. To do that, we need to talk about how to make REST calls with Angular.

Making REST calls with Angular

The simplest way to make a rest call in Angular is by using the `$http` service ([documentation](#)). `$http` is a core Angular service used to make Asynchronous Requests. If we wanted to call our polls API to get a poll via id, we can update our `UserPollCtrl.js` by adding the following code:

```
1 // Get the Poll
2 $scope.poll = ""
3
4 $http.get('/api/v1/polls/1').
5   success(function(data){
6     $scope.poll = data;
7   }).
8   error(function(data,status) {
9     console.log("calling /api/v1/polls/1 returned status " +
10       status);
11 });
```

- Line 2 - Initializes the `poll` scope variable as an empty string to avoid undefined errors later.
- Line 4 - `$http.get` makes a GET request to the specified URL.
- Line 5 - The return value from `$http.get` is a [promise](#). Promises allow us to register callback methods that receive the response object. Here, we are registering a method, `success`, to be called on a success. It receives the data object - e.g, the data returned in the response - and we set our `$scope.poll` variable to this, which should be the JSON representation of our poll, where `id` is 1.

Let's digress for a few minutes to talk about promises ...

A brief aside on `HttpPromise`

The `$http` service has several shortcut methods, corresponding to HTTP verbs, such as `get`, `put`, `post`, etc. When you call these methods they all return a promise object. A promise object is an object that represents a value that is not yet known because JavaScript executes asynchronously. While it takes time for the AJAX request to actually execute, your JavaScript code doesn't wait around for the result - it continues executing.

This is not what we want. What would happen if you have another function that's dependent on the results from the `$http` service and that function fires before the results are returned. That's a problem, and it's exactly what promises solve.

So a promise has states such as `waiting`, `success`, or `error`, which allows you to attach handlers (or methods that will be called) to each of those states. For example, `$http.get().success(function())` specifies the handler to be called when the `HttpPromise` returned by `$http.get()` reaches the success state. The same logic applies to an error handler.

The key advantage here is that we can write asynchronous code like synchronous code (which is much easier) and not have to worry too much about the asynchronous part.

The other important thing to understand about the `HttpPromise` is that it always passes the following parameters into your handler:

Parameter	Data Type	Explanation
<code>data</code>	String or Object	Response body
<code>status</code>	Number	HTTP status code - 200, 404, 500, etc.
<code>headers</code>	Function	Header getter function to be called to get the header item
<code>config</code>	Object	Config object used to generate the request
<code>statusText</code>	String	HTTP Status text of the response

NOTE: In JavaScript you can always just ignore extra parameters that you do not need like we have done in our success function above.

Back to the code

- Line 8 - This is the handler for the error state. In response to an error we just log the HTTP status code to the console.

So that little bit of code allows us to retrieve the information about a Poll to be displayed from our Django backend. Make sure to inject the `$http` service into the Angular Controller:

```
1 pollsApp.controller('UserPollCtrl', function($scope, $http)
```

Now we need to display that information. Let's modify our `djangular_polls/_polls.html` template:

```
1 <section class="info-box" id="polls" ng-app="pollsApp">
2 {% verbatim %}
3   <div ng-controller='UserPollCtrl'>
4     <h1>Poll: [[ poll.title ]]</h1>
5     <div ng-repeat="item in poll.items">
6       <span ng-click='vote(item)' class="glyphicon
7         glyphicon-plus"></span>
8       <strong>[[ item.text ]]</strong>
9       <span class="pull-right">[[ item.percentage | number:0
10        ]]</span>
11       <div class="progress">
12         <div class="progress-bar" role="progressbar"
13           aria-value="[[ item.percentage ]]"
14           aria-valuemin="0" aria-valuemax="100" style="width: [[
15             item.percentage ]]<div>
16       </div>
17     </div>
18   </div>
19 </section>
```

That's a whole lot less HTML!

This is mainly because of the `ng-repeat` on line 5 ([documentation](#)).

`ng-repeat` functions much like `{% for item in poll.items %}` would in a Django template. The main difference is that `ng-repeat` is generally used as an attribute of an HTML tag, and it repeats that tag.

In our case, the `<div>` is repeated, which includes all of the HTML inside the `<div>`. We only have to create the HTML for one of the voting items and use `ng-repeat` to generate a new item for every poll item returned from our `$http.get('/api/v1/polls/1')` call.

With `ng-repeat` we specify `item in poll.items`; this exposes `item` as a variable that we can use in Angular expressions:

- Line 6 - `ng-click='vote(item)'` passes in the current item to the vote function in our controller.
- Line 7 - We can also access properties of the `item` object like we do here with `[[item.text]]`.

The result of the `ng-repeat` function is to create all of the vote items with their corresponding progress bars. Then we can change the vote function in the controller to work with our polls model that we are getting from our JSON call. The new vote function looks like this:

```

1 $scope.total_votes = 0;
2
3 $scope.vote = function(item) {
4     item.votes = item.votes + 1;
5     $scope.total_votes = $scope.total_votes + 1;
6
7     for (i in $scope.poll.items){
8         var temp_item = $scope.poll.items[i];
9         temp_item.percentage = temp_item.votes / $scope.total_votes *
            100;
10    }
11 };

```

Update this in the `userPollCtrl.js` file.

The vote function is now also simpler than it was before, as we can rely on the post model holding all the information so we don't have to jump through the hoops that we did before.

Let's look at this a bit closer:

- Line 1 - Initialize `total_votes` to 0 on page load. We will need to change this eventually to support multiple users.
- Line 4 - Remember from the HTML above we are calling `vote(item)` where `item` is the poll item returned from our original query. Thus we can just add one to the item to account for the new vote.
- Line 5 - Increment the total number of votes so we can calculate percentages.
- Lines 7-10 - Loop through all `poll.items` and update the percentages. *Note: `poll.items` is an array. So `for (i in $scope.poll.items)` is similar to `for(var i = 0; i < $scope.poll.items.length; i++)`, just with a lot less code.*

Now we have a functioning user poll. Right? Not yet. We still need to add some data. We can do that directly from the browseable API.

Navigate to <http://localhost:8000/api/v1/polls/> in your browser and add the following JSON to the “Raw Data” form.

```
1 {
2   "title": "Who is the best Jedi?",
3   "publish_date": "2014-10-21T04:05:24.107Z",
4   "items": [
5     {
6       "id": 1,
7       "name": "Yoda",
8       "text": "Yoda",
9       "votes": 0,
10      "percentage": "0.00"
11    },
12    {
13      "id": 2,
14      "name": "Vader",
15      "text": "Vader",
16      "votes": 0,
17      "percentage": "0.00"
18    },
19    {
20      "id": 3,
21      "name": "Luke",
22      "text": "Luke",
23      "votes": 0,
24      "percentage": "0.00"
25    }
26  ]
27 }
```

Be sure to test it out! Navigate to <http://localhost:8000/>. After you login make sure the Poll is at the bottom right of the page. Test the functionality.

Refactoring - progress bars

One issue we have is that all the progress bars are the same color. Let's rectify that. To do so, we use a feature of `ng-repeat`. When you are looping through items in `ng-repeat`, `$index` represents the current iteration number - i.e., (0, 1, 2, etc.). So we can use that counter to pick a different class to control the color for each progress bar.

First, update `django_polls/_polls.html`:

```
1 <section class="info-box" id="polls" ng-app="pollsApp">
2   <div ng-controller='UserPollCtrl'>
3     <h1>Poll: [[ poll.title ]]</h1>
4     <div ng-repeat="item in poll.items">
5       <span ng-click='vote(item)' class="glyphicon
6         glyphicon-plus"></span>
7       <strong>[[ item.text ]]</strong>
8       <span class="pull-right">[[ item.percentage | number:0
9         ]]</span>
10      <div class="progress">
11        <div class="progress-bar [[ barcolor($index) ]]"
12          role="progressbar"
13          aria-value="[[ item.percentage ]]"
14          aria-valuemin="0" aria-valuemax="100" style="width: [[
15            item.percentage ]]</div>
16      </div>
17    </div>
18  </div>
19 </section>
```

So, we added in `[[ barcolor($index) ]]` as one of the classes. Now when Angular comes across this, it calls `$scope.barcolor()` from the controller and passes in the current iteration number of `ng-repeat`.

Let's update the controller as well in `static/js/userPollCtrl.js`:

```
1 $scope.barcolor = function(i) {
2   colors = ['progress-bar-success', 'progress-bar-info',
3     'progress-bar-warning', 'progress-bar-danger', '']
4   idx = i % colors.length;
5   return colors[idx];
6 }
```

This function uses a list of various Bootstrap classes to control the color of the progress bar. Using % (modulus) ensures that no matter how many vote items we get, we will always get an index that is in the bounds of our `colors` list.

Now all we need to do is make this thing work for multiple users.

Refactoring - multiple users

To do that, we need to keep track of `total_votes` and the votes per poll item on the backend so they can be shared across all users.

Let's first add the `total_votes` as an attribute of our `Poll()` class and make sure it is sent across in the `/api/v1/polls` GET request. To do this, we can add a `@property` to the `Poll()` class like so:

```
1 @property
2 def total_votes(self):
3     return self.poll_items().aggregate(
4         models.Sum('votes')).get('votes__sum', 0)
```

Before we jump into the explanation of what this code does let's talk about using a '@property' for just a second.

1. **What is a @property?** It's a function that you can call without (). So in our case, normally we would call 'model.total_votes()', but by making it a property with @property we can just call it like this - `model.total_votes`
2. **What is it good for?** Other than saving a couple of keystrokes, properties are best used for computed model attributes. You can think of it as a field within the model (sort of), except that it needs some logic to run on the get, set, or delete.
3. **Is it the same thing as a model field?** Absolutely not. Django, for the most part, just ignores a property - i.e., you can't use it in a `querySet` as it won't trigger a migration, etc.). You can think of a property like a field that you don't want django to know about.
4. **Why not just use a function?** Really @property is just syntactic sugar, so if you prefer you could use a function all you want. It's a design choice more than anything else.

Okay. On with the show ..

Take a look at the code again. This uses Django's `aggregate()` function to compute the sum of all the related `PollItem()` for the poll. `aggregate` returns a dictionary that in this case looks like `{ 'votes__sum': <sum_of_votes> }`. But we only want the actual number, so we just call a `get` on the returned dictionary and default the value to 0 in case it's empty. With this method, we calculate `total_votes` every time the property is called; that way we don't have to keep the various database tables in sync.

Because this is a property and not a "true" field, Django Rest Framework won't pick it up by default, so we need to modify our serializer as well:

```

1 class PollSerializer(serializers.ModelSerializer):
2     items = PollItemSerializer(many=True, required=False)
3     total_votes = serializers.Field(source='total_votes')
4
5     class Meta:
6         model = Poll
7         fields = ('title', 'publish_date', 'items', 'total_votes')

```

We instruct the serializer to include a field called `total_votes` which points to our `total_votes` property. With that, our `/api/v1/polls` will return an additional property - `total_votes`.

Finally, we need to calculate the percentage as well within `djangular_polls/models.py`. Let's use a similar technique to calculate the percentage on the fly:

```

1 @property
2 def percentage(self):
3     total = self.poll.total_votes
4     if total:
5         return self.votes / total * 100
6     return 0

```

We use the newly created `total_votes` property of our parent poll to calculate the percentage. We do need to be careful to avoid a divide by zero error. Then we make a similar change to the `PollItemSerializer` as we did to the `PollSerializer`. It now looks like this:

```

1 class PollItemSerializer(serializers.ModelSerializer):
2     percentage = serializers.Field(source='percentage')
3
4     class Meta:
5         model = PollItem
6         fields = ('id', 'poll', 'name', 'text', 'votes',
7                 'percentage')

```

Your `models.py` and the `serializers.py` should now look like this:

`djangular_polls/models.py`

```

1 from django.db import models
2 from django.db.models import Sum
3
4
5 class Poll(models.Model):

```

```

6     title = models.CharField(max_length=255)
7     publish_date = models.DateTimeField(auto_now=True)
8
9     @property
10    def total_votes(self):
11        return
12            self.poll_items().aggregate(Sum('votes')).get('votes__sum',
13                0)
14
15    def poll_items(self):
16        return self.items.all()
17
18    class PollItem(models.Model):
19
20        poll = models.ForeignKey(Poll, related_name='items')
21        name = models.CharField(max_length=30)
22        text = models.CharField(max_length=300)
23        votes = models.IntegerField(default=0)
24
25        @property
26        def percentage(self):
27            total = self.poll.total_votes
28            if total:
29                return self.votes / total * 100
30            return 0
31
32        class Meta:
33            ordering = ['-text']

```

djangular_polls/serializers.py

```

1 from rest_framework import serializers
2 from djangular_polls.models import Poll, PollItem
3
4
5 class PollItemSerializer(serializers.ModelSerializer):
6     percentage = serializers.Field(source='percentage')
7
8     class Meta:
9         model = PollItem

```

```

10         fields = ('id', 'poll', 'name', 'text', 'votes',
11                   'percentage')
12
13 class PollSerializer(serializers.ModelSerializer):
14     items = PollItemSerializer(many=True, required=False)
15     total_votes = serializers.Field(source='total_votes')
16
17     class Meta:
18         model = Poll
19         fields = ('id', 'title', 'publish_date', 'items',
20                   'total_votes')

```

With these changes we can now update our Angular controller to push the calculations to the Django backend so we can share the results amongst multiple users. But first we should update our database to reflect the changed table structure for the Poll / PollItems models, so don't forget to run makemigrations and migrate.

Once that is done we can switch back to our angular controller. Update the vote function within *static/js/userPollCtrl.js* like so:

```

1  $scope.vote = function(item) {
2      item.votes += 1;
3      $http.put('/api/v1/poll_items/'+item.id,item).
4      success(function(data){
5          $http.get('/api/v1/polls/1').success(function(data){
6              $scope.poll = data;
7          }).
8          error(function(data,status){
9              console.log("calling /api/v1/polls/1 returned status " +
10                 status);
11          });
12      }).
13      error(function(data,status){
14          console.log("calling PUT /api/v1/poll_items returned status " +
15             status);
16      });
17  });

```

Here, we're just calling the rest API a few times:

- **Line 2** - Update vote count for the item just voted on.

- **Line 3** - Using a PUT request, send the item back to the backend (which will update the vote count). Note the second parameter of the PUT call is the item itself, which will be converted to JSON and sent back to the server.
- **Line 5** - If the PUT request is successful then refetch all the Poll data, which will also give us updated `total_votes` and percentages for each item. It's important to only call this in the success handler so as to ensure the previous PUT request has completed, otherwise we might not get the most up-to-date data.

Using Angular Factories

We now have a multi-user poll working that is updating the backend. Our code works, but let's clean it up a bit.

One thing that the Angular philosophy stresses is separation of concerns. According to this philosophy, a controller should just be managing scope; it shouldn't be accessing external resources directly. This means we should not be calling `$http` from our controller. (This also makes testing the controller a whole lot easier.)

This is where factories come into play.

An Angular factory always returns an object, which generally has a number of functions attached that are responsible for working with data, handling business rules, or doing any number of tasks. Factories, unlike controllers, also maintain state throughout the lifetime of your Angular application.

For example, you might want to cache the poll id the first time a user calls `getPoll()` for use in other areas of our application. For our purposes, let's create a `PollFactory` to handle the REST API requests for us:

```
1 pollsApp.factory('pollFactory', function($http) {
2
3   var baseUrl = '/api/v1/';
4   var pollUrl = baseUrl + 'polls/';
5   var pollItemsUrl = baseUrl + 'poll_items/';
6
7   var pollFactory = {};
8
9   pollFactory.getPoll = function(id) {
10     return $http.get(pollUrl + id);
11   };
12
13   pollFactory.vote_for_item = function(poll_item) {
14     poll_item.votes +=1;
15     return $http.put(pollItemsUrl + poll_item.id, poll_item);
16   }
17
18   return pollFactory;
19 });
```

Let's go through the code line by line:

- **Line 1** - Here we just declare the factory. Note that we inject the `$http` service, since we will need it to make our REST calls.
- **Lines 3-5** - Define the URLs needed for the REST calls.
- **Line 7** - This is the factory object that we will eventually return, but first we need to add some functions to it.
- **Lines 9-11** - The first function we add is the `getPoll()` function, which will grab the poll from our Django REST API. Note that we are returning a promise (we talked about them previously) and not the actual data. The advantage of returning the promise is that it allows us to do promise chaining (one call after another), which we will make use of later in the controller.
- **Lines 13-16** - Our second function, `vote_for_item`, takes the poll item as an input, increments the vote counter, then updates the poll item through the REST API. Again we are returning a promise which wraps the actual result of the call.
- **Line 18** - Now that we have finished creating our `pollFactory()` object and have given it all the functionality we need. Let's return it, so we can use it in our controller.

Next, let's look at how we use our newly created factory in our controller.

All we have to do is ask Angular to inject the factory into our controller, which we can do with this initial line:

```
1 pollsApp.controller('UserPollCtrl',
2     function($scope, $http, pollFactory) {
```

This is the line used to create our controller. Notice that we added `pollFactory` to the list of dependencies, so Angular will inject the factory for us. Also note that we no longer require `$http`, as that is all handled by the factory now. With the declaration above, we can use the factory anywhere in our controller (or application, for that matter) by simply calling it like this:

```
1 pollFactory.getPoll(1);
```

Let's see how the complete controller looks after we add in the factory:

```
1 pollsApp.controller('UserPollCtrl',function($scope, pollFactory) {
2
3     //get the Poll
4     $scope.poll = ""
5     function setPoll(promise){
6         $scope.poll = promise.data;
7     }
8 }
```

```

9  function getPoll(){
10     return pollFactory.getPoll(1);
11 }
12
13 $scope.barcolor = function(i) {
14     var colors = ['progress-bar-success', 'progress-bar-info',
15                 'progress-bar-warning', 'progress-bar-danger', '']
16     var idx = i % colors.length;
17     return colors[idx];
18 }
19
20 getPoll().then(setPoll);
21
22 $scope.vote = function(item) {
23     pollFactory.vote_for_item(item)
24         .then(getPoll)
25         .then(setPoll);
26 }
27
28 });

```

You'll notice near the top of the controller we created two functions `getPoll` which asks the `pollFactory` to get the poll, and `setPoll` which takes a promise and uses it to set the value of `$scope.poll`. These functions will come in handy later on.

The next function `$scope.barcolor` remains unchanged.

- **Line 22** - `getPoll().then(setPoll)`. This is a simple example of promise chaining. First we call `getPoll` which in turn calls `pollFactory.getPoll` and returns a promise. Next we use a function of the promise called `then` which will be triggered once the promise has completed. We pass in our `setPoll()` function to `then`, which will receive the promise returned by `getPoll` and use the data on that promise to set our `$scope.poll` variable.
- **Line 25** - In our `$scope.vote` function we have another example of promise chaining. Here we first call `pollFactory.vote_for_item`, which issues a PUT request to our `poll_items` API and returns a promise containing the result of that PUT request. When the promise returns, we call our `getPoll()` function with `.then(getPoll)`. This function then gets the newer version of our Poll object with `total_votes` and percentages recalculated, and returns a promise. Finally we call `.then(setPoll)`, which uses that returned promise to update our `$scope.poll` variable.

As you can see, by structuring our factory methods to return promises, we can chain together several function calls and have them execute synchronously. As an added advantage, if any of the calls fail, the subsequent calls in the chain will receive the failed result and can thus react accordingly.

Conclusion

With that, we have some pretty well-factored Angular code that interacts with our RESTful API to provide a multi-user user poll function. While the Django Rest Framework part of this chapter should have been mostly a review, we did learn how to ensure that our URL structure plays nicely with Angular and how to serialize model properties that aren't stored in the database.

On the REST side of things, you should now have a better understanding of controllers and how they are used to manage scope, factories and how they are used to provide access to data, the `$http` service and how it is used to make AJAX calls, and of course promises and how they can be chained together to achieve some pretty cool functionality.

Also, let's not forget about the `ng-repeat` directive and how it can be used to achieve the same functionality as `{% for` in Django templates.

All in all, while this is only a small portion of what Angular offers, it should give you some good ideas about how it can be used to build well-factored, solid client-side functionality.

Exercises

1. For the first exercise let's explore storing state in our factory. Change the `pollFactory.getPoll` function to take no parameters and have it return the single most recent poll. Then cache that poll's id, so next time `getPoll()` is called you have the id of the poll to retrieve.
2. In this chapter we talk about factories but Angular also has a very similar concept called services. Have a read through Michael Herman's [excellent blog post on services](#) so you know when to use each.
3. Currently our application is a bit of a mismatch, we are using jQuery on some parts - i.e., showing achievements - and now Angular on the user polls. For practice, convert the showing achievement functionality, from *application.js*, into Angular code.

Chapter 15

Angular Forms

The final piece of our application that needs to be converted over to Angular is the registration form. This chapter will discuss how to handle forms in Angular when dealing with a Django back-end. We will address submitting and validating forms(client vs. server-side), dealing with CSRF, and basically making Django and Angular Forms work well together.

When creating a form with Django and Angular, you basically have two choices:

1. You can create the form with Angular along with a REST API for it to call on the Django backend.
2. You can also create the form from the Django side, using Django Forms, much like we have already done for the registration form. This allows you to build the form in the back-end and follow a more “traditional” Django development style.

We could argue at length as per which method is better. However, since we have talked about REST APIs and integrating those with Angular before, in this chapter let’s cover using Django Forms (the latter method). If nothing else, at least you’ll have an idea of how to implement both approaches and you can see which one works best for you.

Field Validation

Regardless of which choice we make for creating the form, we need some sort of form validation. Django Forms offer server side validation out of the box.

NOTE: Server side validation refers to validating the user input of a form on the server - e.g., via the Python code, in our case. Form submission and validation usually happen at the same time. If server side validation fails, a list of errors is generally sent back to the client. Meanwhile, **Client side validation** refers to validating user inputs directly in the web browser - e.g., via Angular, in our case.. Validation is often in response to a key press or an input field losing focus. This allows for fields to be validated one at a time, without a trip to the server, and generally provides a more “real-time” feel for the user, meaning that they are notified as soon as they get an error, instead of after the entire form is submitted.

Since we already know how to do server side validation with Django Forms, `form.is_valid()`, let’s look at client side validation with Angular. Before adding any validations, our registration form in `templates/payments/register.html` looks like this:

```
1 <form id="user_form" accept-charset="UTF-8" action="{% url
  'register' %}" class="form-stacked" method="post">
2   {% csrf_token %}
3   {% if form.is_bound and not form.is_valid %}
4     <div class="alert alert-{{error.tags}}"> <a class="close"
      data-dismiss="alert">×</a>
5     <div class="errors">
6       {% for field in form.visible_fields %}
7         {% for error in field.errors %}
8         <p>{{ field.label }}: {{ error }}</p>
9         {% endfor %}
10        {% endfor %}
11        {% for error in form.non_field_errors %}
12        <p>{{ error }}</p>
13        {% endfor %}
14      </div>
15    </div>
16    {% endif %}
17    {% for field in form.visible_fields %}
18    {% include "payments/_field.html" %}
```

```

19 {% endfor %}
20 <div id="change-card" class="clearfix"{% if not
    form.last_4_digits.value %} style="display: none"{% endif %}>
21   Card
22   <div class="input">
23     Using card ending with {{ form.last_4_digits.value }}
24     (<a href="#">change</a>)
25   </div>
26 </div>
27 {% include "payments/_cardform.html" %}
28 </form>

```

Let's parse it into sections:

- **Lines 4 - 16** - these lines handle displaying errors that are returned from the server side validation. We will just leave that as is for the time being.
- **Lines 17 - 19** - these lines handle displaying the fields that are defined in `payments.forms.UserForm`. It uses a template called `payments/_field.html` to render each of the fields. This is what we will first need to modify to do client side validation with Angular.
- **Remaining Lines** - these lines handle displaying the credit card information portion of the form, allowing users to input their credit card info. We have this separate form so we don't have to pass the actual credit card number to our back-end. Instead, once a user submits the form, we have some jQuery in `static/js/application.js` that will call the Stripe API and get a token that is sent to our back-end. This way we don't have to worry about storing people's credit cards and the security that comes along with that.

Coming back to lines 17-19, our user form fields are displayed using the `payments/_field.html` template, which looks like this:

```

1 <div class="clearfix">
2   {{ field.label_tag }}
3   <div class="input">
4     {{ field }}
5   </div>
6 </div>

```

This produces very simple HTML for each field:

```

1 <div class="clearfix">
2   <label for="id_name">Name:</label>

```

```

3 <div class="input">
4   <input id="id_name" name="name" type="text">
5 </div>
6 </div>

```

In order to do Angular validation, we need to first add an `ng-model` to each field in our form. In our template we are using `{{field}}` to render the input fields. Let's think about what is happening with that.

`field` is a property of the form. If you recall, we called `{% for field in form.visible_fields %}`. This will loop through all the fields in our `UserForm` and display them. But how exactly does it know what to display? Looking at our `payments.forms.UserForm` class, we see:

```

1 class UserForm(CardForm):
2     name = forms.CharField(required=True)
3     email = forms.EmailField(required=True)
4     password = forms.CharField(
5         required=True,
6         label=('Password'),
7         widget=forms.PasswordInput(render_value=False)
8     )
9     ver_password = forms.CharField(
10        required=True,
11        label=('Verify Password'),
12        widget=forms.PasswordInput(render_value=False)
13    )

```

This is a list of all the fields. Pay attention to the `password` and `ver_password` fields. They both have an optional argument `widget`. In Django, a widget controls how a field is rendered to HTML. Every field gets a default widget based upon the field type. So `CharField` will have a default `TextInput` widget, `EmailField` will have a default `EmailInput` widget, and so on. For our password fields, we have overwritten the default widget to be a `PasswordInput` widget so that the password isn't display when the user types. This is handled by simply adding the HTML attribute `type='password'` to the input when it is rendered, and the widget does that.

Since the widget ultimately controls how the field is rendered and we want all of our fields to be rendered with an `ng-model` attribute, we can change the widget and tell it to include the `ng-model` attribute for us.

The easiest way to do that is to modify the `__init__` function of our `UserForm`. Behold:

```

1 class UserForm(CardForm):

```

```

2
3     ng_scope_prefix = "userform"
4
5     def __init__(self, *args, **kwargs):
6         super(UserForm, self).__init__(*args, **kwargs)
7         for name, field in self.fields.items():
8             attrs = {"ng-model": "%s.%s" % (self.ng_scope_prefix,
9                                     name)}
10
11             field.widget.attrs.update(attrs)

```

- **Line 3** - In Angular, it's simpler to nest all of our models in one object. In other words, rather than having models named email, name, etc... it's easier if we have userform.email, and userform.name. This way, when it comes time to pass that information around in Angular (i.e., when we are POSTing our form), we can just pass userform and not have to reference each individual field. This also has the added advantage of allowing us to change the names and even the number of fields in the form without having to make corresponding changes in Angular. So grouping together all of the \$scope for a form is a good thing in Angular. And that is what the ng_scope_prefix field is for.
- **Line 5** - Our __init__ function.
- **Line 6** - Always a good idea to call the __init__ of our parent class in case something is happening there.
- **Line 7** - Loop through each field in UserForm.
- **Line 8** - Each widget uses a dictionary called attrs that stores the HTML attributes that will be rendered when the widget is rendered. Here we are creating a dictionary with one entry that has the key of ng-model and the value of userform.<fieldName>.
- **Line 10** - Add our dictionary to the list of attrs the widget will render.

The updated payments.forms.UserForm class now looks like:

```

1 class UserForm(CardForm):
2
3     name = forms.CharField(required=True)
4     email = forms.EmailField(required=True)
5     password = forms.CharField(
6         required=True,
7         label=('Password'),

```

```

8         widget=forms.PasswordInput(render_value=False)
9     )
10    ver_password = forms.CharField(
11        required=True,
12        label=('Verify Password'),
13        widget=forms.PasswordInput(render_value=False)
14    )
15
16    ng_scope_prefix = "userform"
17
18    def __init__(self, *args, **kwargs):
19        super(UserForm, self).__init__(*args, **kwargs)
20        for name, field in self.fields.items():
21            attrs = {"ng-model": "%s.%s" % (self.ng_scope_prefix,
22                                           name)}
23
24            field.widget.attrs.update(attrs)
25
26    def clean(self):
27        cleaned_data = self.cleaned_data
28        password = cleaned_data.get('password')
29        ver_password = cleaned_data.get('ver_password')
30        if password != ver_password:
31            raise forms.ValidationError('Passwords do not match')
32        return cleaned_data

```

With this change, if we refresh the browser and look at the fields rendered on the registration page (<http://localhost:8000/register>, they should look like:

```

1 <div class="clearfix">
2     <label for="id_name">Name:</label>
3     <div class="input">
4         <input id="id_name" name="name" ng-model="userform.name"
5             type="text" class="ng-pristine ng-valid">
6     </div>
7 </div>

```

There you go. We have our nice `ng-model="userform.name"` added to our input. Pretty cool, huh? But wait, there is also something else added: `class="ng-pristine ng-valid"`. We didn't tell it to do that, did we?

Well no, not directly. Angular does that for us: Anytime you add an `ng-model` to an input element, Angular will add those classes, and that's a good thing because we need those classes for data/field validation. Let's look at what classes there are that Angular sets for us to help with the validation. The four we care about are:

Class	Activated...
<code>ng-valid</code>	When the field passes validation
<code>ng-invalid</code>	When the field fails validation - i.e., an empty field with a "required" validation
<code>ng-pristine</code>	Before any interaction on the field - i.e. before you type in the input field
<code>ng-dirty</code>	After the field has been changed - i.e., after the first keystroke in the input field

Notice that by default the field is assigned `ng-pristine`, meaning it hasn't been touched yet, and `ng-valid`, meaning it passes validation.

Try this: Open your JavaScript console and highlight the name input field. Then click on the name field and type something. Watch the classes change to:

```
1 ng-valid ng-dirty
```

This signifies that text has been inputted into the field and it passes validation. Let's look at the email field as an example. Type a valid email, then check out the JavaScript console:

```
1 <input id="id_email" name="email" ng-model="userform.email"
  type="email" class="ng-dirty ng-valid ng-valid-email">
```

You'll notice it has an extra class, `ng-valid-email`, because a field can have multiple validators. This class tells us that the email-validator is currently valid. But how does Angular know this is an email field? Angular uses the HTML5 attributes to determine which validators a field needs and applies them automatically to the field (as long as it has the `ng-model` directive attached). In this case, because the field has the attribute `type="email"`, Angular applies the validator. Angular respects the HTML5 type values as well as the "required" attribute for form validation. You can also use:

- `ng-required` - just sets the `required` attribute to true
- `ng-minlength` or `min`
- `ng-maxlength` or `max`
- `ng-pattern` - validates based on a regex pattern

With these you have quite a bit of flexibility and can validate almost anything you want. (It is also possible to use custom directives to do custom validation.)

The only issue we have now is that in our `UserForm` we have also specified some additional validations, like:

```
1 name = forms.CharField(required = True)
```

And we would like those validations to also be on the front-end. So we can go back to our `payments.forms.UserForm.__init__` function and add the appropriate attributes. As an example, if we wanted to add the required HTML attribute for any field that had `required = True` we could add the following code:

```
1 if field.required:
2     attrs.update({"required": True})
3 if field.min_length:
4     attrs.update({"ng-minlength": field.min_length})
```

Since we don't have any `min_length` validations, let's go ahead and change our name field to require a minimum of three letters:

```
1 name = forms.CharField(required=True, min_length=3)
```

You could of course add as many parameters as you wanted. You have total control over what is rendered.

The `__init__` method should now look like:

```
1 def __init__(self, *args, ** kwargs):
2     super(UserForm, self).__init__(*args, **kwargs)
3     for name, field in self.fields.items():
4         attrs = {"ng-model": "%s.%s" % (self.ng_scope_prefix, name)}
5
6         if field.required:
7             attrs.update({"required": True})
8         if field.min_length:
9             attrs.update({"ng-minlength": field.min_length})
10    field.widget.attrs.update(attrs)
```

Display Error Messages

Now that we have a way to perform validation, we need to alert the user when something is invalid. The simplest way is to add some custom styling to the Angular classes - `ng-valid`, `ng-invalid`, `ng-dirty`. That way whenever the validation changes it triggers the CSS, alerting the end user. Add the following two lines to the `static/css/mec.css` file:

```
1 input.ng-dirty.ng-valid { border:1px solid Green; }
2 input.ng-dirty.ng-invalid { border:1px solid Red; }
```

This will color the input green if it is valid and it's been changed, and likewise will color the input red if it's invalid and has been changed. Go ahead and try it out. Navigate to <http://localhost:8000/register>. If you type `py` in the email field and then move to a new field, it should become outlined in red, because that's not a valid email. Try changing it to `py@gmail.com`. It should now be green.

This isn't super helpful to the user though, because they don't know why the field is invalid. So let's tell them. Angular to the rescue again!

Angular keeps track of validation errors in an `$error` object. You can access it through the form, with the only requirement being that the form has a `name` attribute. Currently our registration form has no name, so assuming we add the name `user_form`, you could check for errors with this code:

```
1 user_form.name.$error.required
```

Or, the more general form:

```
1 <<form_name>>.<<field_name>>.$error.<<validator-name>>
```

This value will be set to `True` if the field is failing the particular validator. Now you could create your own custom error and show it only when the validator is true. For example:

```
1 <span ng-show="user_form.name.$error.required">This field is
   required</span>
```

Here `ng-show` only display the span if the name field is failing the required validation. Of course we don't want to have to write this kind of stuff over and over for each field, so let's revisit our `templates/payments/_field.html` template and modify it to include some custom errors for each field:

```
1 <div class="clearfix">
2   {{ field.label_tag }}
3   <div class="input">
```



```

4     {{ field }}
5 </div>
6 <div class="custom-error"
7     ng-show="{{form.form_name}}.{{field.name}}.$dirty &&
      {{form.form_name}}.{{field.name}}.$invalid">
8     {{field.label}} is invalid:
9     <span
10         ng-show="{{form.form_name}}.{{field.name}}.$error.required">value
        is required.</span>
10    <span
11        ng-show="{{form.form_name}}.{{field.name}}.$error.email">Input
        a valid email address.</span>
11    </div>
12 </div>

```

You can see on line 6 that we are adding a div below each field. That div will show only when the field is in error - e.g., when the field is `$dirty` (which corresponds to the `ng-dirty` class) and `$invalid` (which corresponds to the `ng-invalid` class). Then inside of the div we check each of the error types - required and email - and show an error message for each of those errors.

You'll notice that we are using the template variable `{{form.form_name}}`; we need to add this to our UserForm in `payments.forms.UserForm`:

```

1 form_name = 'user_form'

```

So the class now looks like:

```

1 class UserForm(CardForm):
2
3     name = forms.CharField(required=True, min_length=3)
4     email = forms.EmailField(required=True)
5     password = forms.CharField(
6         required=True,
7         label=('Password'),
8         widget=forms.PasswordInput(render_value=False)
9     )
10    ver_password = forms.CharField(
11        required=True,
12        label=('Verify Password'),
13        widget=forms.PasswordInput(render_value=False)
14    )

```

```

15
16 form_name = 'user_form'
17 ng_scope_prefix = "userform"
18
19 def __init__(self, *args, ** kwargs):
20     super(UserForm, self).__init__(*args, **kwargs)
21     for name, field in self.fields.items():
22         attrs = {"ng-model": "%s.%s" % (self.ng_scope_prefix,
23                                     name)}
24
25         if field.required:
26             attrs.update({"required": True})
27         if field.min_length:
28             attrs.update({"ng-minlength": field.min_length})
29         field.widget.attrs.update(attrs)
30
31 def clean(self):
32     cleaned_data = self.cleaned_data
33     password = cleaned_data.get('password')
34     ver_password = cleaned_data.get('ver_password')
35     if password != ver_password:
36         raise forms.ValidationError('Passwords do not match')
37     return cleaned_data

```

And also in the templates/payments/register.html, let's give the form the same name:

```

1 <form id="{{form.form_name}}" name="{{form.form_name}}"
2     accept-charset="UTF-8" action="{% url 'register' %}"
3     class="form-signin" role="form" method="post" novalidate>

```

In addition to changing the name of the form you may have noticed the last attribute, novalidate. This turns off the native HTML5 validation, because we are going to have Angular handle the validation for us. Finally, let's add a CSS class to style the error message a bit in static/css/mec.css:

```

1 .custom-error {
2     color: #FF0000;
3     font-family: sans-serif;
4 }

```

Now try it out. You'll notice that as you type into the field, an error message will be displayed until you type the correct info to pass the validation, then once you pass the validation, the

error message will disappear and the field will become green. How's that for rapid feedback!

Next we need to apply validation to the credit card fields in `templates/payments/_cardform.html`, which we'll leave as an exercise for you.

As a final touch you may want to disable the register button until all the fields pass validation. We talked about how Angular assigns classes such as `ng-valid` and `ng-invalid` to fields that fail validation. It also assigns those same classes to the form, so you can quickly tell if the entire form is valid or invalid. So using that, we can modify our register button (which is in `templates/payments/_cardform.html`) to look like this:

```
1 <input class="btn btn-lg btn-primary" id="user_submit"
    name="commit" type="submit" value="Register"
    ng-disabled="{{form.form_name}}.$pristine ||
    {{form.form_name}}.$invalid">
```

So, `ng-disabled` will ensure the button is disabled if the form is `$pristine` (meaning no data has been input on the form yet) or if the form is `$invalid` (meaning one of the field validators is failing). So once all data is input correctly then the button will become valid and you can submit the form.

If you do submit the form, the register view function will pick it up and perform server side validation. If there is an error, such as the email is already in use, it will send back the error which will be displayed on top of the form.

Form submission with Angular

With the form validation out of the way, let's now look at how we handle form submissions with Angular. First, we need to add a controller, and then we can have the form call a function within the controller to handle the actual submission in `templates/payments/register.html`:

```
1 <section class="container" ng-controller="RegisterCtrl">
2   <form name="{{form.form_name}}" accept-charset="UTF-8" novalidate
      class="form-signin" role="form" ng-submit="register()">
```

Here we first declare a controller called `RegisterCtrl` (we'll add a file called `static/js/registerCtrl.js` for that). Then notice how we have removed the `id` attribute (`id="{{form.form_name}}"`) that was being used for the jQuery function that calls Stripe on submit, but we are going to convert that to Angular so we don't need the `id` anymore. Finally, `ng-submit="register()"` is called to handle the form submission, so we will need to create the `register()` function in our controller. Also note that we have removed the `action` attribute. If we leave the `action` attribute in place, the form will submit as it normally would, skipping the Angular `ng-submit` handling of form submission.

The rest of the template remains unchanged. Now let's look at our `register()` function to see what happens when the form is submitted. First off, let's remove the jQuery call to Stripe in our `static/js/application.js` file and implement it in Angular. Before we get to the JavaScript portion, though, let's update our `templates/payments/_cardform.html` template with some Angular goodness so it will work better with our Angular call:

```
1 <!--
2 <input type="hidden" name="last_4_digits" id="last_4_digits">
3 <input type="hidden" name="stripe_token" id="stripe_token">
4 -->
5
6 <div id="credit-card">
7   <div id="credit-card-errors" ng-show="stripe_errormsg">
8     <div class="custom-error" id="stripe-error-message">
9       [[ stripe_errormsg ]]</div>
10  </div>
11
12  <div class="clearfix">
13    <label for="credit_card_number">Credit card number</label>
14    <div class="input">
15      <input class="field" id="credit_card_number" type="text"
16        ng-model="card.number" required>
```

```

17     </div>
18 </div>
19 <div class="clearfix">
20   <label for="cvc">Security code (CVC)</label>
21   <div class="input">
22     <input class="small" id="cvc" type="text" ng-model="card.cvc"
23       required min=3>
24   </div>
25 </div>
26 <div class="clearfix">
27   <label for="expiry_date">Expiration date</label>
28   <div class="input">
29     <select class="small" id="expiry_month"
30       ng-model="card.expMonth">
31       {% for month in months %}
32       <option value="{{ month }}" {% if soon.month == month %}
33         selected{% endif %}>{{ month }}</option>
34       {% endfor %}
35     </select>
36     <select class="small" id="expiry_year" ng-model="card.expYear">
37       {% for year in years %}
38       <option value="{{ year }}" {% if soon.year == year %}
39         selected{% endif %}>{{ year }}</option>
40       {% endfor %}
41     </select>
42   </div>
43 </div>
44 <br/>
45 </div>
46 <div class="actions">
47   <input class="btn btn-lg btn-primary" id="user_submit"
48     name="commit" type="submit" value="Register"
49     ng-disabled="{{form.form_name}}.$pristine ||
50       {{form.form_name}}.$invalid">
51 </div>

```

Mainly we have just added the `ng-model` directive to the various form fields. Also, on lines 7-9 we are using the model `stripe_errormsg` to determine if there is an error and to display any error messages returned from Stripe.

Also notice the first two hidden fields that are commented out. We do not need those anymore because we can just store the value in an Angular model without passing them to the template.

Now let's go back to the `register` function in our controller. A straight-forward attempt might look like this:

```
1 mecApp.controller('RegisterCtrl', function($scope, $http) {
2
3     $scope.register = function() {
4         $scope.stripe_errormsg = "";
5         approve_cc();
6     };
7
8     approve_cc = function() {
9         Stripe.createToken($scope.card, function(status, response) {
10             if (response.error) {
11                 $scope.$apply(function(){
12                     $scope.stripe_errormsg = response.error.message;
13                 });
14             } else {
15                 $scope.userform.last_4_digits = response.card.last4;
16                 $scope.userform.stripe_token = response.id;
17             }
18         });
19     };
20 });
```

Save this in a new file called `static/js/registerCtrl.js`. Make sure to update the base template:

```
1 <script src="{% static 'js/registerCtrl.js' %}"
   type="text/javascript"></script>
```

- **Line 3** - Our `register` function, which is called on form submit.
- **Line 4** - This clears out any errors messages that may be displayed on the screen.
- **Line 5** - Call the `approve_cc` function, which handles the Stripe processing.
- **Line 8** - The `approve_cc` function calls `Stripe.createToken` in the same way we did with jQuery. And with the result we set the appropriate models, storing the card's last 4 digits and the Stripe token. If the Stripe call fails, however, we will just store the error message.

- **Line 10-12** - when we call `Stripe.createToken` and then use a callback to handle the return value (as we are doing in this example), we are outside of Angular's "digest loop", which means we need to tell Angular that we have made changes to the model. This is what `$scope.$apply` does. It let's Angular know that we have made some changes so Angular can apply them. Most of the time this isn't necessary because Angular will automatically wrap our calls in a `$scope.$apply` for us (as is the case when we use `$http`), but because Angular isn't aware of Stripe, we need this extra step to ensure our changes are applied.

While the above code works, it uses jQuery-style callbacks as opposed to Angular-style promises. This will make it difficult, or at least a little bit messy, for us to chain together the functionality that we will need, which will be:

call Stripe -> post data to Django -> log user in -> redirect to members' page -> handle errors if they occur

Let's rework the Stripe function call to use promises. Also, let's factor it out into a factory to keep our controller nice and tidy. The factory will look like this:

```

1  mecApp.factory("StripeFactory", function($q, $rootScope) {
2
3      var factory = {}
4      factory.createToken = function(card) {
5          var deferred = $q.defer();
6
7          Stripe.createToken(card, function(status, response) {
8              $rootScope.$apply(function() {
9                  if (response.error) return
10                     deferred.reject(response.error);
11                 return deferred.resolve(response);
12             });
13         });
14
15         return deferred.promise;
16     }
17     return factory;
18 });

```

Notice how the service uses the `$q` API to wrap the `Stripe.createToken` call in a promise. In particular:

- **Line 5** - `$q.defer()` will create the promise object.
- **Line 9** - If `Stripe.createToken` returns errors then resolve the promise as rejected (think of it like “raise an exception” in Python)
- **Line 10** - If `Stripe.createToken` returns without error then resolve the promise (e.g., return with the return value from `Stripe.createToken`)

Also note because the promise is operating outside of Angular’s normal digest loop, we wrapped the promise resolution in `$rootScope.$apply`. This should ensure that the models are updated properly when handling promise resolution.

NOTE: We used `$rootScope` and not `$scope` because this factory isn’t tied to any particular controller and thus doesn’t have access to `$scope`. In other words, the consumer of this factory method doesn’t need to worry about calling `$apply`.

Then to call the service:

```

1  mecApp.controller('RegisterCtrl', function($scope, StripeFactory) {
2
3      var setToken = function(data) {
4          $scope.userform.last_4_digits = data.card.last4;
5          $scope.userform.stripe_token = data.id;
6      }
7
8      logStripeErrors = function(error) {
9          $scope.stripe_errormsg = error.message;
10     }
11
12     $scope.register = function() {
13         $scope.stripe_errormsg = "";
14         StripeFactory.createToken($scope.card)
15             .then(setToken, logStripeErrors);
16     };
17
18 });

```

This code should now start to look similar to our `UserPollCtrl` because we are relying on promises and an external factory to get the work done. With that we have now replaced our jQuery Stripe call with an Angular one. In doing so we have changed our form to an Angular-controlled form, so let’s now look at submitting forms with Angular.

CSRF Protection

The first item that needs to be addressed when submitting forms from Angular to Django is the CSRF protection. If we were to just submit our form as it is, Django's CSRF protection would complain. In order to rectify that, we can add a configuration setting to our Angular module in `static/js/application.js`:

```
1 meApp.config(function($interpolateProvider, $httpProvider) {
2   $interpolateProvider.startSymbol('[[')
3   .endSymbol(']]');
4   $httpProvider.defaults.headers.common['X-CSRFToken'] =
      $('input[name=csrfmiddlewaretoken]').val();
5 });
```

- **Line 4** - is the newly added line here, and it says: add the X-CSRFToken value to the headers of all requests that we send, and set that value to the input named `csrfmiddlewaretoken`. Luckily for us, that is also the name of the input that is created with Django's `{% csrf_protection %}` tag.

CSRF - Sorted.

Next up is making the actual post request from our controller, which would look like:

```
1 $http.post("/register", $scope.userForm)
```

There are two issues with this:

1. Angular sends JSON data, while our `register` view currently expects "url encoded" data. If we want to send URL encoded data in Angular, we would have to encode the data and then add the proper Content-Type header to the post. Alternatively, we could change our Django form to handle the incoming JSON.
2. `$http` makes AJAX requests, and our `register` view is expecting a standard request, so we will have to change what it returns and how it returns it.

In fact, since we are going to send JSON asynchronously, we should probably be thinking about extending our JSON API to include a user resource; in which case, form submissions would create a new user by POSTing to the user resource and then, if successful, log the user in and redirect to the user page. That is a pretty big step from where we are now, so let's look at just changing our existing `register` view function to handle JSON (and we will leave the more proper REST implementation as an exercise for the user).

How I taught an old view some new JSON tricks.

Accepting JSON in our view looks like this (don't make any changes yet, though):

```
1 import json
2
3 if request.method == 'POST':
4     # We only talk AJAX posts now
5     if not request.is_ajax():
6         return HttpResponseBadRequest("I only speak AJAX nowadays")
7
8     data = json.loads(request.body.decode())
9     form = UserForm(data)
10
11     if form.is_valid():
12         #carry on as usual
```

- **Line 1** - We are going to need JSON; import it at the top of the file.
- **Line 3** - We are going to leave the GET request unchanged, but for POST...
- **Line 5** - This line checks the header to see if the request looks like it's an AJAX request. In other words, if the header has X-Requested-With: XMLHttpRequest, this check is going to pass.
- **Line 6** - Insert attitude (err, an error)
- **Line 8** - JSON data will be sent in the body. Remember, though, we are dealing with Python 3 now, and so `request.body` will come back as a byte stream. `json.loads` wants a string, so we use `.decode()` for conversion.
- **Line 9** - From here we are off to the races; we can load up our form with the incoming data and use the Django form the same way we always have.

Providing JSON Responses

Despite all the attitude (the error), our `register` view function is not very smart right now because it accepts an AJAX request and returns a whole truck load of HTML when it probably should be just returning status as JSON. If you submit the form as is now, assuming the data is correct, it will successfully register the user and then just return the `user.html` page. But since this is an AJAX request, our front-end will never redirect to the user page. Let's update the `register` view function in `payments.views.register` to just report status, and then we can do redirection from Angular.

```

1 def register(request):
2     user = None
3     if request.method == 'POST':
4         # We only talk AJAX posts now
5         if not request.is_ajax():
6             return HttpResponseBadRequest("I only speak AJAX
              nowadays")
7
8         data = json.loads(request.body.decode())
9         form = UserForm(data)
10
11        if form.is_valid():
12            try:
13                customer = Customer.create(
14                    "subscription",
15                    email=form.cleaned_data['email'],
16                    description=form.cleaned_data['name'],
17                    card=form.cleaned_data['stripe_token'],
18                    plan="gold",
19                )
20            except Exception as exp:
21                form.addError(exp)
22
23            cd = form.cleaned_data
24            try:
25                with transaction.atomic():
26                    user = User.create(cd['name'], cd['email'],
27                                     cd['password'],
28                                     cd['last_4_digits'],
29                                     stripe_id="")
30
31                    if customer:
32                        user.stripe_id = customer.id
33                        user.save()
34                    else:
35                        UnpaidUsers(email=cd['email']).save()
36
37            except IntegrityError:
38                resp = json.dumps({"status": "fail", "errors":

```

```

37         cd['email'] + ' is already a
           member'})
38     else:
39         request.session['user'] = user.pk
40         resp = json.dumps({"status": "ok", "url": '/'})
41
42         return HttpResponse(resp,
43                               content_type="application/json")
44     else: # form not valid
45         resp = json.dumps({"status": "form-invalid", "errors":
46                             form.errors})
47         return HttpResponse(resp,
48                               content_type="application/json")
49
50     else:
51         form = UserForm()
52
53     return render_to_response(
54         'payments/register.html',
55         {
56             'form': form,
57             'months': list(range(1, 12)),
58             'publishable': settings.STRIPE_PUBLISHABLE,
59             'soon': soon(),
60             'user': user,
61             'years': list(range(2011, 2036)),
62         },
63         context_instance=RequestContext(request)
64     )

```

The top part of the function is the same as what we covered earlier in the chapter. In fact the only difference in the function is what is returned for a POST request:

- **Line 38-39** - If the registration is successful, rather than redirecting the user we now return some JSON saying the status is OK and providing a URL to redirect to.
- **Line 35-36** - If there is an error when registering the user (such as email address already exists) then return some JSON saying status it failed and the associated errors.
- **Line 44-45** - If form validation fails, return the failed result as JSON. You might think, why do we still need to validate on the back-end if we have done it on the front-end? Since we are decoupled from the front-end now, we can't be 100% sure the front-end

validation is working correctly or even ran, so it's best to check on the back-end as well just to be safe.

After updating the `register` view function to return some JSON, let's update the front-end to take advantage of that. First thing, let's create another factory to interact with our Django back-end.

Fixing Tests

But first, let's keep our unit tests working. Since we switched to returning json responses, we will need to update several of the tests in `tests/payments/testViews.py`. Let's look at the `RegisterPageTests` class.

Our register view function now handles both GET and POST requests, so the first thing to do is change our `setUp` function to create the two request that we need. Doing so will make `RegisterPageTests.setUp` look like:

```
1  def setUp(self):
2      self.request_factory = RequestFactory()
3      data = json.dumps({
4          'email': 'python@rocks.com',
5          'name': 'pyRock',
6          'stripe_token': '...',
7          'last_4_digits': '4242',
8          'password': 'bad_password',
9          'ver_password': 'bad_password',
10     })
11     self.post_request = self.request_factory.post(self.url,
12                                                    data=data,
13                                                    content_type="application/json",
14                                                    HTTP_X_REQUESTED_WITH='XMLHttpRequest')
15     self.post_request.session = {}
16     self.request = self.request_factory.get(self.url)
```

Above we have created two requests -

1. The standard GET request, which is `self.request`. That remains unchanged.
2. And a new POST request called `post_request` that issues a POST with the proper json encoded data `data`.

Also note that `post_request` sets the extra parameter `HTTP_X_REQUESTED_WITH='XMLHttpRequest'`. This will make `request.is_ajax()` return `TRUE`.

From there its just a matter of changing most of the tests to use the `post_request` and to ensure the data returned is the appropriate JSON response. For example `RegisterPageTests.test_invalid_form_returns_registration_page` will now look like:

```

1 def test_invalid_form_returns_registration_page(self):
2
3     with mock.patch('payments.forms.UserForm.is_valid') as
         user_mock:
4
5         user_mock.return_value = False
6         self.post_request._data = b'{}'
7         resp = register(self.post_request)
8
9         #should return a list of errors
10        self.assertContains(resp, '"status": "form-invalid"')
11
12        # make sure that we did indeed call our is_valid function
13        self.assertEqual(user_mock.call_count, 1)

```

The important lines are:

- **Line 6** - set the JSON data to null to ensure the user form is invalid.
- **Line 10** - Ensure the JSON Response contains a form-invalid status.

Similar changes need to be made through the remainder of the tests in RegisterPageTests. Go ahead and try to update the tests yourself. If you get stuck you can grab the updated file from the Github repo:

[test/payments/testViews.py](#)

Now back to the front-end javascript code to support the change to our view code.

Breaking Chains

Add the following to `static/js/registerCtrl.js`:

```
1 mecApp.factory("UserFactory", function($http) {
2   var factory = {}
3   factory.register = function(user_data) {
4     return $http.post("/register",
5       user_data).then(function(response)
6       {
7         return response.data;
8       });
9   }
10  return factory;
11 });
```

Here we are taking care of POSTing the data to our back-end and returning the response. Then we just have our controller call the factory, assuming of course our Stripe call passed. We do this with a promise chain.

Let's first have a look at the chain:

```
1 $scope.register = function() {
2   $scope.stripe_errormsg = "";
3   $scope.register_errors = "";
4
5   StripeFactory.createToken($scope.card)
6     .then(setToken, logStripeErrors)
7     .then(UserFactory.register)
8     .then(redirect_to_user_page)
9     .then(null, logRegisterErrors);
10  };
```

Above is our register function (which if you recall is tied to our form submission with `ng-submit`). The first thing it does is clear out any errors that we may have. Then it calls a promise chain starting on line 5. Let's go through each line in the promise chain.

- `StripeFactory.createToken($scope.card)` - Same as before: Call Stripe, passing in the credit card info.
- `.then(setToken, logStripeErrors)` - This line is called after the `createToken` call completes. If `createToken` completes successfully then `setToken` is called; if it fails then `logStripeErrors` is called. `setToken` is listed below:


```

1 var setToken = function(data) {
2   $scope.userform.last_4_digits = data.card.last4;
3   $scope.userform.stripe_token = data.id;
4   return $scope.userform;
5 }

```

Same as before: We appropriate scope data. The final step in this function is to return `$scope.userform`. Why? Because we are going to need it in the next step, so we are just passing on the data to the next promise in the chain.

Moving on to the `logStripeErrors` function:

```

1 var logStripeErrors = function(error) {
2   $scope.stripe_errormsg = error.message;
3   throw ["There was an error processing the credit card"];
4 }

```

error is stored in the `stripe_errormsg` (same as before). But now we are throwing an error! This has to do with how promise chaining works. If an error is thrown in a promise chain, it will be converted to a rejected promise and passed on to the next step in the chain. If the next step doesn't have an `onRejected` handler then that step will be skipped and the promise will be re-thrown, again and again, until there is no more chain or until there is an `onRejected` handler. So looking at our promise chain again:

```

1 StripeFactory.createToken($scope.card)
2   .then(setToken, logStripeErrors)
3   .then(UserFactory.register)
4   .then(redirect_to_user_page)
5   .then(null, logRegisterErrors);

```

If `createToken` returns a rejected promise, it will be handled by `logStripeErrors`. If `logStripeErrors` were to store the error message and do nothing else, it would by default return a fulfilled promise with a value of `null`. This is like saying `deferred.resolve(null)`, which means that the next step in the promise chain, `submitForm`, would be called, and then our form would be submitted even though there was an error on the Stripe side. Not good.

So we throw the error, which is in effect the same as saying `deferred.reject('error msg')` because it will be wrapped for us automatically. Since the line `.then(submitForm)` has no reject handler, the error will be passed to the next line `.then(redirect_to_user_page)`, which also has no `onRejected` promise handler - and so it will be passed on again. Finally the line `.then(null, logRegisterErrors)` has an `onRejected` handler, and so it will receive the value that was thrown from our `logStripeErrors` function and process it.

This makes it possible for us to break out of the chain by throwing an error and never catching it or trying to recover from an error and continuing execution along the chain.

The next step in the chain is the good ol' `UserFactory.register`, which does the POST and returns the response, which is handled by `redirect_to_user_page`:

```
1 var redirect_to_user_page = function(response) {
2   if (response.errors) {
3     throw response.errors;
4   } else {
5     window.location = response.url
6   }
7 }
```

Here we receive the JSON response, and if it has an `errors` key, we throw the errors (so they will be handled by the next `onRejected` handler). If no errors then redirect (with `window.location`, old-school style) to the URL that should have been returned by our POST call.

On a successful registration, this will be the end of our promise chain. But if we have an error, then there is one more `onRejected` handler that will get triggered: `logRegisterErrors`.

```
1 var logRegisterErrors = function(errors) {
2   $scope.register_errors = errors;
3 }
```

Just set them to the appropriate scope value. These errors will be displayed at the top of our form. So if we have a look at `/templates/payments/register.html` then we can see our old error displaying functionality.

This old version-

```
1 {% if form.is_bound and not form.is_valid %}
2   <div class="alert alert-{{error.tags}}"> <a class="close"
3     data-dismiss="alert">×</a>
4   <div class="errors">
5     {% for field in form.visible_fields %}
6       {% for error in field.errors %}
7         <p>{{ field.label }}: {{ error }}</p>
8       {% endfor %}
9     {% endfor %}
10    {% for error in form.non_field_errors %}
11      <p>{{ error }}</p>
12    {% endfor %}
```

```

12     </div>
13 </div>
14 {% endif %}

```

-is replaced by the Angularified version:

```

1 <div class="alert" ng-show="register_errors"> <a class="close"
  data-dismiss="alert">x</a>
2   <div class="errors">
3     <p ng-repeat="error in register_errors">[[ error ]]</p>
4   </div>
5 </div>

```

And there you have it! We can now submit our form with Angular (and do a whole bunch of other cool stuff along the way). That was quite a bit of work but we made it. For convenience, here is the entire `static/js/registerCtrl.js`:

```

1 mecApp.factory("StripeFactory", function($q, $rootScope) {
2
3   var factory = {}
4   factory.createToken = function(card) {
5     var deferred = $q.defer();
6
7     Stripe.createToken(card, function(status, response) {
8       $rootScope.$apply(function() {
9         if (response.error) return
10          deferred.reject(response.error);
11         return deferred.resolve(response);
12       });
13     });
14
15     return deferred.promise;
16   }
17   return factory;
18 });
19
20 mecApp.factory("UserFactory", function($http) {
21   var factory = {}
22   factory.register = function(user_data) {
23     return $http.post("/register",
24       user_data).then(function(response)

```

```

24     {
25         return response.data;
26     });
27 }
28 return factory;
29 });
30
31 mecApp.controller('RegisterCtrl',function($scope, $http,
    StripeFactory, UserFactory) {
32
33     var setToken = function(data) {
34         $scope.userform.last_4_digits = data.card.last4;
35         $scope.userform.stripe_token = data.id;
36         return $scope.userform;
37     }
38
39     var logStripeErrors = function(error) {
40         $scope.stripe_errormsg = error.message;
41         throw ["There was an error processing the credit card"];
42     }
43
44     var logRegisterErrors = function(errors) {
45         $scope.register_errors = errors;
46     }
47
48     var redirect_to_user_page = function(response) {
49         if (response.errors) {
50             throw response.errors;
51         } else {
52             window.location = response.url
53         }
54     }
55
56     $scope.register = function() {
57         $scope.stripe_errormsg = "";
58         $scope.register_errors = "";
59
60         StripeFactory.createToken($scope.card)
61             .then(setToken, logStripeErrors)
62             .then(UserFactory.register)

```

```
63         .then(redirect_to_user_page)
64         .then(null,logRegisterErrors);
65
66     };
67
68 });
```

Conclusion

Okay. So, we got our form all working in Angular. We learned about promises, using them to wrap third-party APIs, and about breaking out of promise chains. We also talked a good deal about validation and displaying error messages to the user and how to keep your Django forms in-sync with your Angular validation. And of course, how to get Angular and Django to play nicely together. These last three chapters should give you enough Angular background to tackle the most common issues you'll face in developing a Django app with an Angular front-end.

In other words, you know enough now to be dangerous! That being said, Angular is a large framework and we have really just scratched the surface. If you're looking for more on Angular there are several resources on the web that you can check out. Below are some good ones to start you off:

- [Egghead Videos](#)
- [Github repo updates with a list of a ton of resources on Angular](#)
- [Very well done blog with up-to-date angular articles](#)

Exercises

1. We are not quite done yet with our conversion to Angular, as that `register` view function is begging for a refactor. A good way to organize things would be to have the current `register` view function just handle the GET requests and return the `register.html` as it does now. As for the POST requests, I would create a new `users` resource and add it to our existing REST API. So rather than posting to `/register`, our Angular front-end will post to `/api/v1/users`. This will allow us to separate the concerns nicely and keep the code a bit cleaner. So, have a go at that.
2. As I said in the earlier part of the chapter, I'm leaving the form validation for `_cardform.html` to the user. True to my word, here it is as an exercise. Put in some validation for credit card, and CVC fields.
3. While the code we added to `templates/payments/_fields.html` is great for our register page, it also affects our sign in page, which is now constantly displaying errors. Fix it!

Chapter 16

MongoDB Time!

You’ve probably heard something about [MongoDB](#) or the more general [NoSQL](#) database craze. In this chapter we are going to explore some of the MongoDB features and how you can use them within the context of Django.

There is a much longer discussion about when to use MongoDB versus when to use a [relational](#) database, but I’m going to mainly sidestep that for the time being. (Maybe I’ll write more on this later). In the meantime I’ll point you to a couple of articles that address that exact point:

- [When not to use MongoDB](#)
- [When to use MongoDB](#)

For our purposes we are going to look at using MongoDB and specifically the [Geospatial](#) capabilities of MongoDB. This will allow us to complete User Story #5 - Galactic Map:

A map displaying the location of each of the registered “padwans”. Useful for physical meetings, for the purpose of real life re-enactments of course. By Galactic here we mean global. This map should provide a graphic view of who is where and allow for zooming in on certain locations.

Building the front-end first

Normally I would start out with the backend - build the models and views and what not - but sometimes its helpful to do things the other way around. If you're having trouble envisioning how a particular feature should be implemented it can often be helpful to mockup a user interface to help you think through the user interactions. You could of course just scribble one out with a pencil, but for giggles we will start to build the entire UI first. Just to get a different perspective.

Of course we might want to do a tiny bit of back-end work first. Let's create the Django app that is going to house all of our user map functionality.

```
1 $ ./manage.py startapp usermap
```

Don't forget to add the new app to the `INSTALLED_APPS` tuple in `settings.py` as well.

Angular and Google Maps for the win

For our front-end we are going to use Google Maps to display the map plus the super handy and aptly named [Google Maps for AngularJS](#) library/[Directive](#) to make it easier to integrate Google Maps and Angular.

Download the minified versions of [angular-google-maps](#) and [lodash.js](#) (be sure to get the "underscore build") and add them to your "static/js" directory.

Now add the two libraries plus a link to the external Google Maps API files to your `__base.html` template. By now the scripts portion of your `__base.html` should look like:

```
1 <!-- scripts -->
2 <script src="https://js.stripe.com/v2/"
   type="text/javascript"></script>
3 <script type="text/javascript">
4   //<![CDATA[
5     Stripe.publishableKey = '{{ publishable }}';
6   //]]>
7 </script>
8
9 <script src="{% static "js/jquery-1.11.1.min.js" %}"
   type="text/javascript"></script>
10 <script src="{% static "js/bootstrap.min.js" %}"></script>
11 <script src="{% static "js/angular.min.js" %}"
   type="text/javascript"></script>
```

```

12 <script src="//maps.googleapis.com/maps/api/js?sensor=false"
    type="text/javascript"></script>
13 <script src="{% static 'js/lodash.underscore.min.js' %}"
    type="text/javascript"></script>
14 <script src="{% static 'js/angular-google-maps.min.js' %}"
    type="text/javascript"></script>
15 <script src="{% static 'js/application.js' %}"
    type="text/javascript"></script>
16 <script src="{% static 'js/userPollCtrl.js' %}"
    type="text/javascript"></script>
17 <script src="{% static 'js/loggedInCtrl.js' %}"
    type="text/javascript"></script>
18 <script src="{% static 'js/registerCtrl.js' %}"
    type="text/javascript"></script>
19 {% block extrajs %}{% endblock %}

```

You can see on lines 11-13 the three JS files we need to start working with Google Maps.

While we're in the `__base.html`, let's change the menu and replace the about page (since we are not really using it) with our new user maps page:

```

1 <nav class="navbar navbar-inverse navbar-static-top"
  role="navigation">
2   <header class="navbar-header">
3     <button type="button" class="navbar-toggle"
        data-toggle="collapse" data-target=".navbar-collapse">
4       <span class="sr-only">Toggle navigation</span>
5       <span class="icon-bar"></span>
6       <span class="icon-bar"></span>
7       <span class="icon-bar"></span>
8     </button>
9     <a class="navbar-brand" href="{% url 'home' %}">Mos Eisley's
      Cantina</a>
10   </header>
11   <div class="navbar-collapse collapse">
12     <ul class="nav navbar-nav">
13       <li class="active"><a href="{% url 'home' %}">Home</a></li>
14       <li><a href="{% url 'usermap' %}">User Map</a></li>
15       <li><a href="{% url 'contact' %}">Contact</a></li>
16       {% if user %}
17         <li><a href="{% url 'sign_out' %}">Logout</a></li>

```

```

18         {% else %}
19         <li><a href="{% url 'sign_in' %}">Login</a></li>
20         <li><a href="{% url 'register' %}">Register</a></li>
21         {% endif %}
22     </ul>
23 </div>
24 </nav>

```

- **Line 16** - this is the new navigation item that points to our usermap page. Of course this line is going to cause the Django templating engine to blow up unless we add the appropriately named url to `django_ecommerce\urls.py`. So, just add another item to the `urlpatterns` tuple:

```

1 url(r'^usermap/', 'usermap.views.usermap', name='usermap'),

```

Next let's define the view `usermap.views.usermap`:

```

1 from django.shortcuts import render_to_response
2
3
4 def usermap(request):
5     return render_to_response('usermap/usermap.html')

```

And then create the template (`templates/usermap/usermap.html`):

```

1 {% extends '__base.html' %}
2
3 {% load staticfiles %}
4
5 {% block content %}
6
7 <div ng-controller="UserMapCtrl">
8     <google-map center="map.center" zoom="map.zoom"
9         options="map.options"></google-map>
10 </div>
11 {% endblock content %}
12
13 {% block extrajs %}
14 <script src='{% static "js/usermapCtrl.js" %}'
15     type="text/javascript"></script>
16 {% endblock %}

```

A few lines to take note of here:

- **Line 7** - As with our earlier chapters on Angular we define a controller just for this page. And as usual we have come up with a highly creative and original name for our controller... UserMapCtrl.
- **Line 8** - A nice Directive courtesy of angular-google-maps. As you have probably already guessed it will insert a google map onto the page. We can also pass in a number of attributes to control how the map is displayed - center, zoom, and options. Check out the [API documentation](#) for more details.
- **Line 14** - This line just loads up our controller where we will configure the map options.

Before we add the controller we need to be sure to inject the google-maps service into our Angular app within *static/js/application.js* by modifying the first line to look like:

```
'javascript var mecApp = angular.module('mecApp', ['google-maps']);
```

The second argument there, which if you recall is our list/array of dependencies, just lists google-maps as a dependency and then Angular will work some “dependency injection” magic for us so that we can use the service anywhere in our Angular application.

Finally, we can create our userMapCtrl controller to actually display the map in *static/js/userMap.Ctrl.js*

```
1 mecApp.controller('UserMapCtrl', function($scope) {  
2  
3     $scope.map = {  
4         center: {  
5             latitude: 38.062056,  
6             longitude: -122.643380  
7         },  
8         zoom: 14,  
9         options: {  
10            mapTypeId: google.maps.MapTypeId.HYBRID,  
11        }  
12    };  
13  
14 });
```

This is a very simple controller that just initializes our google-map object.

Remember this line?

```
1 <google-map center="map.center" zoom="map.zoom"  
    options="map.options"></google-map>
```

We passed a number of attributes to it in order to configure the google-map. Now in our controller we define the value for those attributes:

- `map.options` stores misc options about how the map is displayed.
- `map.zoom` is the zoom level for the map; the higher the number the more zoomed in the map will be.
- `map.center` stores the coordinates of where you want the map to be centered on. (Bonus points if you can figure out the landmark at those particular coordinates.)

Oh and one more thing. If we actually want the map to display we will need to add a height for the container in our css. So add the following line to the end of `static/css/mec.css`:

```
1 .angular-google-map-container {  
2   height: 400px;  
3 }
```

With that you will actually be able to see the map on your web page. And that's it for the basic map.

To complete the usermap feature we are going to place a bunch of markers all over the map showing the locations of our users based on the user location data in our MongoDB backend (which we still need to create).

MongoDB vs SQL

To provide the functionality for User Story 5 we need a representation of each user's location. Let's dive right into some terminology. MongoDB uses different terminology than your standard SQL database, so to avoid confusion let's compare SQL with MongoDB:

Standard SQL	MongoDB
database	database
table	collection
row	document
column	field
index	index
table joins	embedded documents or linking
primary key	primary key

Why does MongoDB use different terminology?

Data Modeling in MongoDB

MongoDB is a document oriented database, while most SQL databases are table oriented databases. Table oriented databases are concerned with modeling data in tables and defining the relationships between those tables. MongoDB on the other hand models data in documents.

Documents

Documents define aggregates of data - e.g.m data that is combined together to form a total quantity or thing.

For example:

- *An advertising Flyer* is an aggregate of a main message and several supporting paragraphs, possibly also consisting of other elements such as images, testimonials, footers and what not. The point here is that a flyer is an aggregate of multiple pieces.
- *An Article* is an aggregate of an author, and abstract and several sections of text. In the case of a blog post, an article may include comments, likes and/or social mentions. Again we are modeling multiple "things", as a larger aggregate of those things.

With a relational database you might model an “Article” with an “Author” table, an “Article” table, and a “Comments” table, etc., and then you would create the necessary joins (relationships) to tie all these tables together so they could function as your data-model. Using an aggregate data-model however you would create an “Article” Document that would contain all the data necessary (authors, article, comments, etc.) to display the article.

Trade offs

There are a few tradeoffs to modeling things as a large aggregate versus a relational model.

1. *Querying efficiency* - using a join to query across several tables as opposed to a single aggregate is generally slower, so often times read speeds will increase by using an aggregate data model.
2. *Data normalization* - in relational data models you are encouraged not to duplicate data; this reduces the storage costs and can also make it easier to maintain the data integrity as each piece of data only lives in one place. However in Aggregate data models you are encouraged to duplicate data. This reduces the need for joins and makes the data model simpler, at the cost of increased storage requirements.
3. *Cascading Updates* - Imagine the “Article” relational data model. Imagine we have an author - let’s call him “Pete” - who wrote 20 articles. Now “Pete” wants to change his name to “Fred”. We just update the authors table to “Fred” and since all the Articles only hold a relationship to the Authors table all the articles will be updated as well. Conversely, in the Aggregate data model, where author information is duplicated in every Article, we would first have to find all articles with author name of “Pete” and then update each of them individually.
4. *Horizontal scalability* - Because aggregate data models store all information together it is relatively simple to scale the database horizontally by adding more nodes and “sharding” the database. With relational databases this can be much more difficult as creating joins across servers can be painful to say the least. So in general relational data models are easier to scale vertically (adding more horse power to the machine), where as aggregate data models scale horizontally (adding more machines) with relative ease.

These trade offs are by no means set in stone, there are just to get you thinking about the effects of using one data model vs another. As always the right choice will depend upon the particular problem that you are trying to solve. Use the trade offs listed here to help get you thinking about where a particular data model would fit best.

Now that we have a better understanding of Mongo. Let’s start using it.

Installing Mongo

We aren't going to get very far with MongoDB until we install it. Luckily its way easier than Postgres. Just head on over to the [mongoDB downloads page](#) and download it. There are also a number of package managers supported that can be found [here](#). `brew install mongo - anyone?`

Next up is to get the python packages we need to support MongoDB. There are two that we are going to use.

- [pymongo](#) - this is the mongodb database driver for python
- [mongoengine](#) - a Django ODM (Object-Document Mapper) for mongodb.

Let's install those:

```
1 $ pip install mongoengine django-rest-framework-mongoengine
```

pymongo is a dependency of mongoengine so you don't need to explicitly install it. After that finishes, update your *requirements.txt* file to include the new libraries:

```
1 $ pip freeze > requirements.txt
```

Cool. Now we are ready to configure Django to start working with mongodb.

Configuring Django for MongoDB

With the above installations complete we can connect to a MongoDB database - we can do CRUD operations against documents, we can execute map reduce statements and generally do anything we want to do with MongoDB.

For example, to connect to a MongoDB database named `test` on your machine you would use the following code from the Python Shell:

```
1 >>> import mongoengine
2 >>> mongoengine.connect('test')
```

That's it. That will create the database named `test` if it doesn't already exist. Of course there are more complicated forms. Say you want to connect to a MongoDB instance running on the machine `192.168.1.15` called `golaith` that requires a username and password. Then your connection might look like:

```
1 >>> import mongoengine
2 >>> mongoengine.connect("mongodb://user:pass@192.168.1.15/golaith")
```

Django doesn't support NoSQL databases natively. Meaning there is no `django.db.backends` for any of the NoSQL databases. So, we need to just configure MongoDB ourselves. To get things working with Django we need to add the above MongoDB connection in our `settings.py` file. It's best to add this right next to the 'DATABASES' settings so that all the database stuff can be found in one place. Thus, we can update `django_ecommerce/settings.py` to like so:

```
1 # DATABASES
2
3 mongoengine.connect("mec-geodata")
4
5 DATABASES = {
6     'default': {
7         'ENGINE': 'django.db.backends.postgresql_psycopg2',
8         'NAME': 'django_db',
9         'USER': 'djangouser',
10        'PASSWORD': 'djangouser',
11        'HOST': 'localhost',
12        'PORT': '5432',
13    }
14 }
```

Make sure to add the import - `import mongoengine`

Some Notes about MongoDB

In truth you don't have to set your MongoDB connection in *settings.py*. It's just a convenient place to do so as that is where all your other configuration is done. But since Django isn't really providing you any support for MongoDB you can make the connection anywhere you want to.

Also notice that we are now using two databases - our Postgres database, `django_db`, and our MongoDB database, `meo-geodata`. For us, this is exactly what we want as we are only going to be using MongoDB to store some geospatial stuff and we will use Postgres for everything else. But what if you wanted to use MongoDB to store everything?

You still need to define a SQL database. I know that seems illogical, but Django, as designed, depends upon having a relational database. So if you plan on only using MongoDB then your *settings.py* would need to look like this:

```
1 import mongoengine
2
3 ...snip...
4
5 mongoengine.connect("my-mongo-database")
6
7 DATABASES = {
8     'default': {
9         'ENGINE': 'django.db.backends.sqlite3',
10        'NAME' : 'dummy_db',
11    }
12 }
```

There are many tutorials out there that will tell you otherwise, but in Django 1.8 you are not going to get very far if you don't have the minimum `DATABASES` defined. With this setup you will basically be storing tables for Django admin and user sessions in the sqlite backend and everything else in MongoDB. If you want to also store the session information in MongoDB add the following two lines to your *settings.py* file:

```
1 # use MongoDB for sessions as well
2 SESSION_ENGINE = 'mongoengine.django.sessions'
3 SESSION_SERIALIZER = 'mongoengine.django.sessions.BSONSerializer'
```

Django Models and MongoDB

With the setup out of the way, lets create our models to be used by Django.

Edit the `usermap/models.py` file to add the models. With MongoDB, our Models will be inheriting from `mongoengine.Document` instead of `django.models`. Mongoengine is the [ODM](#) (Object Document Mapper) developed by 10Gen (the creators of Mongo), which is meant to be used very much like Django's standard model classes, but supports MongoDB on the backend as opposed to a relational database.

The `UserLocation` Model is the simplest of the two that we will be creating so let's look at that first (`usermap.models.UserLocation`):

```
1 from mongoengine.document import Document
2 from mongoengine.fields import EmailField, StringField, \
3     PointField, SequenceField
4
5
6 class UserLocation(Document):
7     email = EmailField(required=True, unique=True, max_length=200)
8     name = StringField(required=True, max_length=200)
9     location = PointField(required=True)
10    mappoint_id = SequenceField(required=True)
```

Creating a `mongoengine.Document` is really similar to creating any other model you would normally create in Django. There are just a couple of variations to point out:

- **Line 6** - Our MongoDB documents all need to inherit from `mongoengine.document.Document`.
- **Lines 7 - 8** - Declaring fields for you document with MongoDB is the same as with the standard Django ORM. As such the two fields (`email` and `name`) should feel very familiar to you.
- **Line 9** - the `location` field uses a type you may not be familiar with that is the `PointField` type. This is one of MongoDB's Geospatial fields which the mongoengine ODM has full support for.
- **Line 10** - `mappoint_id` is used because Google Map's requires a unique numeric id for all map points in order to improve caching/performance.

Nice! So with that we have our database setup. Now let's talk about storing user locations - e.g., geodata in MongoDB. In fact lets just give you a crash course in storing GeoSpatial information in a database. Then we'll come back to our models and finish coding them up.

A Geospatial primer

This primer is not intended to get you up to speed with the enormous [GIS](#) (Geographical Information Systems) industry. Instead, the point here is just to describe the bare minimum of terms so you have an idea what is going on, and so you can talk GIS at a party without sounding like, well, an idiot.

Storing Geospatial information

All geospatial features depend upon how you store the geospatial data. In MongoDB you can store 3 types of [geospatial data](#) - Points, Polygons, and LineStrings. Common use cases for each:

- Points - a particular location (GPS coordinate).
- Polygons - a large shape that spans multiple GPS coordinates, often used for buildings, countries, high pressure zones, forest fire danger zones, etc..
- LineStrings - a series of points, like a road, a flight path, a river, etc..

Each of the above items are stored in a special JSON format called GeoJSON (remember everything in MongoDB is stored as JSON) and thus they are collectively referred to as [GeoJSON](#) Objects.

Calculating Distance

In the Geospatial world indexes are used to calculate distance. More specifically they are used to calculate geometry, but distance is just a function of geometry. The point is that if you want to calculate distance between various GeoJSON objects, you have to index them. And there are several different types of indexes that you can use. Let's look at two of them to give you a feel for how they work.

Generally, when you calculate distance on a graph you use the good old Euclidean geometry:

$$\text{dist}((x, y), (a, b)) = \sqrt{(x - a)^2 + (y - b)^2}$$

This is great for flat surfaces. Thus, if you want to calculate distance against a flat surface in MongoDB you would use a “2d” index.

But the earth is round, so if we want to deal with distances across a globe - aka the spherical distance - we need to be sure we are calculating with the correct distance formula. In MongoDB we are looking for an index called “2dsphere”.

That's all we need to know for now. We are of course glossing over huge amounts of information and ignoring a lot of the finer details, but for now, [YAGNI](#).

Remember:

- There are 3 GeoJSON object types (Points, Polygons, and LineStrings) and
- There are 2 surfaces/index types used to calculate geometries (“2d”, and “2dsphere”)

For our purposes we will be using “2dsphere” indexes.

Mongoengine Support for Geospatial Features

All of what we have talked about above is exposed through [mongoengine](#) so you don't need to go down to the `Mongodb` level to get access to the geospatial stuff. By declaring the appropriate field type in your mongoengine model (aka the Document), mongoengine will handle creating the appropriate index for you.

GeoJSON Object	2d Index	2dsphere Index
Point	GeoPointField	PointField
Polygon	GeoPolygonField	PolygonField
LineString	GeoLineStringField	LineStringField

Jumping back to our `usermap.models.UserLocation` Document-

```
1 class UserLocation(Document):
2     email = EmailField(required=True, unique=True, max_length=200)
3     name = StringField(required=True, max_length=200)
4     location = PointField(required=True)
5     mappoint_id = SequenceField(required=True)
```

-you can see from the last line that our location field has a type of `PointField` - which means it is setup to use a 2dsphere index so we will be able to use it to calculate distances across the globe. And that's *exactly* what we want.

Showing a Dot on a Map

After going through some theory and important background information we now have the following:

1. A front-end displaying a Google Map that will be used to map users.
2. Mongodb installed and configured correctly with Django.
3. A mongoengine Document defined that will function as our main model to store user location info.
4. Some good background info on MongoDB and Geospatial data storage.

Now we are back on track. The next thing to do is show a dot on a map. Preferably a dot that represents a user location.

To do that we still need to do the following things:

1. Create the appropriate serializer for our REST API end point as required by Django Rest Framework.
2. Create a rest endpoint for our `UserLocation` data so we can query it from our mapping front-end.
3. Update the front-end to display the dot.
4. We should probably included testing in there somewhere as well.

Building the REST endpoint

Surprisingly enough we will start with the first item - Building the serializer for our Document. How do we do that?

Well, MongoDB stores everything as JSON right (or more technically [BSON](#) which is just a binary form of JSON). So what do we need to do? Not much actually...

MongoDB Serializers

Mongodb speaks JSON. And we want to send back JSON, so we could simply just return the `UserLocation` as JSON:

```
1 UserLocation.objects().to_json()
```

The `to_json` function exists on mongoengine's `Queryset` and `Document` classes, and with only 9 characters of code all of the documents in the `UserLocation` collection, will be expunged to JSON.

With `to_json` and the corresponding `from_json` you can now easily (de)serialize to/from JSON - which means you don't really need a serializer class at all. Let's leave out the serializer class and let MongoDB do the heavy lifting for us this time. `to_json` all the way!

Restful Endpoint for Mongo

Creating a REST endpoint for MongoDB is similar to creating an endpoint for anything else. With a few minor caveats, of course. We'll stick with Django Rest Framework; however, since we are not using a DRF serializer things will be slightly different than what we did in the previous DRF chapter.

Diving right in, let's create a `usermaps/json_views.py` file, and then let's add a function called `user_locations_lists` to handle the GET and POST requests for our `UserLocations` collection.

GET:

Let's start with the GET request:

```
1 from rest_framework.decorators import api_view
2 from rest_framework.response import Response
3 from usermap.models import UserLocation
4 import json
5
6
7 @api_view(['GET', 'POST'])
8 def user_locations_list(request):
9     if request.method == 'GET':
10         locations = json.loads(UserLocation.objects().to_json())
11         return Response(locations)
```

What's happening?

- **Lines 7 - 8** - Notice here we are using a function and not a class based view. If you recall from the earlier DRF chapter we used a class based view that inherited from `'ListModelMixin` and others. However because mongoengine implements the `queryset object` slightly differently than Django's internal `queryset (which DRF class based views are based on)` the class based views don't really work out of the box. So we need to

just create a function based view and "do it ourselves". Do note though we are using the '@api-view' decorator which if you recall provides us with the graphical test client and the rest of the DRF goodness.

- **Line 9** - DRF is based upon serializers which return an object or a query set as a Python dictionary. Thus the DRF Response object expects to get a dictionary which it will then convert to JSON before returning the JSON representation to the client. However in our case with the mongoengine function `to_json` we already have JSON.

So we have a couple of choices here. We could just not use DRF and instead use a standard Django `HttpResponse` which will happily pass on our JSON data; or, we can give DRF the dictionary it is expecting by using `json.loads()`.

In this example, we have chosen to stick with DRF and use `json.loads()` to pass DRF a dictionary. This may seem a bit convoluted as we converting from JSON to a Python dictionary only so DRF can convert back to JSON. But it's also the most straight forward way to integrate MongoDB with DRF. If performance is an issue here you may look at not using DRF at all for your REST API that have MongoDB backends, or you can explore the mongoengine `to_mongo` function which can convert a MongoDB Document (but not a queryset) to a dictionary.

- **Line 10** - Use DRF's Response object which takes a Python dictionary and returns the corresponding JSON to the client.

With that we have a way to return a list of `UserLocations` as JSON. Exactly what we wanted, and we only used a couple of lines of code.

POST:

The next part of the REST API for `UserLocations` is to allow the user to create a new `UserLocation`. We do that with a POST request:

```
1 from rest_framework import status
2 from rest_framework.decorators import api_view
3 from rest_framework.response import Response
4 from usermap.models import UserLocation
5 import json
6
7
8 @api_view(['GET', 'POST'])
9 def user_locations_list(request):
10     if request.method == 'GET':
```

```

11     locations = json.loads(UserLocation.objects().to_json())
12     return Response(locations)
13 if request.method == 'POST':
14     locations =
15         UserLocation().from_json(json.dumps(request.DATA))
16     locations.save()
17     return Response(
18         json.loads(locations.to_json()),
19         status=status.HTTP_201_CREATED
20     )

```

- **Line 1** - We need to import `rest_framework.status` at the top of our module so we can return the 201 status code.
- **Line 13** - As said before we are handling POST requests here.
- **Line 14** - Here we use the `mongoengine from_json` to create an object from the JSON passed to us by the client (`request.DATA`). Again we jump through a few hoops to get the JSON to the appropriate format, but once we do that we can create our new `UserLocation` with no problem. Also note this is really an [upsert](#), meaning if the `UserLocation` JSON data passed in from the client includes a MongoDB [ObjectID](#) then `from_json` will update the existing document, otherwise it will create a new one.
- **Line 16** - Here we just return the JSON form of the object that was just upserted with the appropriate status of 201.

There you have it, that's the REST API that works with MongoDB and provides you the standard DRF functionality.

One last thing to do is add the URL routes in `usermap/urls.py`:

```

1 from django.conf.urls import patterns, url
2
3
4 urlpatterns = patterns(
5     'usermap.json_views',
6     url(r'^user_locations$', 'user_locations_list'),
7 )

```

And then of course we need to reference that from our master URL file, `django_ecommerce/urls.py`:

```

1 from django.conf.urls import patterns, include, url
2 from django.contrib import admin
3 from payments import views

```

```

4 from main.urls import urlpatterns as main_json_urls
5 from djangular_polls.urls import urlpatterns as
    djangular_polls_json_urls
6 from payments.urls import urlpatterns as payments_json_urls
7 from usermap.urls import urlpatterns as map_json_urls
8
9 admin.autodiscover()
10 main_json_urls.extend(djangular_polls_json_urls)
11 main_json_urls.extend(payments_json_urls)
12 main_json_urls.extend(map_json_urls)
13
14
15 urlpatterns = patterns(
16     '',
17     url(r'^admin/', include(admin.site.urls)),
18     url(r'^$', 'main.views.index', name='home'),
19     url(r'^pages/', include('django.contrib.flatpages.urls')),
20     url(r'^contact/', 'contact.views.contact', name='contact'),
21     url(r'^report$', 'main.views.report', name="report"),
22     url(r'^usermap/', 'usermap.views.usermap', name='usermap'),
23
24     # user registration/authentication
25     url(r'^sign_in$', views.sign_in, name='sign_in'),
26     url(r'^sign_out$', views.sign_out, name='sign_out'),
27     url(r'^register$', views.register, name='register'),
28     url(r'^edit$', views.edit, name='edit'),
29
30     # api
31     url(r'^api/v1/', include('main.urls')),
32 )

```

This is the entirety of *django_ecommerce/urls.py*, but the important lines are:

- **Line 7** - here we import the URLs from our usermap.
- **Line 12** - This is where we add the URLs to our list of JSON URLs so we can access this JSON API for our usermaps application.

That's it for the REST API. Next up... Angular.

Connecting the Dots

Earlier we got our basic map displayed and placed a single dot on it. Now we want those dots to actually represent the user locations. As per usual in Angular, we will create a factory to first grab the data and then use a promise chain in our Controller to bind the data appropriately.

Update *static/js/userMapCtrl.js*:

```
1  mecApp.controller('UserMapCtrl', function($scope, locations) {
2
3      $scope.map = {
4          center: {
5              latitude: 38.062056,
6              longitude: -122.643380
7          },
8          zoom: 14,
9          options: {
10             mapTypeId: google.maps.MapTypeId.HYBRID,
11         }
12     };
13
14     //get all the user locations
15     $scope.locs = [];
16     cache = function(locs){
17         $scope.locs = locs;
18     }
19
20     locations.getAll().then(cache);
21 });
22
23 mecApp.factory('locations', function($http) {
24
25     var locationUrls = '/api/v1/user_locations';
26
27     return {
28         getAll:
29         function() { return
30             $http.get(locationUrls).then(function(response) {
31                 console.log(response);
32                 return response.data;
33             })
34         }
35     }
```

```

32         });
33     },
34 };
35
36 });

```

From the angular controller we can see that we are basically just adding the functionality to talk to the JSON API we just created. Now we need to update the Google map Directive in the template (*templates/usermap/usermap.html*):

```

1 <div ng-controller="UserMapCtrl">
2   <google-map center="map.center" zoom="map.zoom"
      options="map.options">
3     <markers models="locs" coords="'location'"
      idKey="'mappoint_id'"></markers>
4   </google-map>
5 </div>

```

Notice that added a markers Directive as a sub-Directive of our google-map Directive. The markers Directive is an easy way to display a series of markers from a JSON data source. To setup the markers Directive we passed the following attribute values:

- `models=locs` - this corresponds to the `$scope.locs` that we populated in our controller.
- `'coords="'location'"` - this is telling the directive that each object in the model has a `location` property that stores the coordinates.
- `'idKey="'mappoint_id'"` - recalling from earlier when we created our `UserLocationsmodel` with a `SequenceField`, we are now supplying said field to Google Maps, making the mapping gods happy.

This is the minimal setup we need to display some markers. So were good! Test it out. Fire up the app and make sure the user maps work as expected.

Getting User Locations

It's all good to populate the map with a bunch of random points that we came up with. But our user story called for real user locations. So let's use some cool HTML5 magic to grab the users location and store it in MongoDB.

The ideal location to grab the users location is when they first register, so let's look at adding the functionality there.

First, let's add a bit of code to grab the users location. We'll just put it directly into our `registerCtrl()`

```
1 $scope.geoloc = "";
2 if (navigator.geolocation) {
3     navigator.geolocation.getCurrentPosition(function(position){
4         $scope.$apply(function(){
5             $scope.geoloc = position;
6         });
7     });
8 }
```

In the above code segment `navigator.geolocation` is how Angular exposes the HTML5 capability of getting a users location. So we call `getCurrentPositions()` on the navigator and that gives us the location we need, which we store in `$scope.geoloc`.

Now let's add/update a factory in order to add the locations to MongoDB. Because this information pertains directly to the user, let's add the function to the 'UserFactory':

```
1 factory.saveUserLoc = function(coords) {
2     return $http.post("/api/v1/user_locations",
3         coords).then(function(response)
4         {
5             return response.data;
6         });
7 }
```

Next, add the controller to massage the data a bit:

```
1 saveUstrLoc = function() {
2     var data = {'name' : $scope.userform.name,
3                 'email' : $scope.userform.email,
4                 'location' : [$scope.geoloc.coords.longitude,
5                               $scope.geoloc.coords.latitude]};
5 }
```

```
6   UserFactory.saveUserLoc(data);  
7   return $scope.userform;  
8 }
```

The main thing to remember here is you need to store the data in longitude, latitude as opposed to the more obvious latitude / longitude. Also note that we are returning `$scope.userform`, this is so the next call in the user story - e. g., to register the user) gets the ‘userform’ passed in.

So now with this new functionality, let’s update our `RegisterCtrl()` promise chain:

```
1 StripeFactory.createToken($scope.card)  
2     .then(setToken, logStripeErrors)  
3     .then(saveUserLoc)  
4     .then(UserFactory.register)  
5     .then(redirect_to_user_page)  
6     .then(null, logRegisterErrors);
```

So, the only difference is that we added the line `.then(saveUserLoc)` which is now called right before calling `UserFactory.register`.

There you go. You now have a usermap with MongoDB backend.

Conclusion

Two things happened in this chapter that are very important. Perhaps you noticed them.

1. We upgraded our usage of Angular in subtle yet important ways. ²¹ We had to make design decision where there wasn't a clean "right way to do it".

Let's talk about each of these in turn.

Naming Conventions

Remember this promise chain?

```
Locations.getAll().then(cache);
```

This chains together the process of retrieving location data from our server and it does exactly what it says it does.

Compare that to the promise chain we wrote previously in the Djangular chapter.

```
pollFactory.vote_for_item(item).then(getPoll).then(setPoll);
```

While the later is focused on individual parts of the process - e.g., the `pollFactory`, the `poll` - the former is entirely based on the function - `getAll().then(cache)`.

Put another way, the former promise chain uses verbs for names which better describe their function. While the latter primarily focuses on the nouns which better describe the "thing" they are. The seemingly innocuous change of name represent the start of a paradigm shift from nouns to verbs, from objects to functions. The more we work with Angular and JavaScript the more we start to identify with the functional nature of the tools and that identification is communicated back through the code, largely by the way we choose to name things.

SEE ALSO: Now would be a great time to take a quick dive into the philosophy of naming things, and who better to guide you than the amazing author [Steve Yegge](#). Check out the wonderful blog post [Execution in the Kingdom of Nouns](#).

We are starting to move beyond just making stuff work with the language and we are moving into a stage of deeper appreciation of the design of the language - and how it can be used to express elegant solutions. This is the point where computer science and software development really start to get interesting. This is where you move beyond being functional in a language to becoming a student of the language. This is the gateway to mastery but the road becomes far more individualized from here.

Choose Your Own Integrations

The more we use our newly learned skills to tackle problems and come up with solutions the more, we learn about the framework - and what makes it tick. Having gotten this far in the course should have given you a good feel for the framework, but we'll want to practice more. We'll want to open up an empty text editor and create something from nothing, and do that over and over again. And through the challenge of solving differing and evolving problems with the same set of tools, you become better with those tools. And you begin to understand the nuances of the language.

This is part of the reason why we choose to start from an existing application, adding on bits and bits of functionality to it: We can start to see the repetitive nature of software development, and how each time through the iteration of "test -> code -> pub" we reworked things, adjusting parts of the existing framework. Whether it's refactoring the URL structure or adding various pieces of functionality to the sign up process, we turned a dial, flipped a lever, tweaked some code until things looked right and then moved forward.

This is the process of incrementally refining our solutions and building better and more robust software - this is *software craftsmanship*.

It's not just going through a tutorial where everything works. It's about getting things to work together that aren't necessarily meant for each other. Like MongoDB and DRF. While at some level they are a bit incompatible, but by understanding the framework's overall architecture and design decisions, we can come up with a seamless integration (in theory, of course).

Exercises

1. Remember how we said that Django doesn't officially have a `django.db.backends` for Mongo? Well, it also doesn't have any unit testing support for Mongo either. Remember: Django provides a `DjangoTestCase` that will automatically clear out your database and migrate the data models. Write a `MongoTestCase` that clears out a MongoDB database prior to test execution/between test runs.

Need help? Focus on these parts:

- `django.test.runner.DiscoverRunner`
- `pymongo.Connection`
- `mongoengine.connection.disconnect()`

NOTE: This is a difficult exercise, which not everybody will be able to solve. Take a shot at it though. May the force be with you.

Chapter 17

One Admin to Rule Them All

Now that we have finished all of the user stories from “Building a Membership Site”, it’s time to turn our attention to more of the backend, the non-user-facing functionality. For most membership sites we will need to update the site data from time to time. There are several use cases as to why we might need to do that:

1. Adding/removing/ updating marketing items and associated images.
2. Posting new user polls.
3. Posting new user announcements.
4. Dealing with unpaid users

It sure would be nice if there was some sort of way to provide “backend access” to our MEC app so we could do all this directly from the browser, via the Django admin site. In this chapter, we are going to look at the admin site in detail - how to use it and how to add custom functionality.

Basic Admin

I was once asked to sum up Django's basic admin site in one sentence... "It ain't pretty, but it works."

With just a few lines of code you can enable a functioning admin site that will allow you to modify your models from a web interface. It's not the most beautiful interface, but - hey - for two or three lines of code, we can't really complain. Let's start off by taking a look at the admin site.

If you haven't already, you will need to create a user who can access the admin site:

```
1 $ ./manage.py createsuperuser
2 Username: admin
3 Email address: admin@admin.com
4 Password:
5 Password (again):
6 Superuser created successfully.
```

Once done, fire up the server and navigate to the admin site - <http://localhost:8000/admin/>

NOTE: Starting in Django 1.8 the admin site is enabled by default.

Log in with the user you just created, and you should see the basic admin site, which should look something like:

Click around a bit. It's pretty intuitive; just click on the "Add" or "Change" button to perform the subsequent action. Did you notice how none of the models we created are actually shown yet? Let's fix that.

Django administration

Welcome, admin. Change password / Log out

Site administration

Authentication and Authorization

Groups

+Add

Change

Users

+Add

Change

Contact

Contact forms

+Add

Change

Flat Pages

Flat pages

+Add

Change

Payments

Users

+Add

Change

Sites

Sites

+Add

Change

Recent Actions

My Actions

None available

Figure 17.1: Basic Admin

All your model are belong to us

In order to take control of our created models, all we have to do is create an *admin.py* file and specify which models should be editable in the admin site. Let's start with the *main* application, so edit *main/admin.py* to look like:

```
1 from django.contrib import admin
2 from main.models import MarketingItem, StatusReport, Announcement,
  Badge
3
4 admin.site.register((MarketingItem, StatusReport, Announcement,
  Badge,))
```

When we register a model to be displayed in the Admin view, by default Django will do its best to come up with a standard form allowing the user to add and edit the model. Fire up the admin view and have a look at what this produces; you should now see all the above models listed under the “main” section. Try to edit one of the models; it should show you a list of existing models, and you can choose one to edit.

NOTE: If you're not seeing any data in the admin view, you can load the system data with the following command:

```
1 ```sh
2 $ ./manage.py loaddata system_data.json
3 ```
```

Coming back to our auto-generated admin views - for simple cases, what Django creates is all you need. However, you can create your own *ModelAdmin* to control how a model is displayed and/or edited. Let's do that with the *Badge* model.

First, the obligatory before shot:

Not horribly exciting. Let's add a few columns:

```
1 from django.contrib import admin
2 from main.models import MarketingItem, StatusReport, Announcement,
  Badge
3
4 admin.site.register((MarketingItem, StatusReport, Announcement, ))
5
6
7 @admin.register(Badge)
```

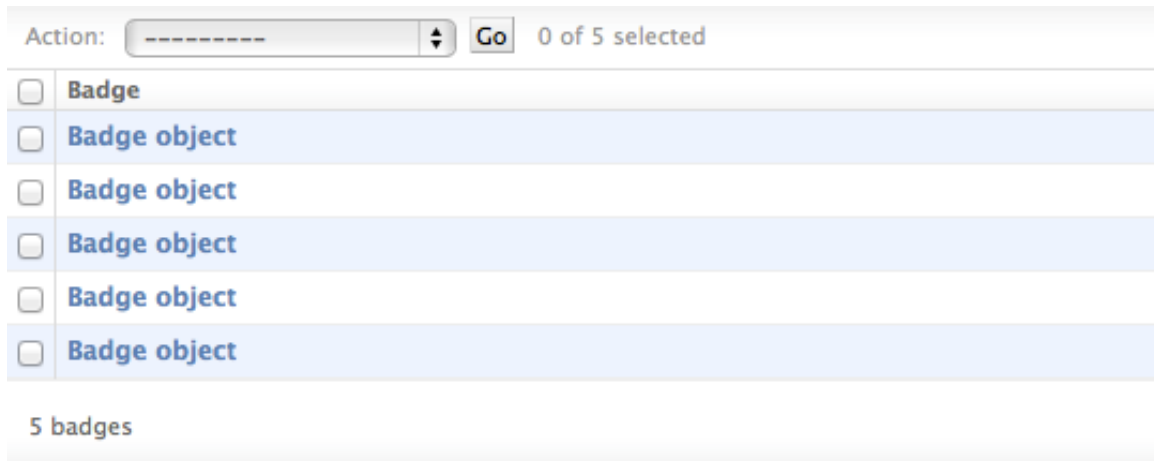


Figure 17.2: listview default

```
8 class BadgeAdmin(admin.ModelAdmin):
9     list_display = ('img', 'name', 'desc', )
```

This will add more information to the list view display so that we can tell what each item is at a glance. Once the above changes are in place, <http://localhost:8000/admin/main/badge/> should look like:



Figure 17.3: listview with columns

Images in the Admin

This is slightly better, but wouldn't it be nice to actually see the images as opposed to the image names?

To do this, and to allow uploading of new images from the admin site, let's update our badge model to use the `models.ImageField`. Before we update our model, let's briefly discuss Django's `ImageField`.

The Image Field

`ImageField` is a Django field that allows for uploading files. Note that it doesn't actually store the image in the database; it stores the image in the file system under the `settings.MEDIA_ROOT` directory. Only the meta data about the image is stored in the database.

To get an `ImageField` to behave properly, we first have to do a bit of setup:

1. Install the required libraries.
2. Define where images are saved.
3. Define the URL from which images are served.
4. Allow the development server to serve those images.

Install required libraries

Because an `ImageField` needs to load / save the image to disk, it requires the `Pillow` library. We can install that with `pip`:

```
1 $ pip install pillow
2 $ pip freeze > requirements.txt
```

This will install the library and update our *requirements.txt* file.

Define where images are saved

The `ImageField` saves images in a directory designated in `settings.MEDIA_ROOT`. Let's be *super* original and call the directory 'media'. In the *settings.py* file, simply change `MEDIA_ROOT` to:

```
1 MEDIA_ROOT = os.path.join(SITE_ROOT, 'media')
```

With this setting in place, when you upload an `ImageField` it will now be stored under "django_ecommerce/media". You can specify a subdirectory with the `upload_to` parameter passed to your `ImageField` constructor. (We will see an example of this in just a minute.)

Define the URL from which images are served

We could do this directly in our `urls.py`, but let's stick to Django's recommendation here and update `settings.MEDIA_URL` like so:

```
1 MEDIA_URL = '/media/'
```

Do note that the trailing forward slash is required.

Allow the development server to serve those images

Normally the Django development server won't give two hoots about your `MEDIA_URL`. We can add a special route that will make it available on the development server. Update `django_ecommerce.urls.py` by adding the following:

```
1
2 from django.conf import settings
3 from django.conf.urls.static import static
4
5 urlpatterns = patterns('',
6
7     ...snipped out existing urls...
8
9     #serve media files during deployment
10 ) + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

That last line will allow the Django development server to find the `MEDIA_ROOT` and serve it under the `MEDIA_URL` setting. It is worth noting that this is considered not viable for production, and so if you set `DEBUG=False` in your `settings.py` file it will turn off this “staticfiles view”.

Image Thumbnails for Admin

Now we are ready to update our `main.models.Badge` class to show the images in the admin view. First, update `main.models.Badge` to look like:

```
1 class Badge(models.Model):
2
3     img = models.ImageField(upload_to="images/")
4     name = models.CharField(max_length=100)
5     desc = models.TextField()
6
7     def thumbnail(self):
8         if self.img:
9             return u'' %
10                 (self.img.url)
11         else:
12             return "no image"
13
14     thumbnail.allow_tags = True
```

```

14
15     class Meta:
16         ordering = ('name',)

```

What did we change?

1. `img = models.ImageField(upload_to="images/")`: changed the `img` field type to `ImageField`. We talked about the `ImageField` already. The `upload_to` parameter will place these images in a subdirectory of our `MEDIA_ROOT` directory. So with this setup, our badge images will be uploaded to `media/images/`.
2. We added a function called `thumbnail()` that returns a thumbnail of the badge. It simply returns an image tag with specified height and width, or the string `no image` if no image has been uploaded.
3. Also, we added the link `'thumbnail.allow_tags = True'`, which we will talk about later.

Now we need to change the `admin.py` slightly so it references our `thumbnail` function:

```

1 @admin.register(Badge)
2 class BadgeAdmin(admin.ModelAdmin):
3
4     list_display = ('thumbnail', 'name', 'desc', )

```

That's it.

Now look at the admin view and, well, we don't have any images. Huh? That's because we previously stored all of our images in the `static/img` directory. So we need to move them to `/media/images/` and update the fields in the database. But, hey - we just configured the admin view to allow us to upload images for badges, so just use the admin view to move the images by clicking on each badge, and then using the "choose file" button.

Once done, your list view will now look like:

Looking good!

Displaying images from an ImageField

Now if we log into our main site - http://localhost:8000/sign_in - and click the "Show Achievements" link, we will see a bunch of missing images. That is because we changed the location where all of our images are stored.

We need to update our `templates/main/_badges.html` by changing the `<img>` tag so that it looks like:

Action: Go 0 of 5 selected		
☐ Thumbnail	Name	Desc
☐ 	I am a Sith Lord	Badge of honor exclusively for the dark side
☐ 	Jedi Master	Only a select few become Jedi
☐ 	May the Force	May the force be with you young padwan
☐ 	Reporting to Base	30 days of status reports
☐ 	What now, master?	Welcome to the rank of Padwan

Figure 17.4: listview_with_pics

```

1 

```

Notice that we are no longer using the `{% static %}` directive, rather we are just using `{{media_url}}{{bdg.img.url}}`. This is the combination of `settings.MEDIA_URL` and the relative URL of our uploaded file provided by the `ImageField`.

And that about wraps things up for Images. Before moving on to the next admin topic, let's look at one more example of customizing the `list_display`.

Controlling `list_display` with Admin class functions

In the above example we defined the `thumbnail` function in our model. But we could have just as easily defined it directly in our `BadgeAdmin` class. For example, if we wanted to also show a list of usernames that had a particular badge, we could update the `BadgeAdmin` class like so:

```

1 @admin.register(Badge)
2 class BadgeAdmin(admin.ModelAdmin):
3

```

```

4 list_display = ('thumbnail', 'name', 'desc',
5                 'users_with_badge', )
6
7 def users_with_badge(self, badge):
8     return ";".join("User: %s" % (u.name) for u in
9                     badge.user_set.all())

```

The function `users_with_badge` could be included in the `BadgeAdmin` class as shown above, or we could have put it in our model as we did with the `thumbnail` function. It's up to you; in terms of best practices, if you're only ever going to use the function for your admin view, put it in the `Admin` class, to keep your model less cluttered. Otherwise, put the function in your model.

Fine-grained control of the list display

When defining a function like `users_with_badge`, the list view in the admin view will have a column with heading `Users with badge`, as can be seen in this screenshot:

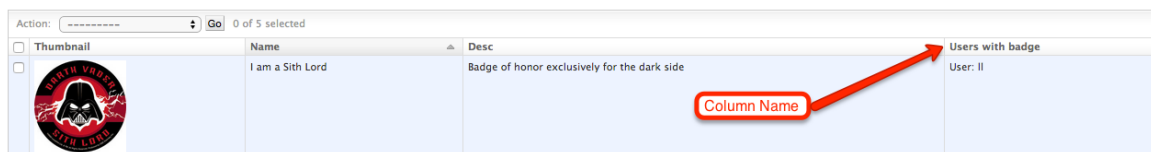


Figure 17.5: `user_with_badge`

Like most things in the admin view, this is also customizable. To change the column heading we can change our `users_with_badge` function to look like this:

```

1 @admin.register(Badge)
2 class BadgeAdmin(admin.ModelAdmin):
3
4     list_display = ('thumbnail', 'name', 'desc',
5                     'users_with_badge', )
6
7     def users_with_badge(self, badge):
8         return ";".join("User: %s" % (u.name) for u in
9                         badge.user_set.all())
9
10    users_with_badge.short_description = "Those who are worthy"

```

The last line in the listing is important here.

In that line we are setting a property of the `users_with_badge` function called `short_description` to the name we want to be displayed in the column heading. If it seems strange to set a property of a function, remind yourself that everything in Python is an object. A function is an object just like a class is an object, and so we can add properties or even other functions to a function if we want.

From Django's point of view, it will ask each item in the `list_display` if it has a `short_description` property. If so, it will use that for the column heading.

`short_description` isn't the only property that we can use to control how things are displayed. Let's say we wanted to format the cells displayed in the `users_with_badges` column. If we update our function to use a little HTML:

```
1 def users_with_badge(self, badge):
2     html = "<h3>Users</h3><ul>"
3     html += "\n".join("<li><strong>%s</strong></li>" % (u.name) for
4         u in
5             badge.user_set.all())
6     html += "</ul>"
7     return html
```

Then in the cells for the `users_with_badge` column we would actually see the raw HTML code. Not great. So we can add the property `allow_tags` to take care of that.

```
1 def users_with_badge(self, badge):
2     html = "<h3>Users</h3><ul>"
3     html += "\n".join("<li><strong>%s</strong></li>" % (u.name) for
4         u in
5             badge.user_set.all())
6     html += "</ul>"
7     return html
8
9 users_with_badge.short_description = "Those who are worthy"
10 users_with_badge.allow_tags = True
```

And then we have a list item. It ain't pretty... but it works.

One last thing: To avoid potential security issues with a known XSS scripting vulnerability, you should always use `format_html` when returning HTML code:

```
1 from django.utils.html import format_html
2
3 def users_with_badge(self, badge):
4     html = "<h3>Users</h3><ul>"
```

```

5     html += "\n".join("<li><strong>%s</strong></li>" % (u.name) for
      u in
6         badge.user_set.all())
7     html += "</ul>"
8     return format_html(html)

```

The *admin.py* file should now look like:

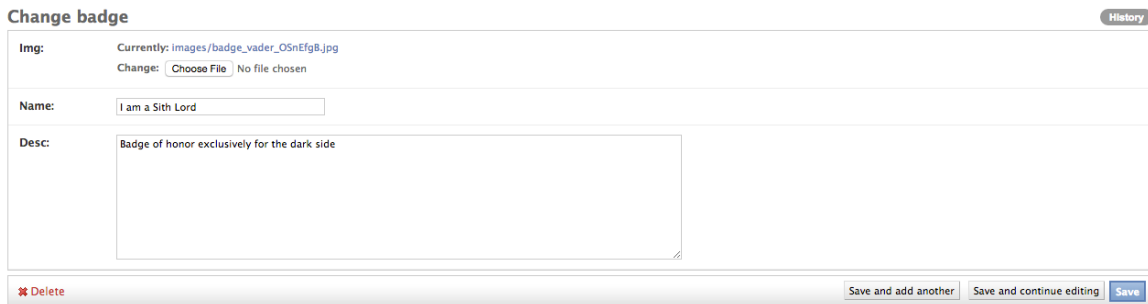
```

1 from django.contrib import admin
2 from django.utils.html import format_html
3
4 from main.models import MarketingItem, StatusReport, Announcement,
  Badge
5
6 admin.site.register((MarketingItem, StatusReport, Announcement, ))
7
8
9 @admin.register(Badge)
10 class BadgeAdmin(admin.ModelAdmin):
11
12     list_display = ('thumbnail', 'name', 'desc',
13                    'users_with_badge', )
14
15     def users_with_badge(self, badge):
16         html = "<h3>Users</h3><ul>"
17         html += "\n".join("<li><strong>%s</strong></li>" % (u.name)
18                          for u in
19                              badge.user_set.all())
20         html += "</ul>"
21         return format_html(html)
22
23     users_with_badge.short_description = "Those who are worthy"
24     users_with_badge.allow_tags = True

```

Editing Stuff

We’ve seen a bunch of ways to control how existing data is displayed, but what about editing it? Of course, from the list view of our badges we can just click on any item in the list, and Django will take us to the “change form” where we can edit to our heart’s content. By default it looks like this:



Change badge History

Img: Currently: images/badge_vader_O5nEfgB.jpg
Change: [Choose File](#) No file chosen

Name:

Desc:

Badge of honor exclusively for the dark side

[Delete](#) [Save and add another](#) [Save and continue editing](#) [Save](#)

Figure 17.6: badges change view

That actually works pretty well for our purposes. So rather than customize this model, let’s look at our `django_polls` app and its two models - `Poll` and `PollItem`. Let’s start with a review of what we just covered. See if you can figure out what the following `PollAdmin` class does (which should be defined in `django_polls/admin.py`):

```
1 from django.contrib import admin
2 from django.utils.html import format_html
3
4 from django_polls.models import Poll
5
6
7 @admin.register(Poll)
8 class PollAdmin(admin.ModelAdmin):
9
10     list_display = ('publish_date', 'title', 'highest_vote',
11                    'list_items', 'total_votes', )
12
13     def highest_vote(self, poll):
14         try:
15             return poll.poll_items().order_by('-votes')[0].text
16         except IndexError:
17             return "No Poll Items for this Poll"
```

```

19 def list_items(self, poll):
20     html = "<h3>%s</h3><ul>" % (poll.title)
21     html += "\n".join("<li><strong>%s</strong> - %d</li>" %
22                       (pi.text, pi.votes) for pi in
23                       poll.poll_items())
24     html += "</ul>"
25     return format_html(html)
26
27 list_items.short_description = "Poll results"
list_items.allow_tags = True

```

We are customizing the list view as we did in the previous section. Here is a screenshot of what the list view will now look like:

Action: Go 0 of 2 selected				
☐ Publish date	Title	Highest vote	Poll results	Total votes
☐ Nov. 5, 2014, 8:31 a.m.	Who would win in a fight?	Yoda blindfolded with one arm tied	Who would win in a fight? - Yoda blindfolded with one arm tied - 12 - AT-AT Walker - 4	16
☐ May 20, 2014, 8:23 p.m.	Who's the most powerful Jedi?	Yoda	Who's the most powerful Jedi? - Yoda - 0 - Qui-Gon Jinn - 0 - Obi-Wan Kenobi - 0 - Me... of course - 0 - Luke Skywalker - 0	0

2 polls

Figure 17.7: Poll list view

As can be seen in the image above, we are not only showing information about the Poll model, but we are showing quite a bit of information about the associated PollItems. They are tightly coupled after all, so it makes sense to show them together.

Editing with inlines

However, if we click on one of the links to edit the Poll, we will only be able to edit the title of the poll... not cool. It would be much better if we could edit the PollItems inline. Don't fear - Django has got you covered with InlineModelAdmin. We need to create one for our PollItems, and then we will tie it to our PollAdmin:

1. We can create either a TabularInline or a StackedInline; the only difference is the look and feel. The TabularInline is probably better for our purposes here. Create it with this nice little two-liner:

```

1 class PollItemInline(admin.TabularInline):
2     model = PollItem

```


2. Next up is to reference the above inline from our PollAdmin. Add this bit of code to the top of the class:

```
1 inlines = [PollItemInline,]
```

`inlines` is a list of `InlineModelAdmin` classes that should be inserted in the change form view. So once we add the inline, our change form for a Poll item becomes much more interesting. It should look like this:

Change poll History

Title:

Poll Items			
Name	Text	Votes	Delete?
PollItem object yoda	Yoda blindfolder with one arm tied	12	☐
PollItem object atat	AT-AT Walker	4	☐
		0	☐
		0	☐
		0	☐

[Add another Poll item](#)

[Delete](#) [Save and add another](#) [Save and continue editing](#) [Save](#)

Figure 17.8: `poll_changeview_with_inline`

3. Edit away! This should provide all the functionality you need. You can modify the Poll, the Poll items, add items, delete items, even change the votes if you're feeling a little bit devious.

Inlines are simple to configure and make it very easy to edit related models in one place. Do note that in the case above, we were using an inline to edit the reverse relationship (that is, edit `PollItems` from `Poll`) as opposed to editing the `Poll` from the `PollItem`. Inlines will work from either direction, allowing you to edit in any way that feels natural.

Customizing inlines

We should also point out that an `InlineModelAdmin` can be customized in much the same way that a regular `ModelAdmin` can. For example, if we don't want our pesky administrators to modify the number of votes for a `PollItem`, we could modify the `PollItemInline` class to look like:

```
1 class PollItemInline(admin.TabularInline):  
2     model = PollItem  
3  
4     readonly_fields = ('votes',)  
5     ordering = ('-votes',)
```

Note we also decided to change the ordering to show the items with the most votes at the top. The complete list of options for `InlineModelAdmins` is [here](#).

Making things Puurdy

We've got some functionality out of the admin view, and it was actually really easy to do. Now let's make it look good.

Since the rest of our site is all using Bootstrap, shouldn't our admin be doing it as well? If you really want to get into the nitty-gritty of customizing / overhauling the look and feel of the admin site then head over to the documentation and check out:

- [Adding a bit of CSS](#)
- [Overriding Admin Templates](#)
- [Replacing the entire admin site](#)

We are going to skip all of that, because [a team of really talented devs](#) has done it for us. Gotta love open source.

We can install django-admin-bootstrapped with pip:

```
1 $ pip install django-admin-bootstrapped
2 $ pip freeze > requirements.txt
```

Next all we have to do is add it to our `INSTALLED_APPS` list somewhere above the `django.contrib.admin` app. The two apps to add are:

```
1 'django_admin_bootstrapped.bootstrap3',
2 'django_admin_bootstrapped',
```

Fire up the server and have a look:

Ahhh, now that's purdy!

Title:

Who would win in a fight?

Poll items

Name		Text	Votes	Delete?
PollItem object	atat	AT-AT Walker	4	☐
PollItem object	yoda	Yoda blindfolded with one arm tied	12	☐
			0	
			0	
			0	

Add another Poll Item

Delete

Save and add another

Save and continue editing

Save

Figure 17.9: admin view bootstrap

Adminstrating Your Users

Since users are the lifeblood of most sites, it's worth taking some extra time to consider the administration of users. First off, our Users (`payments . Users`) have a pretty horrible admin view out of the box, so let's at least get the basic list view cleaned in `payments . admin . py`

```
1 from django.contrib import admin
2 from .models import User
3
4
5 @admin.register(User)
6 class UserAdmin(admin.ModelAdmin):
7
8     list_display = ('name', 'email', 'rank', 'last_4_digits',
9                     'stripe_id', )
10    ordering = ('-created_at', )
```

We've seen this before. It just controls what values are displayed in the list view of all our 'payments / users'. Now we can further customize the 'change form', which is the form used to edit an individual user. Let's do that by using 'fieldsets'. 'fieldsets' are a way to break up the form into several different / unique sections. To better understand, have a look at the following form, which creates a set of three `<fieldset>`s called User Info, Billing, and Badges.

As you can see, the form is broken up into three clearly marked sections, each with a title and some fields underneath. We can achieve this by adding the `fieldsets` attribute to our `UserAdmin` class, like so:

```
1 from django.contrib import admin
2 from .models import User
3
4
5 @admin.register(User)
6 class UserAdmin(admin.ModelAdmin):
7
8     list_display = ('name', 'email', 'rank', 'last_4_digits',
9                     'stripe_id', )
10    ordering = ('-created_at',)
11    fieldsets = (
12        ('User Info', {'fields': ('name', 'email', 'rank',)}),
13        ('Billing', {'fields' : ('stripe_id',)}),
```

User Info

Name:	Al Be Coding
Email:	d@d.com
Rank:	Padwan

Billing

Stripe id:	cus_4cPwyqB7iCZNB

Badges

Badges:	Badge object Badge object Badge object Badge object + Hold down "Control", or "Command" on a Mac, to select more than one.

Figure 17.10: User fieldsets

```
13     ('Badges', {'fields' : ('badges',)}),  
14 )
```

Notice that `fieldsets` is a tuple of two-tuples. Each two-tuple contains the title of the section and a dictionary of attributes for that section. In our case the only attribute we used was the `fields` attribute, which lists the fields to be displayed in the fieldset. More information about fieldsets can be found [here](#).

With that, we now have a decent looking interface to administrate our users. However, we can make the admin interface even more useful. Let's look at exactly how to do that next.

Resetting passwords

Someone is bound to call / email and say, “Your password reset isn’t working, and I can’t get in the site”. This usually means they can’t remember the password they put in when they reset their password 30 seconds ago. What we want to do is to get the password reset for them right. So let’s add that capability to the admin view.

Now if we were using Django’s Default User Model, password resets are included out of the box. You can navigate to <http://127.0.0.1:8000/admin/auth/user/>, select a user, and you will see a change password link. Pretty easy.

However, we are using a custom user model - the one in `payments.models.User`. And the password reset form doesn’t work for custom user models. If you want a quick and dirty hack to get it to work, check out [this Stack Overflow question](#) - but let’s do something a bit more fun and learn more about customizing the admin interface along the way.

So what are we going to do? Simple: We are going to modify the change form for `payments.models.User` to use Angular to make a REST call to reset a user’s password. In order to accomplish this amazing feat, we need to do the following:

1. Customize the ‘change_form’ template for ‘payments / users’ to add a “change password” link.
2. Create a REST API to reset the password.
3. Integrate angular.js into the ‘change_form’ so we can call our REST API.
4. Provide some nice status messages.

Let’s go through each of these topics in a bit more detail.

Tweaking the built-in admin templates

The admin interface is built using Django views and templates in the same way that we built our application. There is really no magic there. This means we can extend (or even replace) the existing templates to add additional functionality. The template we are interested in extending is called `change_form.html`. This is the form used when editing a model instance.

Since we have already customized the `change_form` with the `fieldset` business above, it would be good to not blow away the work we have done there. Rather, let’s see if we can just add a link somewhere near the top of the page to reset the password.

We have a few options for how to extend the `change_form.html` template:

- Extend the `change_form.html` for the entire admin site. (Useful if you want to add something to every single change form for all your models.) To do this add a `templates/admin/change_form.html`
- Extend the `change_form.html` for a particular Django app. If we wanted certain editing functionality for every model in our `django_polls` app, we would choose this option. To do this add a `templates/admin/django_polls/change_form.html`
- Extend the `change_form.html` for a particular model - i.e., for `payments.models.user`. *This is exactly what we'll do.* To do this we just need to create an HTML template named `change_form.html` and put it in `templates/admin/payments/user/change_form.html`. This way we will only override the change form for the User model.

Let's have a look at what our template might look like:

```

1 {% extends "admin/change_form.html" %}
2 {% load staticfiles %}
3 {% load i18n admin_urls %}
4
5 {% block object-tools-items %}
6
7 <li class="dropdown" id="menuReset" is-open="isopen">
8   <a class="dropdown-toggle" href id="navLogin">Reset Password</a>
9   <div class="dropdown-menu" style="padding:17px;">
10     <form class="form" id="resetpwd" name="resetpwd" method="post"
11       ng-submit="resetpass('{{original.id}}')" >
12       {% csrf_token %}
13       <input name="pass" id="pass" type="password"
14         placeholder="New Password" ng-model="pass" required>
15       <input name="pass2" id="pass2" type="password"
16         placeholder="Repeat Password" ng-model="pass2" required><br>
17       <button type="submit" id="btnLogin" class="btn"
18         >Reset Password</button>
19     </form>
20   </div>
21 </li>
22
23 {{ block.super }}
24 {% endblock %}

```

- **Line 1** - Notice that we are extending `admin/change_form.html` which is the default change form that ships with Django's Admin. (Actually, since we installed

django-admin-bootstrapped, it is the template from there... but they are basically the same.) If you're curious as to what the default `change_form.html` template that ships with Django looks like, you can find it [here](#).

- **Line 5** - By plugging into the `object-tools-items` block, we can add additional links across the top of the page (next to the history link).
- **Line 7 - 21** - This is a bootstrap dropdown that will prompt the user to enter a new password twice. It will look like this:

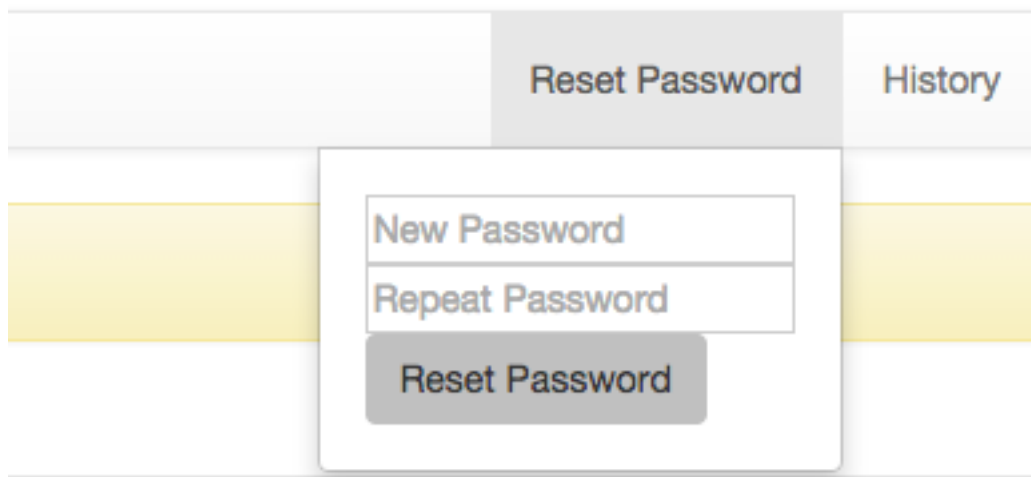


Figure 17.11: Reset Password dropdown

- **Line 7, 11, 14, 16** - If you look closely you can see that we have already “Angularfied” this form. It won't work yet since we don't have Angular set up within the page, but it doesn't hurt to get a headstart on things.
- **Line 12** - Notice on this line we are using the context variable `{{ original.id }}`. In the admin views, the object that we are currently editing is always called `original`, so in this case we are grabbing the id from the object.
- **Second to last line** - Don't forget to call `block.super` to ensure the rest of the stuff in the block loads.

Okay. So at this point we have a link across the top to reset password and a nice little form, but it doesn't do anything. Since we are going to integrate Angular, it's best to create a REST endpoint that we can call to update our password.

Reset Password REST endpoint

We have created a number of REST endpoints already, and here is yet another example. Let's run through it really quick.

The URL we want is `/api/v1/users/password/<user id>`, which translates to the following in `payments/urls.py`:

```
1 from django.conf.urls import patterns, url
2
3
4 urlpatterns = patterns(
5     'payments.json_views',
6     url(r'^users$', 'post_user'),
7     url(r'^users/password/(?P<pk>[0-9]+)$',
8         json_views.ChangePassword.as_view(),
9         name='change_password'),
10 )
```

Notice on **line 5** that we are using the view `json_views.ChangePassword`, which looks like:

```
1 from rest_framework import generics, status, permissions
2 from payments.serializers import PasswordSerializer
3 from django.http import Http404
4
5 class ChangePassword(generics.GenericAPIView):
6     """
7     Change password of any user if superadmin.
8     * pwd
9     * pwd2
10    """
11    permission_classes = (permissions.IsAdminUser,)
12    serializer_class = PasswordSerializer
13
14    def get_object(self, pk):
15        try:
16            return User.objects.get(pk=pk)
17        except User.DoesNotExist:
18            raise Http404
19
20    def put(self, request, pk, format=None):
```

```

21     user = self.get_object(pk)
22     serializer = PasswordSerializer(user, data=request.DATA)
23     if serializer.is_valid():
24         serializer.save()
25         return Response("Password Changed.")
26     return Response(serializer.errors,
27                       status=status.HTTP_400_BAD_REQUEST)

```

A few things to notice about this class.

1. `permission_classes = (permissions.IsAdminUser,)` - This means the user making the request must return `True` for the `is_staff` property.
2. `get_object` function - This grabs the appropriate user based on the `pk` passed in from the URL parser.
3. `put` - We use a `put` function because we are updating an existing user. We don't actually have the logic to reset the password in this function; we create a `PasswordSerializer` and let it handle all the work of updating / setting the users password, then we return the appropriate status.

And the final piece to our new REST API for resetting a user's password is the `'PasswordSerializer'`, which can be found `inpayments.serializers.py`. It should look like this:

```

1  from payments.models import User
2  from rest_framework import serializers
3  PASSWORD_MAX_LENGTH = User._meta.get_field('password').max_length
4
5
6  class PasswordSerializer(serializers.Serializer):
7      """
8      Reset password serializer
9      """
10     password = serializers.CharField(
11         max_length=PASSWORD_MAX_LENGTH
12     )
13     password2 = serializers.CharField(
14         max_length=PASSWORD_MAX_LENGTH,
15     )
16
17     def validate_password2(self, attrs, source):
18         pwd2 = attrs[source]

```

```

19     pwd = attrs['password']
20     if pwd2 != pwd:
21         raise serializers.ValidationError("Passwords don't
22             match")
23
24     return attrs
25
26 def restore_object(self, attrs, instance=None):
27     """ change password """
28     if instance is not None:
29         print("set password")
30         instance.set_password(attrs.get('password'))
31         return instance
32
33     # we don't create new instances
34     return None

```

- For the above serializer we define two fields password and password2, both with a max_length set to what is defined in the payments.models.User model.
- The function validate_password2 is a custom validation function that validates to ensure the two passwords passed in match. If not, it will return an error telling the user “Passwords don’t match”. There is no need to create a validation function for each field, because both fields are checked in the one validator.
- If you recall from our earlier chapters on DRF, restore_object is used when you want to control how an object is created from a Python native object (aka a dictionary). In our case we want to call instance.set_password(attrs.get('password')), which will handle encrypting the new password and adding it to the user (note you still need to call save, but we are doing that in our 'ChangePassword' view). Further note that we only restore_objects if we are passed in an existing User (for the value of instance). This is because it doesn't make sense to try to set the password for a new user (because we don't have any other user data).

And that's it. We now have a fully functioning REST API for changing passwords. You may want to test it out by going to <http://127.0.0.1:8000/api/v1/users/password/1> and using the DRF form.

Now back to the front end to add Angular support and call our newly minted API.

Adding Angular to the Admin Site

Coming back to our template, since we are extending the `admin/change_form.html`, adding Angular support is just a matter of finding the appropriate blocks to use to put in the necessary parts of the template.

First off, we need to reference the appropriate JavaScript files. In our case, `angular.min.js` and an `admin.js`, which we will create to hold the admin-specific code. We will also add another JavaScript file, `ui-bootstrap-tpls-0.<version>.min.js`. This is a library called [UI-Bootstrap](#), which is a set of Bootstrap components written in Angular. We will use this to control the `navLogin` dropdown. Add the following to the bottom of `templates/admin/payments/user/change_form.html`:

```
1 {% block footer %}
2     {{ block.super }}
3     <script src="{% static 'js/angular.min.js' %}"
4         type="text/javascript"></script>
5     <script src="{% static 'js/ui-bootstrap-tpls-0.11.0.min.js' %}"
6         type="text/javascript"></script>
7     <script src="{% static 'js/admin.js' %}"
8         type="text/javascript"></script>
9 {% endblock %}
```

Make sure you also add the UI-Bootstrap library to your “static/js” folder. You can grab it [here](#) (make sure you get version 0.11.0) and then save it to the correct folder.

Now we need to add our `ng-app` and our `ng-controller` directives to the HTML page. To ensure we can use Angular functionality throughout, we want to wrap the entire page in our `ng-app` / `ng-controller` directives. Due to the way the `admin/change_form.html` template is set up, it's not very straight forward on how to do that.

First, we will extend the `navbar` block and add our `div` like so:

```
1 {% block navbar %}
2     <div ng-app="adminApp" ng-controller="AdminCtrl">
3         {{ block.super }}
4     {% endblock %}
```

Notice we didn't close the `div`. We will close it in the footer block. This in effect will create a `div` that wraps almost the entire page. It's a bit odd to do things this way, but we are bound by the blocks that are exposed in `admin/change_form.html`. We could also choose not to inherit from `admin/change_form.html` and instead just create our own template from scratch, but even though that would make for a clean way to declare `ng-app` and `ng-controller`, that's a bit overkill for our purposes.

Putting it all together, our templates/admin/payments/user/change_form.html should now look like this:

```
1 {% extends "admin/change_form.html" %}
2 {% load staticfiles %}
3 {% load i18n admin_urls %}
4 {% block navbar %}
5     <div ng-app="adminApp" ng-controller="AdminCtrl">
6         {{ block.super }}
7     {% endblock %}
8 {% block object-tools-items %}
9     <li class="dropdown" id="menuReset" is-open="isopen">
10         <a class="dropdown-toggle" href
11             id="navLogin">Reset Password</a>
12         <div class="dropdown-menu" style="padding:17px;">
13             <form class="form" id="resetpwd" name="resetpwd" method="post"
14                 ng-submit="resetpass('{{original.id}}')" >
15                 {% csrf_token %}
16                 <input name="pass" id="pass" type="password"
17                     placeholder="New Password" ng-model="pass" required>
18                 <input name="pass2" id="pass2" type="password"
19                     placeholder="Repeat Password" ng-model="pass2"
20                     required><br>
21                 <button type="submit" id="btnLogin" class="btn">
22                     Reset Password</button>
23             </form>
24         </div>
25     </li>
26 {{ block.super }}
27 {% endblock %}
28 {% block footer %}
29 </div> <!-- closes the ng-app div -->
30 {{ block.super }}
31 <script src="{% static 'js/angular.min.js' %}"
32     type="text/javascript"></script>
33 <script src="{% static 'js/ui-bootstrap-tpls-0.11.0.min.js' %}"
34     type='text/javascript'></script>
35 <script src="{% static 'js/admin.js' %}"
36     type="text/javascript"></script>
```

```
34 {% endblock %}
```

Fire up the page in the admin view and click on ‘view source’ or ‘inspect element’ from your browser. You should see our ng-app directive just under the content div like so:

```
1 <div id="content" class="colM">
2   <div ng-app="adminApp" ng-controller="AmdinCtrl" class="ng-scope">
3     <!-- ...snip basically the enire page... -->
4     <!-- END Content -->
5   </div>
6 </div>
7 <!-- closes the g-app div -->
8 <footer id="footer"></footer>
```

Adding the AdminCtrl

Notice that we created an Angular app called adminApp and a controller called AdminCtrl. We will define both of those in our newly created js/admin.js file. Starting off with just the basic definitions, our js/admin.js file should look like:

```
1 var adminApp = angular.module('adminApp', ['ui.bootstrap']);
2
3 adminApp.config(function($interpolateProvider, $httpProvider) {
4   $interpolateProvider.startSymbol('[[').endSymbol(']]');
5   $httpProvider.defaults.headers.common['X-CSRFToken'] =
6     $('input[name=csrfmiddlewaretoken]').val();
7   }
8 );
9 adminApp.controller('AdminCtrl', function($scope) {});
```

In line one we are injecting the ui.bootstrap module into our adminApp (which gives us access to Bootstrap componenets implemented in Angular). Everything else we have covered previous in the Angular chapters, so it should be a review.

Calling the reset password REST Endpoint

First thing we would like to do is hook up the reset password REST endpoint to our form. We can do that with the ng-submit directive for our resetpwd form. As a reminder, here is what the form looks like in our change_form.html template.

```

1 <form class="form" id="resetpwd" name="resetpwd" method="post"
2   ng-submit="resetpass('{{original.id}}')" >
3   {% csrf_token %}
4   <input name="pass" id="pass" type="password"
5     placeholder="New Password" ng-model="pass" required>
6   <input name="pass2" id="pass2" type="password"
7     placeholder="Repeat Password" ng-model="pass2" required><br>
8   <button type="submit" id="btnLogin" class="btn">
9     Reset Password</button>
10 </form>

```

- **Line 2** - Notice that we are calling the resetpass function on ng-submit and passing it {{original.id}}. As a reminder, original is a context variable set by Django admin that is the model instance currently being edited.

Let's create the resetpass function in our AdminCtrl (in js/admin.js). But first we are going to need an Angular factory to call our REST endpoint. Let's call it AdminUserFactory:

```

1 adminApp.factory("AdminUserFactory", function($http) {
2   var factory = {};
3   factory.resetPassword = function(data) {
4     var pwdData = {password : data.pass, password2 : data.pass2};
5     return $http.put("/api/v1/users/password/" + data.user, pwdData)
6       .then(function(response)
7         {
8           return response;
9         });
10  };
11
12  return factory;
13 });

```

We have seen this type of factory before. Basically we are just building the url `api/v1/users/password/<user id>` and sending a PUT request with the password data.

Now we can hook up the factory into our AdminCtrl controller and create the resetpass function:

```

1 adminApp.controller('AdminCtrl', function($scope, AdminUserFactory)
2   {

```



```

3  $scope.resetpass = function(userId) {
4      var data = {
5          'user': userId,
6          'pass' : $scope.pass,
7          'pass2': $scope.pass2
8      };
9      return AdminUserFactory.resetPassword(data);
10 };
11
12 });

```

Above we inject the 'AdminUserFactory' and call it with our resetpass functionality. You can test the admin form now, and it should successfully reset the password for the user. However, from the UI you won't receive any success or failure messages, so let's take care of that as well.

WARNING: In case it's not apparent, with this technique we are sending the passwords in clear-text back to the server. Thus we should really only do this if we are operating over an HTTPS connection so we can ensure our passwords are encrypted while sent across the wire.

Adding Success and Failure Messages

Our Admin Site already makes use of Bootstrap-style alerts to display information to the user. So let's use those alerts to inform the user of the results of the resetpass call. To do that, we can extend the form_top block from admin/change_form.html, which is a section of HTML that will appear right across the top of the form (just below our reset password link) used to edit our model. We can do that by adding the following to our template:

```

1 {% block form_top %}
2     <div class="alert" ng-class="alertClass"
3         ng-show="afterReset">[[ msg ]]</div>
4 {% endblock %}

```

- **Line 2** - Bootstrap alerts use one of four classes for styling - alert-info, alert-warning, alert-success, or alert-danger. Each class simply changes the background color of the alert from white to yellow to green to red respectively. Thus if we successfully reset the password, we want to use the alert-success class on our alert. We'll use the alert-danger class if we fail to update the password. Somewhere in AdminCtrl

we will set the value of `alertClass` to the appropriate class so things will be displayed with the appropriate background color.

- **Line 3** - Here we are using a simple `ng-show` to control if the alert is displayed or not. And the text in the alert will be tied to `$scope.msg`.

Now let's update the `AdminCtrl`:

```
1 adminApp.controller('AdminCtrl', function($scope, AdminUserFactory)
2 {
3     $scope.afterReset = false;
4
5     $scope.resetpass = function(userId) {
6         $scope.afterReset = false;
7         var data = {
8             'user': userId,
9             'pass' : $scope.pass,
10            'pass2': $scope.pass2
11        };
12        AdminUserFactory.resetPassword(data)
13            .then(showAlert, showAlert);
14    };
15
16    var showAlert = function(data) {
17        $scope.afterReset = true;
18        var msg = "";
19        $scope.alertClass = "alert-danger";
20
21        if (data.status == 200) {
22            $scope.alertClass = "alert-success";
23            $scope.pass = "";
24            $scope.pass2 = "";
25        }
26
27        $scope.msg = data.data;
28
29    };
30
31 });
```

Okay. There's a fair bit going on there, so let's break it down bit by bit:

- **Line 3**- Ensure by default that we don't show the alert.
- **Line 12** - Here we are taking advantage of Angular promises to call our showAlert function after the REST call completes. Since our REST call will return a 400 error if the passwords don't match we set showAlert as both the success and failure handler. (That's why it appears to be called twice).
- **Line 15** - Our new showAlert function.
- **Lines 16-18** - The first bit of the showAlert function sets `$scope.afterReset = true`. This will cause our alert to be shown. We also want to be sure to clear out any previous messages in our alert, so we will initialize the msg to null. Let's default to a red background.
- **Lines 20-24** - This is our success handler we want to ensure our alert will have the proper green colored background, and we can also clear out our form.
- **Line 26** - Finally, we set the text of our alert to be the response returned from our REST call.

This will produce alert messages that look like this:

Success message



Password Changed.

Figure 17.12: Alert Success

Failure message



```
{ "password2": ["Passwords don't match"] }
```

Figure 17.13: Alert Failure

The Success message looks fine, but the failure message looks a bit ugly - `{ "password2": ["Passwords don't match"] }`

This is because DRF will return an array of error messages linked to each field that fails validation / has an error. We can add a special case in our showAlert function to handle this.

Replace:

```
1 $scope.msg = data.data
```

With:

```

1 if (typeof data.data == 'string') {
2   msg = data.data;
3 } else {
4   for (var x in data.data) {
5     msg += data.data[x].toString() + " ";
6   }
7 }
8 $scope.msg = msg;

```

This basically says, “If we get back a string (which is the success case) then just display it, otherwise loop through the list of errors and display them all”.

It would be nice to hide the password reset form after we are finished, so let’s make that happen. This is where the ui-bootstrap really helps. If you look back at our HTML for the menuReset list_item, you will see:

```

1 <li class="dropdown" id="menuReset" is-open="isopen">

```

The class “dropdown” is actually an Angular directive defined in ui-bootstrap that allows us to programmatically control if the dropdown is displayed or hidden. **is-open** allows you to specify the scope variable that will control this. Since we have set it to **isopen**, all we have to do is set `$scope.isopen` to `false` in our controller to hide the dropdown, or `true` to show it. Thus we add `$scope.isopen = false` to the end of our `showAlert` function and that will close the dropdown menu. Now if we put it all together, our controller should look like this:

```

1 adminApp.controller('AdminCtrl', function($scope, $http,
2   AdminUserFactory) {
3
4   $scope.afterReset = false;
5   $scope.isopen = false;
6
7   $scope.resetpass = function(userId) {
8     $scope.afterReset = false;
9     var data = {
10       'user': userId,
11       'pass' : $scope.pass,
12       'pass2': $scope.pass2
13     };
14     AdminUserFactory.resetPassword(data)
15       .then(showAlert, showAlert);
16   };
17 }

```

```

16
17 var showAlert = function(data) {
18     $scope.afterReset = true;
19     var msg = "";
20     $scope.alertClass = "alert-danger";
21
22     if (data.status == 200) {
23         $scope.alertClass = "alert-success";
24         $scope.pass = "";
25         $scope.pass2 = "";
26     }
27
28     if (typeof data.data == 'string') {
29         msg = data.data;
30     } else {
31         for (var x in data.data) {
32             msg += data.data[x].toString() + " ";
33         }
34     }
35
36     $scope.msg = msg;
37     $scope.isopen = false;
38 };
39
40 });

```

And there you have it. A fully functioning Angular-enabled password reset function for a custom model in the Admin view. That was a lot of work for a password reset, but hopefully you learned a good deal about the various ways you can customize the admin view and extend it to do whatever you need.

Oh, and one more thing - Since we are using custom user models, we may want to hide Django's Auth Models from the admin view. We can do that by simply adding the following lines to `payments/admin.py`:

```

1 from django.contrib.auth.models import User as DjangoUser
2 from django.contrib.sites.models import Site
3 from django.contrib.auth.models import Group
4
5 admin.site.unregister(DjangoUser)
6 admin.site.unregister(Group)

```

```
7 admin.site.unregister(Site)
```

Conclusion

In this chapter we have talked about a number of ways to modify the admin interface. Let's review quickly.

Adding models to the Admin interface

Just create a `ModelAdmin` and register it with `@admin.register` like this:

```
1 from django.contrib import admin
2 from payments.models import User
3
4 @admin.register(User)
5 class UserAdmin(admin.ModelAdmin):
6     pass
```

Controlling the list view of a model

Control the fields displayed with the `list_display` attribute of a `ModelAdmin`, i.e.:

```
1 class UserAdmin(admin.ModelAdmin):
2     list_display = ('name', 'email', 'rank', 'last_4_digits',
3                     'stripe_id', )
```

Remember that the list of fields can be a field or a callable. For example, we can create a thumbnail view with the following callable:

```
1 def thumbnail(self):
2     if self.img:
3         return u'' %
4             (self.img.url)
5     else:
6         return "no image"
7
8     thumbnail.allow_tags = True
```

Change column headings with the `'short_description'` attribute, i.e.:

```
1 thumbnail.short_description = "badges"
```

Control ordering of the list with the `ordering` attribute using `-` for reverse ordering, i.e.:

```
1 ordering = ('-created_at', )
```

Controlling how we edit models

To control the look / grouping of the `change_form`, we can use the `fieldsets` property, which will break the fields into sections with headings and fields:

```
1 fieldsets = (  
2     ('User Info', {'fields': ('name', 'email', 'rank',)})),  
3     ('Billing', {'fields' : ('stripe_id',)}),  
4     ('Badges', {'fields' : ('badges',)}),  
5 )
```

We can edit related models using inlines:

```
1 class PollItemInline(admin.TabularInline):  
2     model = PollItem  
3  
4 @admin.register(Poll)  
5 class PollAdmin(admin.ModelAdmin):  
6  
7     inlines = (PollItemInline,)
```

We can control which fields are read-only and thus not editable with the `readonly_fields` attribute:

```
1 class PollItemInline(admin.TabularInline):  
2     model = PollItem  
3  
4     readonly_fields = ('votes',)
```

Controlling the look and feel of the admin view

The admin view uses Django Views and Templates just like any other part of Django does, so we can overwrite or extend those views / templates or even create our own. We talked about how to extend the `admin/change_form.html` by creating our own template that extends from that template, and we talked about some of the various blocks that we might want to extend when we customize the template. Just create the template and the appropriate directory and start overriding the blocks you need.

You can go quite far using this technique, or you could even create your own Views / Templates and URLs and completely replace the entire Django admin. Do whatever makes the most sense from you.

Other Admin functionatliy

I have tried to cover the most commonly used parts of the Django Admin so you can hit the ground running. But there is a lot more functionality you can control in the admin such as searching / filtering, customized CSS, the position of the various controls, adding custom validation and more.

The [Admin Docs](#) do a fairly good job of explaining the functionality, so have a read through to find our more.

Exercises

1. We didn't customize the interfaces for several of our models, as the customization would be very similar to what we have already done in this chapter. For extra practice, customize both the list view and the change form for the following models:

- `Main. Announcements`
- `Main. Marketing Items`
- `Main. Status reports`

Note both `Announcements` and `Marketing Items` will benefit from the thumbnail view and `ImageField` setup that we did for `Badges`.

2. For our `Payments . Users` object in the change form you will notice that badges section isn't very helpful. See if you can change that section to show a list of the actual badges so it's possible for the user to know what badges they are adding / removing from the user.

Chapter 18

Testing, Testing, and More Testing

Why do we need more testing?

Throughout this book I have been singing the praises of unit testing. It has helped us catch some errors in our application, made it easier to refactor, and changed our application as our requirements changed - and hopefully it has given us a sense of security about the working nature of our application.

However this security may be a bit unfounded, as we have only really been testing half of the application - namely the Python half. In the more recent chapters we have added a lot of front-end Angular code, which is becoming increasingly important to the overall functionality of our application. However, at this point we haven't done much testing for that functionality. In this chapter we are going to cover strategies to test the front-end (and the back-end) of our application as one whole unit. This is commonly referred to as GUI testing or end-to-end (E2E) testing.

What needs to be GUI tested?

If you recall from the chapter on **Test Driven Development**, we talked about the importance of unit testing and how it can help in producing solid design and quality code. GUI testing is the next level after your foundation is covered through *unit* (and some *integration*) testing. We can think of GUI testing as the icing on the cake to really help solidify that quality of your application. For our app in particular, GUI testing can help us in the following areas:

1. Since GUI testing is end-to-end it will help to cover/test our Angular front-end code.
2. Further GUI testing can easily cover integrations with external services, like when we use Stripe to validate a credit card.
3. Although we know the individual units of our python code work, we don't have tests to ensure the end-to-end business processes work as expected and that our python back-end code is correctly integrating with our front-end Angular code.

While unit testing is very much about ensuring the individual pieces of code function as expected, GUI testing is much more about ensuring that the application as a whole does what it supposed to do. Put another way, we are now more concerned with meeting the *business requirements* as opposed to in unit testing, which is about the *technical requirements*. When we combine the two styles of testing, we can have a high degree of certainty that the application functions as expected.

That being said, it is import to put the various types of testing in their appropriate place and use them accordingly. To that end, it's very helpful to look to the **Agile Testing Pyramid** as a reference that helps us determine the amount of effort we should invest into each of the various types of testing:

As can be seen from the Agile Testing Pyramid above, the majority of our focus in testing should be in creating unit tests, followed closely by system/integration tests, with the smallest number of tests being the GUI tests. There are many reasons for this, but it all generally boils down to the following two:

1. Unit tests are cheaper/faster to write and run.
2. Tests that cover more application logic are more prone to failure and difficult maintenance.

With that in mind, we want to roughly shoot for - 50% unit test, 30% integration tests, and 20% GUI tests. While it is important to understand that these numbers are not hard and fast, they are a good rule of thumb to help us understand where are efforts are best focused.

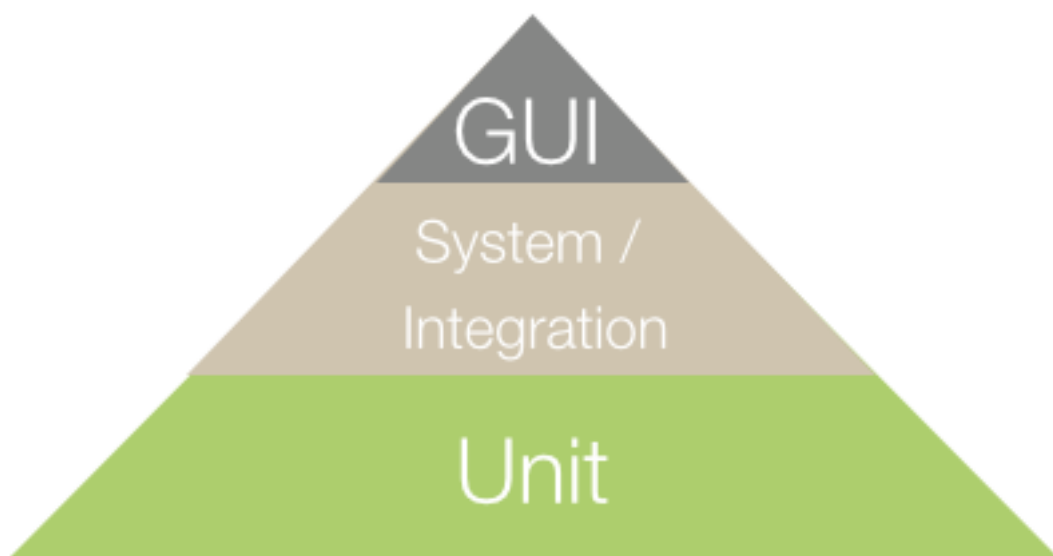


Figure 18.1: Agile Testing Pyramid

Ensuring Appropriate Testing Coverage

Looking inward at our application, let's apply these numbers to the testing that we are doing. To date, we should have roughly the following number of tests:

Test Type	Number of Tests	Percentage
Unit Test	32	68%
Integration Tests	15	32%
GUI Tests	0	0%

Basically we have 47 tests in our `../tests` folder. 15 of those are integration tests since they deal with the database and are not strictly unit tests. But honestly, the split is a bit arbitrary (as we discussed in the **Unit Testing** chapter at the beginning of this course).

Looking at the percentages, we can see that we are not too far off from the suggested numbers in the Agile Testing Pyramid. We just need to add the GUI tests, roughly 12 of them. Do note though it's more important to look at our requirements and make sure we have tests for those requirements than to just say, we are going to add 12 tests because that will ensure we have 20% GUI tests. Just keep in mind that if we start having significantly fewer or more tests than 12 we should be asking ourselves questions about whether we have the appropriate number of GUI tests.

What about JavaScript testing?

One thing that is missing from our discussion about testing is JavaScript testing - e.g., writing unit tests specifically for our Angular code base. There is value in unit testing Angular code as well. However we will cover a lot of what would be covered by Angular unit tests with the GUI testing. In the interest of brevity we therefore won't be covering Angular unit testing in this chapter.

For those readers who are interested to dig into angular unit testing. Here are a few resources which may be of use.

- [The Angular Docs](#)
- [Test Angular with Karma](#)
- [Testing AngularJS With Protractor and Karma](#)

GUI Testing with Django

Enough theory, let's jump into GUI Testing with Django.

Install Selenium

First thing's first, we need to get [Selenium](#) (the tool we use for automated GUI Testing) installed:

```
1 $ pip install selenium
2 $ pip freeze > requirements.txt
```

Set up the Directory Structure

Next, since GUI Tests can take a much longer time to run than unit tests, let's create a directory structure that allows us to easily run either unit test or GUI tests - or both. To do that, let's change the test folder to look like:

```
1
2 __init__.py
3 gui
4     __init__.py
5     testGui.py
6 unit
7     __init__.py
8     contact
9         __init__.py
10        testContactModels.py
11    main
12        __init__.py
13        testJSONViews.py
14        testMainPageView.py
15        testSerializers.py
16    payments
17        __init__.py
18        testCustomer.py
19        testForms.py
20        testUserModel.py
21        testViews.py
```

Under our “tests” directory we created two sub-folders, “gui” and “unit(don't forget that each folder needs an `__init__.py`’). Then we can run the tests we want by typing the following from the “django_ecommerce” directory:

1. Unit tests: `./manage.py test ../tests/unit`
2. GUI tests: `./manage.py test ../tests/gui`
3. All tests: `./manage.py test ../tests`

With that out of the way, we can write our first test case.

Our First GUI Test Case

Create a file called *testGui.py* in the “tests/gui” folder:

```
1 from django.contrib.staticfiles.testing import
    StaticLiveServerTestCase
2 from selenium import webdriver
3
4
5 class LoginTests(StaticLiveServerTestCase):
6
7     @classmethod
8     def setUpClass(cls):
9         cls.browser = webdriver.Firefox()
10        super(LoginTests, cls).setUpClass()
11
12    @classmethod
13    def tearDownClass(cls):
14        cls.browser.quit()
15        super(LoginTests, cls).tearDownClass()
16
17    def test_login(self):
18        self.browser.get('%s%s' % (self.live_server_url,
19                                   '/sign_in'))
```

What's going on here?

- **Line 1:** Django provides a `LiveServerTestCase` that will automatically startup a server, but `LiveServerTestCase` doesn't correctly serve static files, upon which our application depends. Thus, we are going to use `StaticLiveServerTestCase` as that will ensure our static files are served correctly by the LiveServer.
- **Line 2** - In Selenium a `webdriver` represents a browser. There is a `webdriver` for all the major browsers. Also the module `selenium.webdriver` serves as an entry point into all of Selenium's functionality.
- **Line 9** - Initialize the `webdriver` (in this case the Firefox `webdriver`) and store it in a class variable. This line will also cause our Firefox web browser to open when the test is ran.
- **Line 14** - When our tests finish, we want to close the browser; calling `quit()` will close the browser and end the associated system process.

- **Line 17** - Our first test. Selenium has a [rich API](#), which we will discuss more in this chapter. The first API call you need to know about is `get`, which opens a URL in the browser. `self.live_server_url` is provided by Django's `LiveServerTestCase` and points to the root URL for the server started by the test case.

Now run the test with the following command from `django_ecommerce`:

```
1 $ ./manage.py test ../tests/gui
```

You should see the Firefox browser open up and show the sign in page briefly before closing. Be sure that you see the correctly formatted page with the appropriate looking CSS styles. If you see a page that looks like it doesn't have CSS, you probably didn't configure your static files correctly.

The Selenium API

That test got things working, but it doesn't really do much. To get Selenium to do stuff (i.e., execute GUI tests) we need to examine the API first. With Selenium, the main point of entry into the API is the webdriver (which, again, can be thought of as the browser). So as we have shown above, the first step is grabbing a reference to the webdriver:

```
1 browser = webdriver.Firefox()
```

Once we have a reference called `browser` in the line above, we can then instruct the webdriver to perform various actions. We can break these actions up into three main categories (there are more, but let's start with these three first):

1. Locating elements
2. Acting on those elements
3. Waitings for things to happen

Jumping back to the previous example...

```
1 self.browser.get('%s%s' % (self.live_server_url, '/sign_in'))
```

This is an example of navigation, as it causes the browser to navigate to the specified URL. (In our case the `sign_in` page.) Once we are on that page, we probably want to enter a username and password. To do that, we have to locate the appropriate elements, which brings us to...

Locating Elements

The webdriver in Selenium exposes an API that lets you interact with a webpage's DOM (e.g., HTML structure) to locate elements. As soon as you locate an element, a `WebElement` is returned that exposes element-specific interactions such as `send_keys`, `clear`, `click`, etc.

In Selenium there are several ways to locate elements, but you typically need to concern yourself with only two or three of these methods.

Locating an element by ID

This should be your default, go-to way to locate an element. Why? Because locating an HTML element by its HTML ID is about the fastest way you get the browser to parse the DOM and actually locate the element.

Here's how you do it:

```
1 email_textbox = self.browser.find_element_by_id("id_email")
```

Likewise, for the password text box we would use this line:

```
1 pwd_textbox = self.browser.find_element_by_id("id_password")
```

The above two examples will tell the browser to search through the DOM for the element with the ID specified (remember in HTML, IDs *should* be unique) and return it.

Locating elements that don't have ids

Sometimes we want to interact with elements that don't have IDs. For example maybe we want to check to see if the sign-in page has a header that says "Sign In". The HTML element for that looks like this:

```
1 <h2 class="form-signin-heading">Sign in</h2>
```

Since this element doesn't have an ID let's look at the other "locators" provided by Selenium, namely:

- `find_element_by_name`
- `find_element_by_xpath`
- `find_element_by_link_text`
- `find_element_by_partial_link_text`
- `find_element_by_tag_name`
- `find_element_by_class_name`
- `find_element_by_css_selector`

Each one of these will use a different attribute to search through the DOM and find an element. However as a best practice there are only two of these "locators" that I would recommend using:

- `find_element_by_name`
- `find_element_by_css_selector`

All the other "selectors" have issues; they are either potentially slow, like `find_element_by_xpath`, generally not specific enough like `find_element_by_tag_name` or `find_element_by_class_name` or they are vulnerable to breaking when/if you decide to translate your application like `find_element_by_link_text` and `find_element_by_partial_link_text`.

So that leaves us with:

- `find_element_by_name` – this is essentially the same as `find_element_by_id`, but it searches for the name attribute instead of the `id` attribute. Since many HTML forms use name instead of `id`, this is a pretty safe option to use.

The next option (and my personal favorite locator) is `find_element_by_css_selector`. This uses the same syntax you use in your CSS files; you pass it a CSS selector to find and element.

Given the HTML form below-

```

1 <label for="id_email">Email:</label>
2 <div class="input">
3   <input id="id_email" name="email" type="email">
4 </div>
5 <div class="custom-error ng-hide"
6   ng-show="signin_form.email.$dirty &&
   signin_form.email.$invalid">
7   Email is invalid:<span
   ng-show="signin_form.email.$error.required"
   class="ng-hide">value is required.</span>
8   <span ng-show="signin_form.email.$error.email"
   class="ng-hide">Input a valid email address.</span>
9 </div>
10 <div class="clearfix">
11   <label for="id_password">Password:</label>
12   <div class="input">
13     <input id="id_password" name="password" type="password">
14   </div>

```

-here are a few examples of CSS selectors:

```

1 self.browser.find_element_by_css_selector("[type=email][name=email]")
   # returns the element on line 3
2 self.browser.find_element_by_css_selector("#id_password") # this is
   the same as find_element_by_id
3 self.browser.find_element_by_css_selector(".ng-hide") # find
   element with ng-hide class

```

Basically anything that you can find/style with your CSS you can locate with Selenium by using `find_element_by_css_selector`! Pretty cool.

WARNING: One big caveat to keep in mind: `find_element_by_css_selector` uses CSS 2.0 selectors and not the 3.0 selectors (made popular by jQuery) that you may be used to. See [here](#) for a complete list of valid CSS 2.0 selectors that will work with Selenium.

Really between the following three “golden selectors”-

- `find_element_by_id`
- `find_element_by_name`
- `find_element_by_css_selector`

-you should be able to locate any element in your application.

Acting on elements you have located

Once you locate an element, with a command like-

```
1 email_textbox = self.browser.find_element_by_id("id_email")
```

-you will have a 'WebElement' that you can interact with. In Selenium the 'WebElement' has a rich API which is described [here](#). We will focus on some of the more common methods in the API.

For our purposes, we want to sign into our application, so we are going to need to input an email address and password in the appropriate text boxes.

We can do that with the following bit of code. Update *testGui.py*:

```
1 def test_login(self):
2     self.browser.get('%s%s' % (self.live_server_url, '/sign_in'))
3     email_textbox = self.browser.find_element_by_id("id_email")
4     pwd_textbox = self.browser.find_element_by_id("id_password")
5     email_textbox.send_keys("test@test.com")
6     pwd_textbox.send_keys("password")
```

Notice the last two lines where we used the `send_keys()` function to type a value into the appropriate text boxes. `send_keys` simulates a user typing into the text box, so if you run the test now with the following command:

```
1 $ ./manage.py test ../tests/gui
```

You should actually see the values being typed into the appropriate text boxes.

Next step is to click on the "Sign In" button. We can do that with the following code, which first locates the element, and then clicks it:

```
1 self.browser.find_element_by_name("commit").click()
```

Here we have done two things in one line just to show that you can chain functions in Selenium if you so desire. Also, since we are submitting a form, we can choose to use the `submit()` function, which will find the form that encompasses the element and call `submit` on it directly:

```
1 self.browser.find_element_by_name("commit").submit()
```

Again, update the code. Regardless of which method you choose, if you run the test again, you should see the following output:

```

1 ./manage.py test ../tests/gui
2 Creating test database for alias 'default'...
3
4 <ul class="errorlist"><li>Incorrect email address or
  password</li></ul>
5 .
6 -----
7 Ran 1 test in 5.772s
8
9 OK
10 Destroying test database for alias 'default'...

```

Notice that Selenium outputted to us what gets returned when we submit the form/click the button. This is helpful for debugging, and it shows us that we are getting an error message because the user doesn't exist in the database. This is true, because we are starting with a fresh database where the test user doesn't exist. We could fix that by creating a user in our setUpClass function. But first, let's convert this to a negative test so we can verify that we do indeed get an error message if we input invalid data. This will also show us how to verify information on the screen.

To do that, let's change the test_login() function like so:

```

1 def test_failed_login(self):
2     self.browser.get('%s%s' % (self.live_server_url, '/sign_in'))
3     email_textbox = self.browser.find_element_by_id("id_email")
4     pwd_textbox = self.browser.find_element_by_id("id_password")
5     email_textbox.send_keys("test@test.com")
6     pwd_textbox.send_keys("password")
7
8     # click sign in
9     self.browser.find_element_by_name("commit").submit()
10
11     # find the error element
12     invalid_login =
13         self.browser.find_element_by_css_selector(".errors")
14     self.assertEqual(invalid_login.text,
15                     "Incorrect email address or password")

```

What did we change?

1. We renamed the test to the more appropriate test_failed_login

2. We added the last three lines, which grab the “errors” div and check that the text in the div is equal to the string we expected.

We could do a slight variation to check that our email validator works correctly like so:

```
1 def test_failed_login_invalid_email(self):
2     self.browser.get('%s%s' % (self.live_server_url, '/sign_in'))
3     email_textbox = self.browser.find_element_by_id("id_email")
4     pwd_textbox = self.browser.find_element_by_id("id_password")
5     email_textbox.send_keys("test@")
6     pwd_textbox.send_keys("password")
7
8     # click signin
9     self.browser.find_element_by_name("commit").submit()
10
11    # find the error element
12    invalid_login =
13        self.browser.find_element_by_css_selector(".errors")
14    self.assertEqual(invalid_login.text,
15                     "Email: Enter a valid email address.")
```

This is basically the same test, but we change the text we send to email_textbox and change the associated check for the error message. You could do the same thing to check for other validations such as password required, email required, etc.

Let's move on to the positive case where login is successful. First, let's create a user in the setUp() function.

```
1 def setUp(self):
2     self.valid_test_user = User.create(
3         "tester", "test@valid.com", "test", 1234)
4
5 def tearDown(self):
6     self.valid_test_user.delete()
```

Above we create the user, and then clean up the created user. Now to write our login test:

```
1 def test_login(self):
2     self.browser.get('%s%s' % (self.live_server_url, '/sign_in'))
3     email_textbox = self.browser.find_element_by_id("id_email")
4     pwd_textbox = self.browser.find_element_by_id("id_password")
5     email_textbox.send_keys("test@valid.com")
```

```

6     pwd_textbox.send_keys("test")
7
8     # click sign in
9     self.browser.find_element_by_name("commit").submit()
10
11    # ensure the user box is shown, as this means we have logged in
12    self.assertTrue(
13        self.browser.find_element_by_id("user_info").is_displayed())

```

Again, very similar to the previous tests, but here on the last line we are trying to find an element with the ID of `user_info`. Then we call `is_displayed()` on that element, which will return true if it is visible to the user. Since we know the User Info box only shows up after a successful login, this is a good visual cue we can use to verify that our test is running correctly.

Selenium Best Practice #1 You should add a check for a visual cue after each page transition or AJAX action (that should change the display). This way our test will start to function in the same way a manual tester would test your application, e.g., do a bit of work, check that the application is responding correctly, do a bit more work, check again, etc.

Waiting for things to happen

The way our script is written right now, it will basically run as fast as possible, expecting the browser/our application to keep up. (Actually that's not entirely true; Selenium is smart enough to wait for a page to load, but page loads don't include AJAX actions.) Thus, in our test cases so far, Selenium will wait for the page to load after we navigate to the `/sign_in` page, but for many of the other steps, such as displaying the error message, we are using AJAX calls to update the page. Selenium doesn't know anything about AJAX. Since everything is running on our local machine, we should be okay, but if we start to test in parallel on a remote server or our computer randomly slows down for other reasons, our tests may fail because Selenium has gotten ahead of our application.

Selenium Best Practice #2 Always wait for things to happen, to make your tests more robust. Things like AJAX calls, javascript animations, or slow test clients can take time and Selenium isn't very patient. Best to wait...

Selenium gives us two main ways of waiting for things to happen:

1. Implicitly
2. Explicitly

Implicit Waits

Webdriver provides the function `implicitly_wait()` that will set up an implicit wait for each locator call. We can enable this in our `setUpClass` method by changing it to read as follows:

```
1 @classmethod
2 def setUpClass(cls):
3     cls.browser = webdriver.Firefox()
4     cls.browser.implicitly_wait(10)
5     super(LoginTests, cls).setUpClass()
```

Line 4 is the important one here. We are telling the webdriver that if at first it doesn't find an element (when we call any of our `find_element` functions), then keep retrying to find that element for up to ten seconds. If the webdriver finds that element in the time span, then everything is good and the script will continue; if not, then the script will fail.

In the interest of making our test less brittle and more reliable, it is a good idea to always include a call to `implicitly_wait`. However, there is one drawback: if you have an

implicitly_wait call and you want to check that an element is **not** displayed, you would write code like this:

```
1 self.assertFalse(self.browser.find_element_by_id("user_info").is_displayed())
```

Keep in mind that in this case, it would take the full 10 seconds for this line to complete because of the implicitly_wait call we made in our setUpClass.

Explicit Waits

For this reason - waiting - some people don't like to use implicit waits. There are also times when you know a particular operation may take longer than other things on the page (perhaps because you're accessing an external API). In those cases (or if you just want to be explicit as in [The Zen of Python](#), you can use an explicit wait.

Going back to our test_failed_login(), let's put an explicit wait in when checking for errors. The entire updated test case would look like this:

```
1 from selenium.webdriver.common.by import By
2 from selenium.webdriver.support.ui import WebDriverWait
3 from selenium.webdriver.support import expected_conditions as EC
4
5 def test_falied_login(self):
6     self.browser.get('%s%s' % (self.live_server_url, '/sign_in'))
7     email_textbox = self.browser.find_element_by_id("id_email")
8     pwd_textbox = self.browser.find_element_by_id("id_password")
9     email_textbox.send_keys("test@test.com")
10    pwd_textbox.send_keys("password")
11
12    # click sign in
13    self.browser.find_element_by_name("commit").submit()
14
15    # find the error element
16    invalid_login = WebDriverWait(self.browser, 10).until(
17        EC.visibility_of_element_located((By.CSS_SELECTOR,
18            ".errors")))
19
20    self.assertEqual(invalid_login.text,
11         "Incorrect email address or password")
```

- **Lines 1 - 3** - We need these additional imports; put them at the top of your module.

- **Lines 5 - 14** - Nothing new here.
- **Lines 16 - 17** - Let's break this apart as there are a number of things going on here.

The first is:

```
1 WebDriverWait(self.browser, 10)
```

Alone this command will just cause the script to pause for ten seconds - not a great idea (as we already discussed). However, we can chain the `until` function to our wait, which will cause the wait to stop waiting as soon as the `until` function returns a truthy result. `until` takes a [callable](#) that accepts a single argument.

Let's look at a quick example:

```
1 WebDriverWait(self.browser, 10).until(lambda x: True)
```

This code will immediately exit out of the `WebDriverWait` since the result is truthy. Then, when `WebDriverWait` exits, it will return the value returned by the callable passed to `until`, which in this case is `True`.

Usually, we want to wait until something happens on the screen, like an element becomes present, or visible, or not visible, or something like that. This is where the `expected_conditions` comes in to play. `expected_conditions` is a Selenium-provided module that allows us to do a number of checks on the web page. In our specific example, we are using this bit of code (where `EC` is the `expected_conditions` module):

```
1 EC.visibility_of_element_located((By.CSS_SELECTOR, ".errors"))
```

Here, we are calling the `visibility_of_element_located()` class in the `expected_conditions` module. This class will not only call `webdriver.find_element` but will also call `is_displayed` and wait until that returns `True` before it returns the `WebElement`.

In effect these two lines will just keep trying to find the `.errors` element (by default every 0.5 seconds) until it is either found (and visible) or until ten seconds pass. If not found then the call will raise a `selenium.common.exceptions.TimeoutException`.

There are other `expected_conditions` you can use; have a look [here](#). Or you can call your own function. Besides `until` there is also an `until_not` if you want to wait until something is `Falsy`.

Page Objects

Now that we know how to find elements, wait for them and act on them, we can do most anything we need to do with Selenium. At this point most people just start writing a lot of test cases, not giving much thought to maintaining and/or updating the test cases in the future. Since we have already learned the fact that GUI tests are expensive to maintain, it's worth looking at how we might reduce the cost of maintaining these test cases.

One way to do that is to use the **Page Objects** pattern.

As the name implies, using Page Objects basically means creating an object for each page of your application, and encapsulating the GUI testing code for the corresponding page in that object. For example we could create a login page object and put all the functionality associated with login in that page object, like filling our username and password, checking error messages and so on.

This analogy works pretty well with a simple login page but with AJAX-rich applications, the name Page Object might be a bit misleading, because often a page is too big to encapsulate in a single object. While our `sign_in` page maps well to the term page object, our user page probably doesn't. This is because our user page consists of multiple separate pieces of functionality - i.e., announcements, status updates, badges, user info, polls, etc.. Thus, for the user page, we should look to create a page object for each of the separate pieces of functionality (such as user polls, status updates, etc..)

Again, the point is to make things more maintainable and easy to understand. Motivation for the need to use page objects can be obtained by looking at the GUI tests we have written thus far in the chapter, like:

```
1 def test_failed_login(self):
2     self.browser.get('%s%s' % (self.live_server_url, '/sign_in'))
3     email_textbox = self.browser.find_element_by_id("id_email")
4     pwd_textbox = self.browser.find_element_by_id("id_password")
5     email_textbox.send_keys("test@test.com")
6     pwd_textbox.send_keys("password")
7
8     # click sign in
9     self.browser.find_element_by_name("commit").submit()
10
11    # find the error element
12    invalid_login = WebDriverWait(self.browser, 10).until(
13        EC.visibility_of_element_located((By.CSS_SELECTOR,
14                                         ".errors")))
```

```

14
15     self.assertEqual(invalid_login.text,
16                       "Incorrect email address or password")
17
18 def test_failed_login_invalid_email(self):
19     self.browser.get('%s%s' % (self.live_server_url, '/sign_in'))
20     email_textbox = self.browser.find_element_by_id("id_email")
21     pwd_textbox = self.browser.find_element_by_id("id_password")
22     email_textbox.send_keys("test@")
23     pwd_textbox.send_keys("password")
24
25     # click signin
26     self.browser.find_element_by_name("commit").submit()
27
28     # find the error element
29     invalid_login =
30         self.browser.find_element_by_css_selector(".errors")
31     self.assertEqual(invalid_login.text,
32                     "Email: Enter a valid email address.")

```

We can see there is a lot of duplication of code. And just like with our production code, if we remove this duplication and share code it will make it cheaper to maintain our GUI tests. To do that, we use the Page Object pattern, of course.

Let's get a clearer picture of how this works by creating a `sign_in` page object.

First, let's create a new folder - `tests/gui/pages/`. Be sure to put the `__init__.py` file in there, and then create our first page object class in a new file called `testPage.py`. Starting off with a base class, let's create the 'SeleniumPage' class:

Page Object Base Classes

```

1 class SeleniumPage(object):
2     '''Place to allow for any site-wide configuration you may want
3     for you GUI testing.
4     '''
5
6     def __init__(self, driver, base_url="", wait_time=10):
7         self.driver = driver
8         self.base_url= base_url
9         self.wait_time= wait_time

```

Nothing fancy here - The class just allows us to set a few necessary variables. Next, create a base class for the elements on the page:

“Python from selenium.webdriver.support.ui import WebDriverWait from selenium.webdriver.support import expected_conditions as EC

class SeleniumElement(object):

```
1 def __init__(self, locator):
2     self.locator = locator
3
4 def __get__(self, obj, owner):
5     driver = obj.driver
6     wait_time = obj.wait_time
7     return WebDriverWait(driver, wait_time).until(
8         EC.visibility_of_element_located(self.locator))
```

“

A couple of things of interest here:

1. The `__init__` function initializes the element with a locator... so we can find it. A locator, here, takes the form of something like (By.ID, "id_email")
2. The `__get__` function is one of those “special Python methods”. It’s used for property access, so when we call the get property for this object, this bit of code will run. Pay attention to the second parameter, `obj`. This is the object that the property is called on. In other words, if you were do to this-

```
1 class Page(SeleniumPage):
2     email = SeleniumElement((By.ID, "id_email"))
3     def dummy_method(self):
4         self.email
```

-the last line of the code above, where we call `self.email`, would call the `__get__` method of `email` and pass in `Page` for the `obj` parameter.

Why does this matter? For our purposes, it’s quite handy as it allows us an easy way to pass values (i.e. `driver` and `wait_time`) from the containing class (`Page`) to our element (`email`). This lets us set explicit wait times once per page object and not have to worry about setting it each time we try to access an element.

Without further ado... the Page Object

With the base classes written, let’s look at what our `SignInPage` page object looks like:


```

1 class SignInPage(SeleniumPage):
2
3     email_textbox = SeleniumElement((By.ID, "id_email"))
4     pwd_textbox = SeleniumElement((By.ID, "id_password"))
5     sign_in_button = SeleniumElement((By.NAME, "commit"))
6     error_msg_elem = SeleniumElement((By.CSS_SELECTOR, ".errors"))
7     sign_in_title = SeleniumElement((By.CSS_SELECTOR,
8                                     ".form-signin-heading"))
9
10    def do_login(self, email, pwd):
11        self.email_textbox.send_keys(email)
12        self.pwd_textbox.send_keys(pwd)
13        self.sign_in_button.submit()
14
15    @property
16    def error_msg(self):
17        return self.error_msg_elem.text
18
19    @property
20    def rel_url(self):
21        return '/sign_in'
22
23    def go_to(self):
24        self.driver.get('%s%s' % (self.base_url, self.rel_url))
25        assert self.sign_in_title.text == "Sign in"

```

Let's look at this class piece-by-piece:

- **Lines 3 - 7** - This is where we define the element on the page. Defining elements this way ensures that all the elements are using our custom locating code in `SeleniumElement.__get__`, plus it exposes the elements to the unit test in case we want to use them directly.
- **Lines 9 - 12** - For common functions that we need to perform on a page to test it, it's helpful to create a method in our page object. As such, we have created the `do_login()` method on these lines, which fills in the email and password fields and submits the form.
- **Lines 14-16** - Since we will be testing the error messages displayed frequently, we have created a nice helper property to make that easy to do.
- **Lines 18-20** - In the spirit of encapsulation, we can put the relative URL for our page here so we don't need to litter our tests with the URL.

- **Lines 22-24** - For page objects that represent an actual page, it's helpful to have a `go_to()` function that will navigate to the page and verify that we are indeed on the correct page.

That's our page object. For clarification purposes, here is the entire code for the sign in test:

```

1 from selenium import webdriver
2 from selenium.webdriver.common.by import By
3 from selenium.webdriver.support.ui import WebDriverWait
4 from selenium.webdriver.support import expected_conditions as EC
5
6 from payments.models import User
7 from django.contrib.staticfiles.testing import
   StaticLiveServerTestCase
8
9
10 class SeleniumPage(object):
11     '''Place to allow for any site-wide configuration you may want
12     for you GUI testing.
13     '''
14
15     def __init__(self, driver, base_url=None, wait_time=10):
16         self.driver = driver
17         self.base_url = base_url
18         self.wait_time = wait_time
19
20
21 class SeleniumElement(object):
22
23     def __init__(self, locator):
24         self.locator = locator
25
26     def __get__(self, obj, owner):
27         driver = obj.driver
28         wait_time = obj.wait_time
29         return WebDriverWait(driver, wait_time).until(
30             EC.visibility_of_element_located(self.locator))
31
32
33 class SignInPage(SeleniumPage):

```

```

34     email_textbox = SeleniumElement((By.ID, "id_email"))
35     pwd_textbox = SeleniumElement((By.ID, "id_password"))
36     sign_in_button = SeleniumElement((By.NAME, "commit"))
37     error_msg_elem = SeleniumElement((By.CSS_SELECTOR, ".errors"))
38     sign_in_title = SeleniumElement((By.CSS_SELECTOR,
39         ".form-signin-heading"))
40
41     def do_login(self, email, pwd):
42         self.email_textbox.send_keys(email)
43         self.pwd_textbox.send_keys(pwd)
44         self.sign_in_button.submit()
45
46     @property
47     def error_msg(self):
48         return self.error_msg_elem.text
49
50     @property
51     def rel_url(self):
52         return '/sign_in'
53
54     def go_to(self):
55         self.driver.get('%s%s' % (self.base_url, self.rel_url))
56         assert self.sign_in_title.text == "Sign in"

```

That's it! Now let's refactor our tests.

Page Object in Action

We could rewrite our entire page test set:

```

1 from selenium import webdriver
2 from selenium.webdriver.common.by import By
3 from selenium.webdriver.support.ui import WebDriverWait
4 from selenium.webdriver.support import expected_conditions as EC
5
6 from payments.models import User
7 from django.contrib.staticfiles.testing import
8     StaticLiveServerTestCase

```

```

9
10 class LoginTests(StaticLiveServerTestCase):
11
12     @classmethod
13     def setUpClass(cls):
14         cls.browser = webdriver.Firefox()
15         cls.browser.implicitly_wait(10)
16         super(LoginTests, cls).setUpClass()
17
18     @classmethod
19     def tearDownClass(cls):
20         cls.browser.quit()
21         super(LoginTests, cls).tearDownClass()
22
23     def setUp(self):
24         self.valid_test_user = User.create(
25             "tester", "test@valid.com", "test", 1234)
26         self.sign_in_page = SignInPage(self.browser,
27             self.live_server_url)
28
29     def tearDown(self):
30         self.valid_test_user.delete()
31
32     def test_login(self):
33         self.sign_in_page.go_to()
34         self.sign_in_page.do_login("test@valid.com", "test")
35         self.assertTrue(
36             self.browser.find_element_by_id("user_info").is_displayed())
37
38     def test_failed_login(self):
39         self.sign_in_page.go_to()
40         self.sign_in_page.do_login("test@test.com", "password")
41         self.assertEqual(self.sign_in_page.error_msg,
42             "Incorrect email address or password")
43
44     def test_failed_login_invalid_email(self):
45         self.sign_in_page.go_to()
46         self.sign_in_page.do_login("test@", "password")
47         self.assertEqual(self.sign_in_page.error_msg,
48             "Email: Enter a valid email address.")

```

```

48
49
50 class SeleniumPage(object):
51     '''Place to allow for any site-wide configuration you may want
52     for you GUI testing.
53     '''
54
55     def __init__(self, driver, base_url=None, wait_time=10):
56         self.driver = driver
57         self.base_url = base_url
58         self.wait_time = wait_time
59
60
61 class SeleniumElement(object):
62
63     def __init__(self, locator):
64         self.locator = locator
65
66     def __get__(self, obj, owner):
67         driver = obj.driver
68         wait_time = obj.wait_time
69         return WebDriverWait(driver, wait_time).until(
70             EC.visibility_of_element_located(self.locator))
71
72
73 class SignInPage(SeleniumPage):
74
75     email_textbox = SeleniumElement((By.ID, "id_email"))
76     pwd_textbox = SeleniumElement((By.ID, "id_password"))
77     sign_in_button = SeleniumElement((By.NAME, "commit"))
78     error_msg_elem = SeleniumElement((By.CSS_SELECTOR, ".errors"))
79     sign_in_title = SeleniumElement((By.CSS_SELECTOR,
80         ".form-signin-heading"))
81
82     def do_login(self, email, pwd):
83         self.email_textbox.send_keys(email)
84         self.pwd_textbox.send_keys(pwd)
85         self.sign_in_button.submit()
86
87     @property

```

```

87     def error_msg(self):
88         return self.error_msg_elem.text
89
90     @property
91     def rel_url(self):
92         return '/sign_in'
93
94     def go_to(self):
95         self.driver.get('%s%s' % (self.base_url, self.rel_url))
96         assert self.sign_in_title.text == "Sign in"

```

As you can see, not only have we saved a lot of code, but our tests are pretty clear and readable now as well. Double win!

Believe it or not, that's 90% of what you need to know to successfully test your web page with Selenium. Of course, that remaining 10% can be pretty tricky. In order to help with that, the final section in this chapter will list a few gotchas you may run into when writing tests for our application.

When Selenium Doesn't Work

Sometimes in GUI automation, things don't really work the way you think they might. This section is meant to be a rapid-fire list of common problems that you may have when testing our application and how to solve them.

Selecting elements from a dropdown

This is not supported out of the box.

How do you select from drop-downs such as expiration month/year on the registration page? Keep in mind that Selenium works on the DOM. So when you call `driver.find_element_by()`, Selenium will search through the entire page to find the element. But did you know that each `WebElement` also supports the `find_element_by` function? Thus to set a dropdown, find the dropdown element, then use that element to find the option you want. Then call `click`. So if we wanted to select the year '2017' from our expiration year dropdown, the code would look like this:

```
1 dd = self.driver.find_element_by_id('expiry_year')
2 option = dd.find_element_by_css_selector("option[value='2017']")
3 option.click()
```

In the second line we are only searching through the HTML elements that are children of the `expiry_year` drop down. This ensures we don't click on the wrong dropdown option.

Controlling Browser Options

If you remember back to the **MongoDB Time** chapter, we created a function to get the user's current location from the browser. So if we try to register while running a Selenium test, the browser will give us a popup asking if we want to allow the website to get our current location. We can use a browser profile to let the browser know that we always want to answer "yes" to that question. For Firefox, we have the Selenium object `FirefoxProfile` that allows us to control those sorts of things. Here is how we would use that to ensure we always get back a geolocation:

```
1 from selenium.webdriver.firefox.firefox_profile import
   FirefoxProfile
2
3 @classmethod
4 def setUpClass(cls):
```

```

5     profile = FirefoxProfile()
6     profile.set_preference('geo.prompt.testing', True)
7     profile.set_preference('geo.prompt.testing.allow', True)
8     cls.browser = webdriver.Firefox(profile)
9
10    cls.browser.implicitly_wait(10)
11    super(RegistrationTests, cls).setUpClass()

```

- **Line 1** - The `FirefoxProfile()` is the class that encapsulates all the browser specific settings for Firefox.
- **Lines 5 - 7** - Create the `FirefoxProfile()` and set the preferences that we need for our test.
- **Line 8** - Create a new Firefox webdriver, passing in the profile we just created.

Now you can control any of the hundreds of preferences that are available in Firefox.

You need to run some JavaScript code

The above code only works in certain versions of Firefox, and not at all for other browsers. But you can also mock out the geolocation call with a bit of JavaScript. To run JavaScript within Selenium you use the `webdriver.execute_script()` function. Here's an example that will set our Angular scope as if the HTML5 geolocation was called:

```

1 email_textbox = self.driver.get_element_by_id("id_email")
2
3 self.driver.execute_script('''
4     angular.element($("#id_email")).scope().geoloc = {
5         'coords': {'latitude':'1', 'longitude':'2'}
6     };
7     angular.element(document.body).injector().get('$rootScope').$apply();''')

```

How does that work?

A few things are going on here:

1. `driver.execute_script` is Selenium's way to call JavaScript from your GUI tests.
2. We call `angular.element($("#id_email")).scope()` to get the scope of our controller. Keep in mind that Angular can have multiple scopes on a page so we are explicitly saying get the scope for the element `#id_email`. Note we can use any element that is in the Angular scope we are looking for. On the registration page, this could be anything in the registration form.

3. As for the second command in our `execute_script` call, we are getting Angular's `rootScope` and calling `apply` (which you should always do when updating scope variables from outside of Angular).

After this bit of JavaScript is executed, we will have `$scope.geoloc` in our `registrationCtrl` set to the value we specified, thus we can do the geolocation for the user with whatever lat / long we decide to provide.

This is helpful as sometimes you may need to set certain Angular values to get this to work correctly in your Selenium tests.

Running tests on multiple browsers

Selenium supports many different browsers, including Firefox, Chrome, IE, Opera, Safari, PhantomJS, and Android / iOS mobile browsers out of the box. Using a different browser is as simple as creating a different webdriver. For example, to run against Chrome, simply change your `setUpClass` as such:

```
1 @classmethod
2 def setUpClass(cls):
3     cls.browser = webdriver.Chrome()
4     cls.browser.implicitly_wait(10)
5     super(RegistrationTests, cls).setUpClass()
```

Likewise just change the name to the browser that you want. Do note however that other browsers may have certain setup requirements. Details about what needs to be configured before using a particular browser can be found [here](#):

- [IE](#)
- [Chrome](#)
- [Opera](#)
- [Firefox](#)
- [Safari](#)
- [PhantomJS](#)
- [mobile browsers](#)

Limitations of Django's LiveServerTestCase

While Django's `LiveServerTestCase` and related `StaticLiveServerTestCase` are indeed helpful when it comes to testing with Selenium, they do have some limitations. The main limitation is due to the fact that `LiveServerTestCase` is single-threaded.

For many scenarios, like the login scenario we covered in this chapter, a single thread is all you need. However, this will lead to problems when you start to make calls to external servers. In particular, on our registration page when we try to make a call to Stripe both on the front and back end, this will cause strange errors that are difficult to debug. The solution would be to have `LiveServerTestCase` be multi-threaded or run in a completely separate process. This is a lot more difficult than one might imagine, so this hasn't been implemented yet in Django.

However, we can do a simplistic version of this on our own by creating a custom test runner (much like we did in the Mongo exercises). You will need this for the exercises, so here is the code for the custom test runner:

```
1 import os
2 import sys
3 import subprocess
4 from django.test.runner import DiscoverRunner
5 from django_ecommerce.guitest_settings import SERVER_ADDR
6
7
8 class LiveServerTestRunner(DiscoverRunner):
9
10     def setup_databases(self, **kwargs):
11         retval = super(LiveServerTestRunner,
12                        self).setup_databases(**kwargs)
13         self.spawn_server()
14         return retval
15
16     def spawn_server(self):
17         gui_settings = 'django_ecommerce.guitest_settings'
18         server_command = ['./manage.py', 'runserver',
19                          SERVER_ADDR, '--settings='+gui_settings]
20         self.server_p = subprocess.Popen(
21             server_command,
22             stdout=subprocess.PIPE,
```

```

23         close_fds=True,
24         preexec_fn=os.setsid
25     )
26     print("server process started up... continuing with test
           execution")
27
28     def kill_server(self):
29         try:
30             print("killing server process...")
31             os.killpg(os.getpgid(self.server_p.pid), 15)
32             self.server_p.wait()
33         except:
34             print("exception", sys.exc_info()[0])
35
36     def teardown_databases(self, old_config, **kwargs):
37         self.kill_server()
38         return super(LiveServerTestRunner, self).teardown_databases(
39             old_config, **kwargs)

```

Save it as *runner.py* in the “django_ecommerce” directory.

How does it work?

In a nutshell, we create a custom test runner, and after we have created the test database in `setup_databases`, we use `subprocess.Popen` to launch a Django development server in a different process. Then in `teardown_databases` before we delete the database we kill off the development server.

Timing is import here, because we have used a custom settings file (shown later) that will tell our development server to connect to our test database. So we must be sure the development server starts after the database is created, and before we try to drop the database.

The actual work of spawning the development server in a child process is housed in the `spawn_server` function. Here are the important pieces of that function:

- **Line 1** - We are using a settings file called `django_ecommerce.guitest_settings`, which houses the database connection definition and the address we want to use for our development server:

```

1  # Django settings for running gui tests with django_ecommerce
   project.
2  from .settings import *
3

```

```

4 DATABASES = {
5     'default': {
6         'ENGINE': 'django.db.backends.postgresql_psycopg2',
7         'NAME': 'test_django_db',
8         'USER': 'djangouser',
9         'PASSWORD': 'djangouser',
10        'HOST': 'localhost',
11        'PORT': '5432',
12    }
13 }
14
15 SERVER_ADDR = "127.0.0.1:9001"

```

Notice that at the start we import the existing settings to make sure we have all the settings we might need. Also notice how we have set the database name to `test_django_db`. This is the name of the database that our test runner will generate. Since both our test and our development server are using the same database, it makes it easier to set up test data in our tests. Bonus!

Make sure to create the new settings file as *guitest_settings.py*.

- **Lines 2 - 3** - Coming back to our `spawn_server` function, these two lines are the command line that we want to execute. Notice how we grab the `SERVER_ADDR` from our `guitest_settings` and also pass in `guitest_settings` as our custom settings file for our development server.
- **Lines 4 - 8** - This is how we spawn the child process. Basically we are setting it to run in a child process and capturing the `stdout` and `stderr` so they aren't displayed to the terminal. The last two options are necessary to properly clean up the process when we want to kill it later. For more information on the highly useful `subprocess` library, check out the [official docs](#)

Finally, after we spawn the server, our tests will run. When we have completed all of our tests, we want to kill the development server, which is done in the `kill_server` function. Here we are using the special function `os.killpg` because when we run the initial `manage.py runserver ...` command, it actually spawns a couple of processes, so `os.killpg` will kill the whole process tree.

With this test runner, we no longer need to use the `LiveServerTestCase` because our test runner is firing up the “LiveServer” for us. All we need to do is use the appropriate URL. We can get this in our tests by importing it directly from our custom settings files. A test using our `LiveServerTestRunner` could look like this:

```

1 from django_ecommerce.guitest_settings import SERVER_ADDR
2
3 class LoginTests(TestCase):
4
5     def setUp(self):
6         self.valid_test_user = User.create("tester",
7             "test@valid.com", "test", 1234)
8         self.sign_in_page = SignInPage(self.browser,
9             "http://" + SERVER_ADDR)

```

There are only two differences from our previous test case:

- We are inheriting from a regular `TestCase` and not `StaticLiveServerTestCase`, and
- We import the URL for the server from our `guitest_settings` because our regular `TestCase` doesn't have an associated `live_server_url`.

Make sure to update all of the GUI tests using the above format.

With that, you can run your Selenium tests without having to worry about the limitations of the `LiveServerTestCase`. To run your tests using this test runner, use the following command:

```

1 $ ./manage.py test ../tests --testrunner=runner.LiveServerTestRunner

```

Conclusion

As you have seen, Selenium can be a very powerful way to execute GUI tests with Python. It integrates well into Django and makes controlling the browser relatively straight-forward. We can easily find elements and act on them, and it has a simple API for waiting for things on the page to happen. While Selenium focus mainly on basic web functionality, it can also test web 2.0 AJAX functionality without too much trouble.

The difficulties you run into will likely be around Selenium + Angular; while it's not perfect, it is getting better every day. Don't be afraid of the `execute_script` function so you can modify your Angular variables / values as necessary. Take some time to learn the features of Selenium and see if it's the right GUI testing tool for you. Otherwise you can always look at the Angular specific testing tools, such as [Protractor](#) which are mentioned in the **What about javascript testing** section of this chapter.

Exercises

1. Write a Page Object for the Registration Page, and use it to come up with test for successful and unsuccessful registrations. You'll need to use the `LiveServerTestRunner` for the tests to execute successfully.

Chapter 19

Deploy

You, dear reader, have come a long way. We started with a simple login and registration page, and built the greatest Star Wars Fan Site the world has ever know. (Greatest being a somewhat subjective term.) Nevertheless, we built something cool. Pat your self on the back. You did it!

But we are not quite done, though. It's time to share the site with the world. That means sticking it up on a server somewhere, maybe getting a domain name and, well... deploying the site. This chapter will walk you through what you need to know to get a Django site deployed on a server in the cloud.

Where to Deploy

There are literally thousands of different web hosts that you could use to deploy your Django app. But basically all of them fall into three different types:

Server/VPS (Virtual Private Server)

The “standard” and most common type of hosting environment is one where you get access to a (usually) virtualized machine that has a basic OS installed on it and not much else. This give you the greatest control and flexibility, but it can be a bit more work to manage. This option also has the advantage of generally being the cheapest option. *Because of its ubiquity and affordability, this is the type of deployment we will cover in this chapter.*

Managed (Server/VPS)

Basically the same as a Server/VPS (which by contrast is often called Unmanaged Hosting), but the hosting company will provide support for OS upgrades, backups, etc... Because the company needs to support the server, there are generally some limitation as to what software and configurations you can install. Rackspace made a name for itself by providing Managed Servers/VPS.

PaaS

PaaS, or Platform as a Service, is a setup that attempts to remove the complexity of deployment from the user. Using a PaaS, such as Heroku, can make deployment as simple as pushing a commit to GitHub (after the prerequisite setup has been complete). Oftentimes a PaaS will offer additional features such as automatic scaling, backups, deployment, etc... They are a good option if you don’t want to worry about deployment, but you generally have very limited flexibility when using the PaaS. Example PaaS providers include [Heroku](#), [Amazon’s Elastic Beanstalk](#), and [Google App Engine](#).

It’s worth clicking some of the links above and exploring some of the options, but for now, we are going to stick with deployment on a Server/VPS. You will learn the most by going this route, and it’s always good to understand how deployment works.

What to Deploy

Since we are going to use a VPS that means we will start with a blank OS. To host a Django web app in production we will need three additional pieces of software (aka servers) to ensure the application will run without any issues in the more demanding environment (e.g., multiple users, more requests). The three (actually four) servers that we will need are:

1. Database Server - PostgreSQL and MongoDB, by now you're familiar with the database server, as you've been using it in development also.
2. Web Server / Proxy - In development we rely on the Django Test Server (i.e. `manage.py runserver`) to handle web requests and return results. The Django Test Server is great for development, but its not meant to be deployed into production. So since we are going into Production we are going to use a Server designed to be able to handle multiple simultaneous requests, serve static files (i.e., javascript, css, images, etc..), and generally not fall over when multiple users start making requests. The particular server we are going to use for this is called [Nginx](#). Nginx is a fast, powerful, and most importantly easy to configure HTTP server. In our setup it will be the first thing that responds when a user requests a web page, and it will either serve the static files directly to the user, or pass the request to our application server (see below) which will run the Python code and return the results to the user.
3. Application Server - Again in development `manage.py runserver` supplies us with both a web server (to handle web requests) and an application server that runs the Django code and returns the results for a particular request. In production we will use a separate application server called [Gunicorn](#). Gunicorn is a [WSGI server](#) written in Python that supports Django out of the box. This means we can use Gunicorn to host our Django application (much like the Django test server would). The difference is that Gunicorn is written with a production scenario in mind - its more secure and will perform better than Django's Test Server.

When we have all of these servers setup correctly. The flow of a user's web request will look like this:

That's how it all works. Now that we have a better idea of how the pieces fit together, let's jump right into the nitty-gritty of deployment.

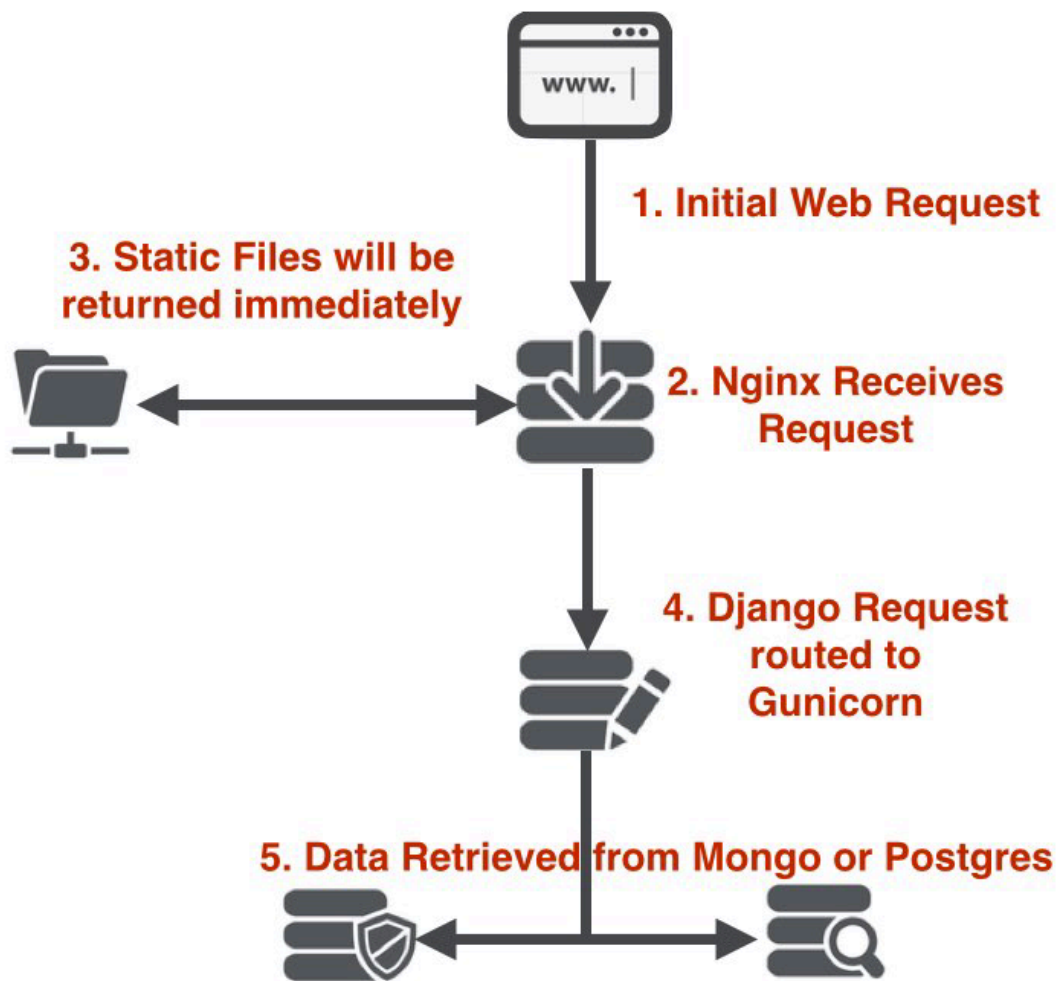


Figure 19.1: Deployment Description

Firing up the VPS

For the purpose of this chapter we are going to utilize an excellent VPS provider called [DigitalOcean](#) (but these instructions should work with most any VPS). If you want to use another provider, we are going to set up a VPS running Ubuntu 14.4 x64. If you want to use DigitalOcean, you can set up an account using this [link](#), which will give you \$10 in credit for free, which is enough to go through this tutorial and then some. Once you're signed up, follow these steps to get your VPS (what DigitalOcean calls a "Droplet") up and running:

DigitalOcean Setup

- Log into your DigitalOcean Portal
- Click the top button on the left-hand navigation panel that says "Create Droplet"
- Type in a host name; let's call it MEC.
- Under "select size" you can choose the first option, \$5/month.
- Select whatever region you want.
- Don't check anything in "Available Settings".
- For "Select Image", select Ubuntu 14.04 x64.
- Click "Create Droplet".

This will create a Droplet (a VPS) based on the settings you just specified. Setup should take just a couple of minutes, and then you can access your droplet through SSH; DigitalOcean will email you the root password and IP address. Once the creation is done, you have a freshly minted VPS and root access.

You can now do whatever you want.

Configuring the OS

Now how do we install the software we need on the VPS?

First, SSH into your Ubuntu machine as root and then run an upgrade and install the necessary packages

```
1 $ ssh root@<your-machines-ip>
2 $ apt-get update && apt-get upgrade
3 $ sudo apt-get install python-virtualenv libpq-dev python-dev
   postgresql postgresql-contrib nginx git libpython3.4-dev
```

Here we installed nginx, postgres, git and virtualenv, plus the necessary support packages. This will ensure we have the necessary operating system packages available. Now we need to configure things.

Postgres setup

Let's work back to front, and configure the database first. These are the same instructions covered in the Upgrade chapter when we installed postgres on our dev machine, but we will repeat them here to make things easier to find:

1. Verify installation is correct:

```
1 $ psql --version
```

And you should get something like the following:

```
1 psql (PostgreSQL) 9.3.7
```

2. Now that Postgres is installed, you need to set up a database user and create an account for Django to use. When Postgres is installed, the system will create a user named postgres. Let's switch to that user so we can create an account for Django to use:

```
1 $ sudo su postgres
```

3. Create a Django user in the database:

```
1 $ createuser -P.djangouser
```

Enter the password twice and remember it; for security reasons it's best not to use the same password you used in development. We will show you how to update your *settings.py* file accordingly later in the chapter.

4. Now using the postgres shell, create a new database to use with Django. **Note:** Don't type the lines starting with a **#**. These are comments for your benefit.

```
1 $ psql
2
3 # Once in the shell, type the following to create the database:
4 CREATE DATABASE django_db OWNER djangouser ENCODING 'UTF8';
5
6 # Then to quit the shell, type:
7 $ \q
```

5. Then we can set up permissions for Postgres to use by editing the file `/etc/postgresql/9.3/main/pg_hba.conf` with vim. Just add the following line to the end of the file:

```
1 local    django_db          djangouser    md5
```

```
1 Then save the file and exit the text editor. The above line says,
   "`djangouser` user can access the `django_db` database if they
   are initiating a local connection and using an md5-encrypted
   password".
```

6. Switch back to the root user:

```
1 $ exit
```

7. Finally restart the postgres service:

```
1 $ /etc/init.d/postgresql restart
```

This should restart postgres, and you should now be able to access the database. Check that your newly created user can access the database with the following command:

```
1 $ psql django_db djangouser
```

This will prompt you for the password; type it in, and you should get to the database prompt. You can execute any SQL statements that you want from here, but at this point we just want to make sure we can access the database, so just do a `\q` and exit out of the database shell. You're all set; postgres is working! You probably want to do a final exit from the command line to get back to the shell of your normal user.

If you do encounter any problems installing PostgreSQL, check the [wiki](#). It has a lot of good troubleshooting tips.

Set up Mongo

We have two databases now, so let's not forget to install mongo. Start off by installing the packages:

```
1 $ apt-get install mongo-db
```

That's all the configuration necessary. You can make sure it's working by typing `mongo` from the command prompt and making sure it drops you into the mongo shell.

Python Setup

We can start by creating a virtual environment. The following line will create one using Python 3:

```
1 $ virtualenv /opt/mec_env -p `which python3`
```

Switch to the `mec_env` directory, then clone our mec app from GitHub:

```
1 $ cd /opt/mec_env
2 $ git clone https://github.com/realpython/book3-exercises.git
   mec_app
```

NOTE Make sure you clone the app from your Github repo.

That will fetch the code from GitHub and put it into a directory called `mec_app`.

Let's activate our virtualenv:

```
1 $ source bin/activate
```

And then install all the necessary dependencies for our app:

```
1 $ cd mec_app
2 $ pip install -r requirements.txt
```

This will get all of our dependencies set up. However, we are going to need to make some changes in order for Django to run in our new “production” environment. We will do that in a later section called Django Setup.

Configuring Nginx

Nginx configuration just involves setting up a config file to get Nginx to listen on the incoming port and to have it pass requests onto Gunicorn, which is what will be hosting our Django app. By default, after you install Nginx (which we did in the earlier step where we executed `apt-get install nginx`) it will listen on port 80. So if you open up your browser and navigate to the URL of your DigitalOcean Droplet, you should see the following:

Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

Figure 19.2: nginx main screen

This means that Nginx is configured correctly. Now we need to configure it to serve our static files and to point to Gunicorn for requests that need to be routed to Django. To do that, we need to add a config file.

Nginx lives in the directory `/etc/nginx`. The main configuration file is called `/etc/nginx/nginx.conf`; this is where you make changes that will affect the entire system. For our purposes, you can safely leave this file alone, but it's worth having a look through it if you want to get a better idea of what's going on with Nginx.

We are going to add a new file called `/etc/nginx/sites-available/mec`, the contents of which should look like:

```
1 upstream app_server_djangoapp {
2     server localhost:8001 fail_timeout=0;
3 }
4
5 server {
6     listen 80;
7     server_name <<put in ipaddress or hostname of your server
        here>>;
```



```

8
9     access_log /var/log/nginx/mec-access.log;
10    error_log /var/log/nginx/mec-error.log info;
11
12    keepalive_timeout 5;
13
14    # path for static files
15    location /static {
16        autoindex on;
17        alias /opt/mec_env/static;
18    }
19
20    location / {
21        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
22        proxy_set_header Host $http_host;
23        proxy_redirect off;
24
25        if (!-f $request_filename) {
26            proxy_pass http://app_server_djangoapp;
27            break;
28        }
29    }
30 }

```

This will tell Nginx to serve static files from the directory `/opt/mec_env/static/`, which will be set up shortly. Other requests will be passed on to whatever is running on port 8001 on the server, which should be Gunicorn and our Django application.

To activate this configuration, do the following:

```

1 $ cd /etc/nginx/sites-enabled
2 $ ln -s ../sites-available/mec

```

The above will establish a symbolic link from the file we just created in the `/etc/nginx/sites-enabled` folder. This folder is where nginx will look for configuration so we need to do this so Nginx will see our configuration file and then things will work properly.

Next we need to deactivate the default Nginx configuration:

```

1 $ rm default

```

Then your Nginx configuration should be solid.

Configuring Gunicorn

Gunicorn is the server that will run your Django app. We can install it with pip:

```
1 $ cd /opt/mec_env
2 $ source bin/activate
3 $ pip install gunicorn
```

This will install Gunicorn. It takes a number of configuration options to work correctly, so let's create a new bash script that will run Gunicorn for us when we want to. Edit the file `/opt/mec_app/mec_env/deploy/gunicorn_start`:

```
1 #!/bin/bash
2
3 #Section 1 - Define Script Variables
4 NAME="Mec"                                # Name of the
5                                           application
6 DJANGODIR=/opt/mec_env/mec_app/django_ecommerce # Django
7                                           project directory
8 USER=www-data                             # the user to
9                                           run as
10 GROUP=www-data                            # the group to
11                                           run as
12 NUM_WORKERS=3                             # how many
13                                           worker processes should Gunicorn spawn
14 DJANGO_SETTINGS_MODULE=django_ecommerce.settings # which
15                                           settings file should Django use
16 DJANGO_WSGI_MODULE=django_ecommerce.wsgi    # WSGI module
17                                           name
18
19 #Section 2 - activate virtualenv and create environment variables
20 echo "Starting $NAME as `whoami`"
21
22 # Activate the virtual environment
23 cd $DJANGODIR
24 source ../../bin/activate
25 export DJANGO_SETTINGS_MODULE=$DJANGO_SETTINGS_MODULE
26 export PYTHONPATH=$DJANGODIR:$PYTHONPATH
27
28 # Section 3 - Start Gunicorn
```

```

23 exec gunicorn ${DJANGO_WSGI_MODULE}:application \
24     --name $NAME \
25     --workers $NUM_WORKERS \
26     --user=$USER --group=$GROUP \
27     --log-level=debug \
28     --bind=127.0.0.1:8001

```

Let's go through the script. From the comments we can see that it is broken up into three sections. The first section defines the variables that we will use to execute Gunicorn in the third section.

USER and GROUP are used to set Gunicorn to run as www-data, which is a user created during our installation that is most often used to run web/app server. This is for security concerns to limit access to the system.

In section 2 we activate our virtualenv (since that is where Gunicorn is installed) and export the environment variables needed. It's not uncommon to use this section to export environment variables that influence how Django runs, such as a new database connection for example.

The final section starts Gunicorn using the variables defined in section 1.

We can now test this out by running our gunicorn_start script from the command line. That can be done with the following commands:

```

1 $ cd /opt/mec_env/mec_app/deploy
2 $ chmod +x gunicorn_start      # make sure the script is executable
3 $ ./gunicorn_start

```

Note that the second line of the above listing only has to be executed once to ensure we can execute the file. When you run the script, you should see a bunch of output about your configuration, ending with:

```

1 [2015-02-15 23:02:35 -0500] [23401] [INFO] Starting gunicorn 19.2.1
2 [2015-02-15 23:02:35 -0500] [23401] [DEBUG] Arbiter booted
3 [2015-02-15 23:02:35 -0500] [23401] [INFO] Listening at:
   http://127.0.0.1:8001 (23401)
4 [2015-02-15 23:02:35 -0500] [23401] [INFO] Using worker: sync
5 [2015-02-15 23:02:35 -0500] [23410] [INFO] Booting worker with pid:
   23410
6 [2015-02-15 23:02:35 -0500] [23411] [INFO] Booting worker with pid:
   23411
7 [2015-02-15 23:02:35 -0500] [23412] [INFO] Booting worker with pid:
   23412

```

```
8 [2015-02-15 23:02:35 -0500] [23401] [DEBUG] 3 workers
9 [2015-02-15 23:02:36 -0500] [23401] [DEBUG] 3 workers
```

Of course the dates and the actual pid numbers will change, but it should have that basic output, and then it will continue to the line-

```
1 [2015-02-15 23:02:36 -0500] [23401] [DEBUG] 3 workers
```

-repeated over and over again. This is Gunicorn running. You can now go to your browser and navigate to the server IP, and you should no longer see the “welcome to nginx” screen. Instead you’ll probably see some ugly Django error. But that’s a good thing; it means you have Nginx talking to your Django application. We just haven’t finished configuring the Django application.

Before we switch to the Django application though, we need to finish up with Gunicorn. Right now if we execute `gunicorn_start`, that command will stop running as soon as we log out of our SSH session. What we want is for Gunicorn to run like a service, meaning it starts on system boot and keeps running forever. We can do that with Supervisor.

Configuring Supervisor

Supervisor is a Python program that is intended to keep other programs running. From the project’s own description: *Supervisor is a client/server system that allows its users to monitor and control a number of processes on UNIX-like operating systems.*

In other words, we use Supervisor to make sure other processes (like Gunicorn) keep running. Setup is straightforward. Start by installing it via:

```
1 $ apt-get install supervisor
```

Now create a configuration that will keep Gunicorn running. Edit the file `/etc/supervisor/conf.d/mec.conf` to include the following:

```
1 [program: mec]
2 directory = /opt/mec_env/mec_app
3 user = www-data
4 command = /opt/mec_env/mec_app/deploy/gunicorn_start
5 stdout_logfile = /opt/mec_env/log/supervisor-logfile.log
6 stderr_logfile = /opt/mec_env/log/supervisor-error-logfile.log
```

We will need to create the log directory for the log files:

```
1 $ mkdir /opt/mec_env/log
```

Finally, restart supervisor:

```
1 $ /etc/init.d/supervisor restart
```

It will probably take a few seconds. To see if it started, check the web page and see if you still have the ugly Django error. If so, you're good.

If not, check the main supervisor log file:

```
1 $ less /var/log/supervisor/supervisord.log
```

Or the application specific log files we just set up: sh

```
1 $ tail /opt/mec_env/log/*
```

Somewhere in there, it should have your problem.

Supervisor will now “manage” Gunicorn and ensure it keeps running all the time.

Now let's get the Django app itself configured correctly to work on the new environment.

Django Setup

There are a few things we need to modify in our `django_ecommerce/settings.py` file to get things to run on our production server. However, if we make the necessary changes for production to `django_ecommerce/settings.py` then things won't work in our development environment.

We need two `settings.py` files - one for development and one for production. There are two common ways to handle this. One is to actually create two separate `settings.py` files, and the other is to use environment variables. We will discuss using separate files here as it's probably the easiest when getting started.

Step 1. Create a `settings_prod.py` file

In our deployment directory, let's create a file called, you guessed it, `settings_prod.py` - it should look like this:

```
1 # turn off debugging in production
2 DEBUG = False
3
4 # settings for the production database
5 DATABASES = {
6     'default': {
7         'ENGINE': 'django.db.backends.postgresql_psycopg2',
8         'NAME': 'django_db',
9         'USER': 'djangouser',
10        'PASSWORD': '<< password you set for you new database>>',
11        'HOST': 'localhost',
12        'PORT': '5432',
13    }
14 }
15
16 # allowed hosts for our production site
17 ALLOWED_HOSTS = [<<ip address of your server>>']
18
19 STATIC_ROOT = '/opt/mec_env/static/'
20 MEDIA_ROOT = '/opt/mec_env/media/'
```

This file will be used to overwrite the settings in your development `settings.py` file. The settings should be self-explanatory; we are configuring things to work in our production server.

Also, be sure that both `/opt/mec_env/static` and `/opt/mec_env/media` exist on the server with the following command:

```
1 $ mkdir /opt/mec_env/static
2 $ mkdir /opt/mec_env/media
```

Step 2. Link to the new settings file.

Now to make Django aware of the new `settings_prod.py` file, add the following lines to the end of your existing `django_ecommerce/settings.py` file:

```
1 try:
2     from .settings_prod import *
3 except ImportError:
4     pass
```

What does this do? It tries to import `settings_prod` from the same directory as our normal `settings.py` file. Since `settings_prod` is imported as the last statement in `settings.py`, whatever values are set in `settings_prod.py` will overwrite the values already set in `settings.py`.

Step 3. Move `settings_prod.py` into place

To make things work (on our server), copy `settings_prod.py` into the correct place:

```
1 $ cd /opt/mec_env/mec_app
2 $ cp deploy/settings_prod.py django_ecommerce/django_ecommerce/
```

This will move the file into place, and then when Django starts up it will read our production settings. To get Django to restart, just restart Gunicorn, which is done using Supervisor:

```
1 $ /etc/init.d/supervisor restart
```

How do we know it worked? Reload your web page, and you'll see a 500 error, no more debug error messages, because we have set `DEBUG=False`. Now to get rid of those pesky 500 errors.

Step 4. Run your migrations

We are getting the 500 error because we don't have our database set up correctly. Let's fix that:

```
1 $ cd /opt/mec_env/mec_app/django_ecommerce
2 $ source ../../bin/activate
3 $ ./manage.py migrate
```

Now reload the web page, and no more 500 error, yay! But, the page looks very ugly, boo!

Step 5. Put the static files in the correct place

The page looks bad because Django can't find our static files. Remember in our `settings_prod.py` we set the static root to be `/opt/mec_env/static`. But there is nothing there. Django has a handy built-in command that will move all of our static files to that directory for us, though:

```
1 $ ./manage.py collectstatic
```

The command will warn you that it's going to overwrite anything in that directory. Say 'yes', and then it should copy all the files, finishing up with a message like:

```
1 414 static files copied to '/opt/mec_env/static'
```

Now reload the web page and... voilà (spoken with a strong French accent)! And there you have it. You are deployed into production! Congratulations.

So that's it, right? Wrong... we are just getting started, actually. Now that we have the deployment working, we need to-

1. Keep track of all the config files
2. Create a deployment script so that we can update production with a single script when the time comes to make updates.

Configuration as Code

A term often used in DevOps circles, “configuration as code”, means keeping track of your configuration files so that deployment can be automated, thus reducing the dreaded “it worked on my machine” issue.

For our setup, this means gathering the configuration files for Nginx, Gunicorn, and Supervisor, plus our `settings_prod.py`, and storing them in GitHub. We started to do this already, but let’s make sure we have everything in the correct place.

We want a directory structure like this:

```
1 deploy
2   gunicorn_start
3   nginx
4     sites-available
5       mec
6   settings_prod.py
7   supervisor
8     mec.conf
```

As a review of the deployment process, here are the contents of each of those files:

gunicorn_start

```
1 #!/bin/bash
2
3 NAME="Mec"                                # Name of the
   application                               application
4 DJANGO_DIR=/opt/mec_env/mec_app/django_ecommerce # Django
   project directory                         project directory
5 USER=www-data                             # the user to
   run as                                   run as
6 GROUP=www-data                             # the group to
   run as                                   run as
7 NUM_WORKERS=3                             # how many
   worker processes should Gunicorn spawn  worker processes should Gunicorn spawn
8 DJANGO_SETTINGS_MODULE=django_ecommerce.settings # which
   settings file should Django use         settings file should Django use
9 DJANGO_WSGI_MODULE=django_ecommerce.wsgi      # WSGI module
   name                                    name
10
11 echo "Starting $NAME as `whoami`"
```

```

12
13 # Activate the virtual environment
14 cd $DJANGODIR
15 source ../../bin/activate
16 export DJANGO_SETTINGS_MODULE=$DJANGO_SETTINGS_MODULE
17 export PYTHONPATH=$DJANGODIR:$PYTHONPATH
18
19 # Start your Django Unicorn
20 exec gunicorn ${DJANGO_WSGI_MODULE}:application \
21     --name $NAME \
22     --workers $NUM_WORKERS \
23     --user=$USER --group=$GROUP \
24     --log-level=debug \
25     --bind=127.0.0.1:8001

```

nginx/sites-available/mec

```

1 upstream app_server_djangoapp {
2     server localhost:8001 fail_timeout=0;
3 }
4
5 server {
6     listen 80;
7     server_name 128.199.202.178;
8
9     access_log /var/log/nginx/mec-access.log;
10    error_log /var/log/nginx/mec-error.log info;
11
12    keepalive_timeout 5;
13
14    # path for static files
15    location /static {
16        autoindex on;
17        alias / opt/mec_env/static;
18    }
19
20    location / {
21        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
22        proxy_set_header Host $http_host;
23        proxy_redirect off;
24

```

```

25         if (!-f $request_filename) {
26             proxy_pass http://app_server_djangoapp;
27             break;
28         }
29     }
30 }

```

supervisor/mec.conf

```

1 [program: mec]
2 directory = /opt/mec_env/mec_app
3 user = www-data
4 command = /opt/mec_env/mec_app/deploy/gunicorn_start
5 stdout_logfile = /opt/mec_env/log/supervisor-logfile.log
6 stderr_logfile = /opt/mec_env/log/supervisor-error-logfile.log

```

settings_prod.py

```

1 # turn off debugging in production
2 DEBUG=False
3
4 # settings for the production database
5 DATABASES = {
6     'default': {
7         'ENGINE':
8             'django.db.backends.postgresql_psycopg2',
9         'NAME': 'django_db',
10        'USER': 'djangousr',
11        'PASSWORD': 'djangousr',
12        'HOST' : 'localhost',
13        'PORT' : '5432',
14    }
15 }
16
17 # allowed hosts for our production site
18 ALLOWED_HOSTS = ['128.199.202.178']
19
20 STATIC_ROOT = '/opt/mec_env/static/'
21 MEDIA_ROOT = '/opt/mec_env/media/'

```

That's it for our configuration. Once you have all those files in the correct place, commit it all and push it up to your GitHub account.

Let's script it

At the point, we have a production environment working, and we are keeping track of all the configuration for our production environment. But next time we need to update the configuration, move to a different server, or simply deploy some new code, there is a good chance that we will probably have forgotten the steps necessary for installation. Plus, it's annoying to have to manually type in all that stuff if we just want to update production.

Let's fix that with an auto-deployment script.

An auto-what script?

It's a simple Python file that will SSH into our server, grab the latest code, make sure everything is configured right, and redeploy the application. Sounds like a lot? Don't worry; it's actually not too much work thanks to a nice Python library called [Fabric](#).

Fabric allows us to script SSH and shell commands using Python. What could be better?

Writing your first Fabric script

First thing to do is of course install Fabric. Unfortunately as of this writing, Fabric is not quite ready for Python 3, although it is close. So we will use a fork that does work on Python 3. To use the fork, update your `requirements.txt` file and add this line to the end:

```
1 -e git+https://github.com/pashinin/fabric.git@p33#egg=Fabric
2 six==1.9.0
```

Then you can install it with:

```
1 $ pip install -r requirements.txt
```

Once you have Fabric installed, we can use it to create a script that will automate our deployment. Let's look at a simple version of the script first (we will expand upon this as we go on).

Start by creating `deploy/fabfile.py`:

```
1 from fabric.api import env, cd, run, prefix, lcd, settings, local
2
3 env.hosts = ['<<domain or ip of your server>>']
4 env.user = 'root'
5
```

```

6
7 def update_app():
8     with cd("/opt/mec_env/mec_app"):
9         run("git pull")
10    with cd("/opt/mec_env/mec_app/django_ecommerce"):
11        with prefix("source /opt/mec_env/bin/activate"):
12            run("pip install -r ../requirements.txt")
13            run("./manage.py migrate")
14            run("./manage.py collectstatic")

```

Let's look at what's going on, line-by-line:

- **Line 1** - import the fabric API
- **Line 3** - specify the host we are going to run on by IP or domain; if there are multiple items in this list then Fabric will apply the same function to all servers in the list
- **Line 4** - `env.user` specifies the username you want to SSH with. If you don't provide this, Fabric will try to use the username you are logged in as on your local machine
- **Line 7** - Fabric works by executing a number of commands over SSH; this is the function that has a series of commands necessary to update the application on our remote machine
- **Line 8** - the `cd` function does just what you think; it changes to a directory; here we use it in a context manager, so each subsequent command will execute inside that directory
- **Line 9** - execute `git pull` to get the latest code from GitHub; because this line is inside the `cd` context manager, Fabric will actually be executing a command like: `cd /opt/mec_env/mec_app && git pull`
- **Line 10** - another `cd` context manager, this time to the directory where `manage.py` lives
- **Line 11** - `prefix` is another context manager that says, "execute this command just before you execute any other command"; here we are using this to ensure we have our virtualenv activated before we execute any of our commands
- **Line 12** - install any new requirements listed in `requirements.txt`; because this command is nested inside of two context managers, the ultimate command that will be executed is: `cd /opt/mec_env/mec_app/django_ecommerce && source /opt/mec_env/bin/activate && pip install -r ../requirements.txt`
- **Line 13** - run migrations
- **Line 14** - run `collectstatic`; on this line because `collectstatic` prompts the user to enter yes or no, Fabric will detect this and prompt you on your local machine, which makes it easier as sometimes you need to interact with the remote machine

Keep in mind that we can tweak the last couple of commands so they never ask us any questions:

```
1 run("./manage.py migrate --noinput")
2 run("./manage.py collectstatic --noinput")
```

The `--noinput` argument is provided by Django for just this type of situation where you always want to answer “yes” and not be prompted.

In addition, if you don’t want to be prompted with a password to log into your server, you can set up an SSH keypair so that no password is required. For DigitalOcean [this how-to](#) will describe just how that’s done.

Once the script is done, use the `fab` command to run it. Make sure you’re in the deploy directory, then run:

```
1 $ fab update_app
```

And off to the races you go!

As you can see, scripting a basic deployment with Fabric is pretty straightforward. We probably want to add one more function to update the config files (for Nginx, Supervisor, and Django) automatically just in case they ever change, but I’ll leave that as an exercise for the reader.

Continuous Integration

By automating the deployment of your application with Fabric, you not only save time, but you also help to prevent errors during deployment by ensuring steps aren't missed or forgotten. Taken to the next level, if there were a way to ensure that all of our tests were run and passed before any deployment happened, we would be adding even more quality assurance. If we then performed the step of running all of our tests (plus potentially deploying) each time we made a commit to our codebase, we would be practicing continuous integration.

The term, coined by [Grady Booch](#) back in the [Extreme Programming](#) days, and often referred to as simply **CI**, literally refers to integrating the various branches of code (e.g., each developer's local branch) with the mainline code daily. In git this translates to:

- `git commit` any local changes
- `git pull` fix any merge conflicts
- `git push`

But with automated unit and integration tests, we can do better by adding test runs into the mix.

For a single developer, we can actually build a continuous integration system like this with Fabric and a few more lines of code. Let's add a function called `integrate` to our `deploy/fabfile.py`:

```
1 def integrate():
2     with lcd("../django_ecommerce/"):
3         local("./manage.py test ../tests/unit")
4
5         with settings(warn_only=True):
6             local("git add -p && git commit")
7
8             local("git pull")
9             local("./manage.py test ../tests")
10            local("git push")
```

And what does all that do?

- **Line 2** - `lcd` changes the directory on the local machine; here we make sure we are in the correct directory for running our tests.

- **Line 4** - normally if a line fails, Fabric will abort execution; for the `git add / git commit` line, it will fail if we haven't made any local changes, but maybe we still want to integrate, so this line tells Fabric to spew out a warning and continue execution
- **Line 8** - after we commit, let's get the latest from origin (aka GitHub); this is the "integration" step.
- **Line 9** - after we integrate, run the full suite of tests to make sure everything is still working
- **Line 10** - if all of our tests pass, push all changes back to GitHub

Then to practice continuous integration, instead of running `git push / git pull` manually, every time you're ready to push a change, just execute:

```
1 $ fab integrate update_app
```

That will run the two functions and your fabfile, and you're cooking up some CI goodness.

Continuous Integration Tools

In addition to the simple Fabric-based CI we just implemented, which I'm dubbing the simplest CI system on earth, there are several third-party CI servers, the entire purpose of which is to listen to your code repository and automatically run CI builds every time some new code appears in the repository. There are many tools out there that do this, so I don't intend to cover them all. Rather I'll point you to a couple of great resources to get more information about CI system and how to work with them.

1. The second book in the Real Python series [Web Development for Python](#) has a full chapter on Travis CI, one of the most popular CI servers out there.
2. The article [Docker in Action: Fitter, Happier, More Productive](#) discusses CI setup with Docker.

Conclusion

We covered a lot in this chapter. We went through the complete set up of a “virgin” VPS, getting it set up with Nginx, Gunicorn, postgresql, MongoDB and Django. Then, as if that wasn’t enough, we showed you how to automate the entire process. And we even looked at some suggestions on how to implement continuous integration to ensure your app is always well tested, integrated and behaving nicely. We came a long way in this chapter; you should be proud of yourself for getting through it.

Furthermore, you should now have a better understanding of what goes into deployment, how to configure the various servers and hopefully a bit about how to troubleshoot problems when they arise. Now you know enough to be dangerous.

Exercises

1. While a VPS represents the most common case for deploying a Django app, there are several others. Have a read through the following articles to get an understanding of how to deploy using other architectures:
 - **Amazon Elastic Beanstalk** - [This article](#) details a step-by-step guide on how to deploy a Django app to AWS with Elastic Beanstalk.
 - **Heroku** - How to [Migrate your Django Project to Heroku](#).
 - **Dokku** - This is basically Heroku on the cheap. Find out how to get Django set up with Dokku [here](#).
 - **Docker** - While mentioned previous in the chapter, this [article](#) is an excellent write-up on how to use Docker for deployment + development + CI.
2. Didn't think you were going to get out of this chapter without writing any code, did you? Remember back in the Fabric section when we talked about creating another function to update all of your configuration files? Well, now is the time to do that. Create a function called `update_config()` that automatically updates your Nginx, Supervisor and Django config / setting files for you.

When the function is complete you should be able to execute:

```
1 $ fab integrate update_app update_config
```

Bonus points if you can get it all to work so the user only has to type:

```
1 $ fab ci
```

Chapter 20

Conclusion

That's about it. You should now have a fully functioning website, deployed to production, with tests and auto-deployment and all kinds of goodness! Hurray, you made it. It was a lot of hard work and dedication, but doesn't it feel great knowing that you have come this far? By now you should have a good understanding of what it takes to make a web site. We covered a ton of stuff - from mixins to transactions, from inclusion tags to form handling, from Mongo to Python 3 to Angular to promises... and a whole lot more in between.

While we have tried to cover a great deal, there is so much out there in the world of computer programming that we have barely scratched the surface. But that's okay. You don't need to know everything - nobody does. In fact my advice if you intend to continue growing as a software developer is to go deep not broad. There are far too many generalists / script kiddies / developers who only know enough to be dangerous. Hopefully we have not only geared you up with a well-rounded knowledge about how to develop web pages but also an understanding of how things work architecturally, what informs the choices that developers make when structuring code or web apps, and the tradeoffs associated with any sizable development effort.

At this point if you just start thinking more about the structure and layout of your data and code, then I think I've done my job. Keep coding. Keep learning. And always strive for a deeper understanding of why code works the way it does. That curiosity and subsequent exploration is what leads to true software craftsmanship.

Thank you very much for allowing me to share what knowledge I have with you. I do hope it was a fulfilling and pleasurable experience. I'm always eager, to hear how things went. Let me know what you build, what you learn, or what problems you encounter along your journey. Until then, all the best and do enjoy your coding adventures (even when the code doesn't do what you tell it to.)

Jeremy Johnson, Michael Herman, and Fletcher Heisler

@j1z0 / jeremy@realpython.com

Chapter 21

Appendix A - Solutions to Exercises

Software Craftsmanship

Exercise 1

Question: Our URL routing testing example only tested one route. Write tests to test the other routes. Where would you put the test to verify the pages/ route? Do you need to do anything special to test the admin routes?

There are, of course, many solutions to this. Let's add a `ViewTesterMixin()` class to `payments/test` in order to help with the view tests so we're not repeating ourselves:

```
1 from .views import sign_in, sign_out
2 from django.core.urlresolvers import resolve
3
4
5 class ViewTesterMixin(object):
6
7     @classmethod
8     def setupViewTester(cls, url, view_func, expected_html,
9                        status_code=200,
10                       session={}):
11         request_factory = RequestFactory()
12         cls.request = request_factory.get(url)
13         cls.request.session = session
14         cls.status_code = status_code
15         cls.url = url
```

```

16     cls.view_func = staticmethod(view_func)
17     cls.expected_html = expected_html
18
19     def test_resolves_to_correct_view(self):
20         test_view = resolve(self.url)
21         self.assertEqual(test_view.func, self.view_func)
22
23     def test_returns_appropriate_response_code(self):
24         resp = self.view_func(self.request)
25         self.assertEqual(resp.status_code, self.status_code)
26
27     def test_returns_correct_html(self):
28         resp = self.view_func(self.request)
29         self.assertEqual(resp.content, self.expected_html)

```

And it's meant to be used like this:

```

1 class SignInPageTests(TestCase, ViewTesterMixin):
2
3     @classmethod
4     def setUpClass(cls):
5         html = render_to_response(
6             'sign_in.html',
7             {
8                 'form': SigninForm(),
9                 'user': None
10            }
11        )
12
13        ViewTesterMixin.setupViewTester(
14            '/sign_in',
15            sign_in,
16            html.content
17        )

```

The `ViewTesterMixin()` creates some default tests that can be run against most standard tests.

In our case, we use it for all the view functions in the payments app. These are the same functions that we previously implemented to test our view function, except these are all placed in a base class.

So, all you have to do is call the `setUpViewTest()` class method, which is meant to be called from the derived `setUpClass` class method. Once you the setup is taken care of, the `ViewTesterMixin()` will simply perform the basic tests for routing, checking return codes, and making sure you're using the appropriate template.

Once setup, you can implement any other tests you like.

he great thing about using Mixins for testing is that they can reduce a lot of the boilerplate (e.g., common test routines) that you often run against Django's system.

As for the second part of the question. Don't bother testing the admin views as they are generated by Django, so we can assume they are correct because they are already tested by the Django Unit Tests.

For the pages route, that is actually not that easy to do in Django 1.5. If you are sticking to the [out-of-the-box test discovery](#), there isn't a base test file. So we'll leave this out until we get to the chapter on upgrading to Django 1.8, and then you can see how the improved test discovery in Django 1.8 (which was introduced in Django 1.6) will allow you to better structure your tests to model the application under test.

Exercise 2

Question: Write a simple test to verify the functionality of the `sign_out` view. Do you recall how to handle the session?

```
1 class SignOutPageTests(TestCase, ViewTesterMixin):
2
3     @classmethod
4     def setUpClass(cls):
5         ViewTesterMixin.setUpViewTester(
6             '/sign_out',
7             sign_out,
8             "", # a redirect will return no html
9             status_code=302,
10            session={"user": "dummy"},
11        )
12
13     def setUp(self):
14         #sign_out clears the session, so let's reset it overtime
15         self.request.session = {"user": "dummy"}
```

There are two things to note here.

1. We are passing in `session = {"user": "dummy"}` to our `setUpViewTest()` function.
2. And if we look at the third and fourth lines of `setUpViewTester()`:

```
1 cls.request = request_factory.get(url)
2 cls.request.session = session
```

We can see that it shoves the session into the request returned by our request factory. If you don't recall what the `RequestFactory()` is all about, have a quick re-read of the *Mocks, fakes, test doubles, dummy object, stubs* section of the *Testing Routes and Views* chapter.

Exercise 3

Question: Write a test for the `contact/models`. What do you really need to test? Do you need to use the database backend?

If you recall from the *Testing Models* chapter, we said to only test the functionality you write for a model. Thus, for the `ContactForm` model we only need to test two things:

1. The fact that the model will return the email value as the string representation.
2. The fact that the queries to `ContactForms` are always ordered by the timestamp.

Note that there is a bug in the `ContactForm()` class; the `class` Meta needs to be indented so it is a member class of `ContactForm`. Make sure to update before running the tests!

Here is the code for the test:

```
1 from django.test import TestCase
2 from .models import ContactForm
3 from datetime import datetime, timedelta
4
5
6 class UserModelTest(TestCase):
7
8     @classmethod
9     def setUpClass(cls):
10         ContactForm(email="test@dummy.com", name="test").save()
11         ContactForm(email="j@j.com", name="jj").save()
12         cls.firstUser = ContactForm(
```



```

13         email="first@first.com",
14         name="first",
15         timestamp=datetime.today() + timedelta(days=2)
16     )
17     cls.firstUser.save()
18     #cls.test_user=User(email="j@j.com", name='test user')
19     #cls.test_user.save()
20
21     def test_contactform_str_returns_email(self):
22         self.assertEqual("first@first.com", str(self.firstUser))
23
24     def test_ordering(self):
25         contacts = ContactForm.objects.all()
26         self.assertEqual(self.firstUser, contacts[0])

```

Exercise 4

Question: QA teams are particularly keen on ‘boundary checking’. Research what it is, if you are not familiar with it, then write some unit tests for the CardForm from the payments app to ensure that boundary checking is working correctly.

First, what’s boundary checking? Check out the Wikipedia [article](#).

To accomplish boundary checking we can make use of our FormTesterMixin() and have it check for validation errors when we pass in values that are too long or too short.

Here is what the test might look like:

```

1     def test_card_form_data_validation_for_invalid_data(self):
2         invalid_data_list = [
3             {
4                 'data': {'last_4_digits': '123'},
5                 'error': (
6                     'last_4_digits',
7                     [u'Ensure this value has at least 4 characters
8                       (it has 3).']
9                 )
10            },
11            {
12                'data': {'last_4_digits': '12345'},
13                'error': (

```

```

13         'last_4_digits',
14         [u'Ensure this value has at most 4 characters
15           (it has 5).']
16     )
17 }
18 ]
19
20 for invalid_data in invalid_data_list:
21     self.assertFormError(
22         CardForm,
23         invalid_data['error'][0],
24         invalid_data['error'][1],
25         invalid_data["data"]
26     )

```

Before moving on, be sure that all your new tests pass: `./manage.py test`.

Test Driven Development

Exercise 1

Question: If you really want to get your head around mocks, try mocking out the `test_registering_user_twice_cause_error_msg()` test. Start by mocking out the `User.create` function so that it throws the appropriate errors.

HINT: [this documentation](#) is a great resource and the best place to start. In particular, search for `side_effect`.

The first part of this exercise is relatively straightforward. Use the patch decorator with the `side_effect` parameter like this:

```
1 @mock.patch('payments.models.User.save', side_effect=IntegrityError)
2 def test_registering_user_twice_cause_error_msg(self, save_mock):
3     # create the request used to test the view
4     self.request.session = {}
5     #...snipped the rest for brevity...#
```

The `side_effect` parameter says, “When this function is called, raise the `IntegrityError` exception.” Also note that since you are manually throwing the error, you don’t need to create the data in the database, so you can remove the first couple of lines from the test function as well:

```
1 #create a user with same email so we get an integrity error
2 user = User(name='pyRock', email='python@rocks.com')
3 user.save()
```

Run the test. Did you get an error?

```
1 Creating test database for alias 'default'...
2 .....F....
3 .
4 .....
5 =====
6 FAIL: test_registering_user_twice_cause_error_msg
   (payments.tests.RegisterPageTests)
7 -----
8 Traceback (most recent call last):
9   File
   "/Users/michaelherman/.virtualenvs/chp9/lib/python2.7/site-packages/mock.py",
   line 1201, in patched
```

```

10     return func(*args, **kwargs)
11 File
12     "/Users/michaelherman/Documents/repos/realpython/book3-exercises/_chapters/ch
13     line 362, in test_registering_user_twice_cause_error_msg
14     self.assertEqual(len(users), 1)
15 AssertionError: 0 != 1
16 -----
17 Ran 20 tests in 0.295s
18 FAILED (failures=1)
19 Destroying test database for alias 'default'...

```

What else do you need to change - and why?

Question: Want more? Mock out the UserForm as well. Good luck.

This part is a bit more involved. There are many ways to achieve this with the mock library, but the simplest way is to create a class and use the patch decorator to replace the UserForm with the class you created specifically for testing.

Here is what the class looks like:

```

1 def get_MockUserForm(self):
2     from django import forms
3
4     class MockUserForm(forms.Form):
5
6         def is_valid(self):
7             return True
8
9         @property
10        def cleaned_data(self):
11            return {
12                'email': 'python@rocks.com',
13                'name': 'pyRock',
14                'stripe_token': '...',
15                'last_4_digits': '4242',
16                'password': 'bad_password',
17                'ver_password': 'bad_password',
18            }
19

```

```

20     def addError(self, error):
21         pass
22
23     return MockUserForm()

```

A few quick points:

1. The class is wrapped in a function so it's easy to reach.
2. Only the methods/properties that will be called by this test are created. This, to get the tests to behave right, we only used the methods need to do the as minimal amount of work as possible.

Once the Mock class is created, it can override the “real” UserForm class with the mock class by using the patch decorator.

The full test now looks like this:

```

1  @mock.patch('payments.views.UserForm', get_MockUserForm)
2  @mock.patch('payments.models.User.save',
3  side_effect=IntegrityError)
4  def test_registering_user_twice_cause_error_msg(self,
5  save_mock):
6
7      #create the request used to test the view
8      self.request.session = {}
9      self.request.method = 'POST'
10     self.request.POST = {}
11
12     #create the expected html
13     html = render_to_response(
14         'register.html',
15         {
16             'form': self.get_MockUserForm(),
17             'months': list(range(1, 12)),
18             'publishable': settings.STRIPE_PUBLISHABLE,
19             'soon': soon(),
20             'user': None,
21             'years': list(range(2011, 2036)),
22         }
23     )

```

```

23     #mock out stripe so we don't hit their server
24     with mock.patch('stripe.Customer') as stripe_mock:
25
26         config = {'create.return_value': mock.Mock()}
27         stripe_mock.configure_mock(**config)
28
29     #run the test
30     resp = register(self.request)
31
32     #verify that we did things correctly
33     self.assertEqual(resp.content, html.content)
34     self.assertEqual(resp.status_code, 200)
35     self.assertEqual(self.request.session, {})
36
37     #assert there is no records in the database.
38     users = User.objects.filter(email="python@rocks.com")
39     self.assertEqual(len(users), 0)

```

Notice that the patched object is `payments.views.UserForm` *NOT* `payments.forms.UserForm`.

The trickiest thing about the mock library is knowing where to patch. Keep in mind that you want to patch the object used in your SUT (System Under Test).

In this case since we're testing `payments.views.registration`, we want the `UserForm` imported in the view module. If you're still unclear about this, re-read [this section of the docs](#).

Exercise 2

Question: As alluded to in the conclusion, remove the customer creation logic from `register()` and place it into a separate `CustomerManager()` class. Re-read the first paragraph of the conclusion before you start, and don't forget to update the tests accordingly.

The solution for this is pretty straightforward: Grab all the Stripe logic and wrap it in a class. This example is a bit contrived because at this point in our application it may not make a lot of sense to do this, but the point here is about TDD and how it helps you with refactoring. To make this change, the first thing you would do is create a simple test to help design the new `Customer()` class:

```

1 class CustomerTests(TestCase):
2

```

```

3     def test_create_subscription(self):
4         with mock.patch('stripe.Customer.create') as create_mock:
5             cust_data = {'description': 'test user', 'email':
6                           'test@test.com',
7                           'card': '4242', 'plan': 'gold'}
8             Customer.create(**cust_data)
9
9             create_mock.assert_called_with(**cust_data)

```

This test says, “The `Customer.create` function is used to call Stripe with the arguments passed in.”

You could implement a simple solution to that to place in `payments.views` like this:

```

1 class Customer(object):
2
3     @classmethod
4     def create(cls, **kwargs):
5         return stripe.Customer.create(**kwargs)

```

Then make sure to update the following import in `payments.tests`:

```

1 from payments.views import soon, register, Customer

```

This is probably the simplest way to achieve this. (Note: Instead of using `**kwargs` you could enumerate the names.) This will pass the test, but the next requirement is to support both subscription and one_time payments.

To do that, let’s update the tests:

```

1 class CustomerTests(TestCase):
2
3     def test_create_subscription(self):
4         with mock.patch('stripe.Customer.create') as create_mock:
5             cust_data = {'description': 'test user', 'email':
6                           'test@test.com',
7                           'card': '4242', 'plan': 'gold'}
8             Customer.create("subscription", **cust_data)
9
9             create_mock.assert_called_with(**cust_data)
10
11     def test_create_one_time_bill(self):
12         with mock.patch('stripe.Charge.create') as charge_mock:

```

```

13         cust_data = {'description': 'email',
14                       'card': '1234',
15                       'amount': '5000',
16                       'currency': 'usd'}
17         Customer.create("one_time", **cust_data)
18
19         charge_mock.assert_called_with(**cust_data)

```

Here, you could have created two separate functions, but we designed it with one function and passed in the type of billing as the first argument. You could make both of these test pass with a slight modification to the original function:

```

1 class Customer(object):
2
3     @classmethod
4     def create(cls, billing_method="subscription", **kwargs):
5         if billing_method == "subscription":
6             return stripe.Customer.create(**kwargs)
7         elif billing_method == "one_time":
8             return stripe.Charge.create(**kwargs)

```

And there you have it.

Next, let's actually substitute the new `Customer()` class in place of `stripe.Customer` in our `register` function:

```

1 customer = Customer.create(
2     email=form.cleaned_data['email'],
3     description=form.cleaned_data['name'],
4     card=form.cleaned_data['stripe_token'],
5     plan="gold",
6 )

```

Finally, update our test cases so that we do not reference Stripe at all. Here's an example of an updated test case:

```

1 @mock.patch('payments.views.Customer.create')
2 @mock.patch.object(User, 'create')
3 def test_registering_new_user_returns_successfully(self,
4     create_mock, stripe_mock):

```

Not too much different. We changed the first patch to `payments.views.Customer.create` instead of `'stripe'`. In fact, we would have left it as is, and the test would have still passed - but

this way we keep our test ignorant of the fact that we are using Stripe in case we later want to use something other than Stripe to process payments or change how we interact with Stripe.

And of course you could continue factoring out the Stripe stuff from the `edit()` function as well if you wanted, but we'll stop here.

Finally, run all of your tests again to ensure we didn't break any functionality in another app:

```
1 ./manage.py test
```

They should all pass:

```
1 reating test database for alias 'default'...
2 .....
3 .
4 .....
5 -----
6 Ran 555 tests in 11.508s
7
8 OK (skipped=1, expected failures=1)
```

Cheers!

Bootstrap 3 and Best Effort Design

Exercise 1

Questions: **Bootstrap is a front-end framework, and although we didn't touch much on it in this chapter, it uses a number of CSS classes to insert things on the page, making it look nice and provide the responsive nature of the page. It does this by providing a large number of classes that can be attached to any HTML element to help with placement. All of these capabilities are described on the [Bootstraps CSS page](#). Have a look through it, and then let's put some of those classes to use.

- 1 - In the main carousel, the text, "Join the Dark Side" on the Darth Vader image, blocks the image of Darth himself. Using the Bootstrap / carousel CSS, can you move the text **and** sign up button to the left of the image so as to **not** cover Lord Vader?
- 2 - If we do the above change, everything looks fine until we view things on a phone (**or** make our browser really small). Once we do that, the text covers up Darth Vader completely. Can you make it so on small screens the text **is in** the "normal position" (centered / lower portion of the image) **and for** larger images it's **on the left**.**

Part 1

For the first part, you could just add the style inline.

```
1 <figure class="item active row">
2   
3   <figcaption class="carousel-caption"
4     style="top:0%;left:5%;right:60%;text-align:left">
5     <h1 class>Join the Dark Side</h1>
6     <p>Or the light side. Doesn't matter. If you're into Star Wars
7       then this is the place for you.</p>
8     <p><a class="btn btn-lg btn-primary" href="{% url 'register'
9       %}" role="button">Sign up today</a></p>
10  </figcaption>
11 </figure>
```

Here, we simply changed the style of the figcaption to push the caption to the top and left top:0%;left:5% and have the text wrap at 60% of the right edge, right:60%. We then aligned the text left: text-align:left.

Doing this overrides the CSS for the class `carousel-caption` that is defined both in `bootstrap.css` and `carousel.css`. As a best practice, though, we should keep our styling in CSS files and out of our HTML/templates - but we don't want this style to apply to all `.carousel-caption` items. So let's give this particular caption an id (say `id="darth_caption"`) and then update our `me.css` (where we put all of our custom CSS rules) files as follows:

```
1 #darth_caption {  
2   top: 0%;  
3   left: 5%;  
4   right: 60%;  
5   text-align: left;  
6 }
```

This has the same effect but keeps the styling from cluttering up our HTML.

Part 2

Bootstrap's [Responsive Utilities](#) give you the power to create different styling depending upon the size of the screen that is being used. Check out the chart on the Bootstrap page. By using those special classes, you can show or hide items based on the screen size. For example, if you used `class="hidden-lg"`, then the HTML element associated to that class would be hidden on large screens.

In our example, we essentially use two `figcaptions`. One for large screens, and the other for all other screens:

```
1 <figcaption class="carousel-caption visible-lg-block"  
   id="darth_caption">  
2   <h1>Join the Dark Side</h1>  
3   <p>Or the light side. Doesn't matter. If your into Star Wars then  
     this is the place for you.</p>  
4   <p><a class="btn btn-lg btn-primary" href="{% url 'register' %}"  
       role="button">Sign up today</a></p>  
5 </figcaption>  
6 <figcaption class="carousel-caption hidden-lg">  
7   <h1>Join the Dark Side</h1>  
8   <p>Or the light side. Doesn't matter. If your into Star Wars then  
     this is the place for you.</p>  
9   <p><a class="btn btn-lg btn-primary" href="{% url 'register' %}"  
       role="button">Sign up today</a></p>  
10 </figcaption>
```

That should do it! If you make your browser screen smaller and smaller, you will see that eventually the caption will jump back to the middle lower third. If you make your browser larger, then the caption will jump to the left of Darth Vader in the image. Congratulations, you now understand the basics of how responsive websites are built.

Exercise 2

Question: In this chapter, we updated the Home Page but we haven't done anything about the Contact Page, the Login Page, or the Register Page. Bootstrapify them. Try to make them look awesome. The Bootstrap examples [page](#) is a good place to go to get some simple ideas to implement. Remember: try to make the pages semantic, reuse the Django templates that you already wrote where possible, and most of all have fun.

There is no right or wrong answer for this section. The point is just to explore Bootstrap and see what you come up with. Below you'll see some examples. Let's start with the login page.

Short and sweet.

Sign-in page

MOS EISLEY'S CANTINA **HOME** **ABOUT** **CONTACT** **LOGIN** **REGISTER**

SIGN IN

EMAIL:

PASSWORD:

SIGN IN

© 2014 [The Jedi Council](#) · Powered by about 37 AT-AT Walkers, Python 3 and Django 1.6 · [Back to top](#)

Figure 21.1: Sign-in page

```

1 {% extends "__base.html" %}
2 {% load staticfiles %}
3 {% block extra_css %}
4     <link href="{% static 'css/signin.css' %}" rel="stylesheet">
5 {% endblock %}
6 {% block content %}
7     <div class="container">
8         <form accept-charset="UTF-8" action="{% url 'sign_in' %}"
9             class="form-signin" role="form" method="post">
10             {% csrf_token %}
11             <h1 class="form-signin-heading">Sign in</h1>
12             {% if form.is_bound and not form.is_valid %}
13             <div class="alert-message block-message error">
14                 <div class="errors">
15                     {% for field in form.visible_fields %}
16                         {% for error in field.errors %}
17                             <p>{{ field.label }}: {{ error }}</p>
18                         {% endfor %}
19                     {% endfor %}
20                     {% for error in form.non_field_errors %}
21                         <p>{{ error }}</p>
22                     {% endfor %}
23                 </div>
24             </div>
25             {% endif %}
26             {% for field in form %}{% include "payments/_field.html" %}{%
27                 endfor %}
28             <input class="btn btn-lg btn-primary btn-lg" name="commit"
29                 type="submit" value="Sign in">
30         </form>
31     </div>
32 {% endblock %}

```

This is not much change to our previous sign-in template. The main difference is the new stylesheet. Notice on lines 3-5 we have the following code.

```

1 {% block extra_css %}
2     <link href="{% static 'css/signin.css' %}" rel="stylesheet">
3 {% endblock %}

```

Here we created another block in the `_base.html` template so that we can easily add new

stylesheets from inherited pages:

```
1 {% block extra_css %}  
2 {% endblock %}
```

The modified `__base.html` now looks like:

```
1 {% load static %}  
2  
3 <!DOCTYPE html>  
4 <html lang="en">  
5   <head>  
6     <meta charset="utf-8">  
7     <meta http-equiv="X-UA-Compatible" content="IE=edge">  
8     <meta name="viewport" content="width=device-width,  
9       initial-scale=1">  
10  
11     <title>Mos Eisley's Cantina</title>  
12  
13     <!-- Bootstrap -->  
14     <link href= "{% static "css/bootstrap.min.css" %}"  
15       rel="stylesheet">  
16     <!-- custom styles -->  
17     <link href= "{% static "css/mec.css" %}" rel="stylesheet">  
18     {% block extra_css %}  
19     {% endblock %}  
20  
21     <!-- HTML5 Shim and Respond.js IE8 support of HTML5 elements  
22       and media queries -->  
23     <!-- WARNING: Respond.js doesn't work if you view the page via  
24       file:// -->  
25     <!--[if lt IE 9]>  
26       <script  
27         src="https://oss.maxcdn.com/html5shiv/3.7.2/html5shiv.min.js"></script>  
28       <script  
29         src="https://oss.maxcdn.com/respond/1.4.2/respond.min.js"></script>  
30     <![endif]-->  
31   </head>  
32   <body>  
33     ... snip ...
```

As you can see, just after the stylesheets on line 23-24 we added the `{% block extra_css %}` block. This allows us to add additional CSS or tags in the header. This is a helpful technique to make your reusable templates more flexible. It's always a balance between making your templates more flexible and making things easy to maintain; don't get carried away with adding blocks everywhere. But a few blocks in the right places in your `__base.html` can be very useful.

The final piece of the puzzle of the sign-in page is the `signin.css` that is responsible for the styling. It looks like this:

```
1 body {
2     padding-bottom: 40px;
3     background-color: #eee;
4 }
5
6 .form-signin {
7     max-width: 330px;
8     padding: 15px;
9     margin: 0 auto;
10    font-family: 'starjedi', sans-serif;
11 }
12
13 .form-signin .form-signin-heading,
14 .form-signin .checkbox {
15     margin-bottom: 10px;
16 }
17
18 .form-signin .checkbox {
19     font-weight: normal;
20 }
21
22 .form-signin .form-control {
23     position: relative;
24     height: auto;
25     -webkit-box-sizing: border-box;
26     -moz-box-sizing: border-box;
27     box-sizing: border-box;
28     padding: 10px;
29     font-size: 16px;
30 }
31
```

```

32 .form-signin .form-control:focus {
33     z-index: 2;
34 }
35
36 .form-signin input[type="email"] {
37     margin-bottom: -1px;
38     border-bottom-right-radius: 0;
39     border-bottom-left-radius: 0;
40 }
41
42 .form-signin input[type="password"] {
43     margin-bottom: 10px;
44     border-top-left-radius: 0;
45     border-top-right-radius: 0;
46 }

```

NOTE: This particular stylesheet is relatively short as far as stylesheets go, so we could have included it into the `me.css`. This would reduce the number of extra pages that need to be downloaded and thus improve the response time of the website slightly. However, we've left it separate as a way to demonstrate a good use-case for the Django templates block directive.

We can apply similar formatting to the contact page so that we have a nice consistent theme to our site. Grab the file from the `chpo8` folder in the repo.

Exercise 3

Question: Previously in the chapter we introduced the `marketing__circle_item` template tag. The one issue we had with it was that it required a whole lot of data to be passed into it. Let's see if we can fix that. Inclusion tags don't have to have data passed in. Instead, they can inherit context from the parent template. This is done by passing in `takes_context=True` to the inclusion tag decorator like so:

```

1 @register.inclusion_tag('main/templatetags/circle_item.html',
    takes_context=True)

```

If we did this for our `marketing__circle_item` tag, we wouldn't have to pass in all that data; we could just read it from the context.

Go ahead and make that change, then you will need to update the `main.views.index` function to add the appropriate data to the context when you call `render_to_response`.

Once that is all done, you can stop hard-coding all the data in the HTML template and instead pass it to the template from the view function.

For bonus points, create a marketing_info model. Read all the necessary data from the model in the index view function and pass it into the template.

After ensuring that we have indeed set our marketing_circle_item to takes_context=True-

```
1 from django import template
2 register = template.Library()
3
4
5 @register.inclusion_tag(
6     'main/templatetags/circle_item.html',
7     takes_context=True
8 )
9 def marketing__circle_item(context):
10     return context
```

-the next thing to do is update our associated view function to pass in the context that we need for our marketing items. Updating main.views to do that will cause it to look something like:

```
1 from django.shortcuts import render_to_response
2 from payments.models import User
3 #from main.templatetags.main_marketing import marketing__circle_item
4
5
6 class market_item(object):
7
8     def __init__(self, img_name, heading, caption,
9                 button_link="register",
10                 button_title="View details"):
11         self.img_name = img_name
12         self.heading = heading
13         self.caption = caption
14         self.button_link = button_link
15         self.button_title = button_title
16
17 market_items = [
18     market_item(
19         img_name="yoda.jpg",
```

```

19     heading="Hone your Jedi Skills",
20     caption="All members have access to our unique"
21     " training and achievements latters. Progress through the "
22     "levels and show everyone who the top Jedi Master is!",
23     button_title="Sign Up Now"
24 ),
25 market_item(
26     img_name="clone_army.jpg",
27     heading="Build your Clan",
28     caption="Engage in meaningful conversation, or "
29     "bloodthirsty battle! If it's related to "
30     "Star Wars, in any way, you better believe we do it.",
31     button_title="Sign Up Now"
32 ),
33 market_item(
34     img_name="leia.jpg",
35     heading="Find Love",
36     caption="Everybody knows Star Wars fans are the "
37     "best mates for Star Wars fans. Find your "
38     "Princess Leia or Han Solo and explore the "
39     "stars together.",
40     button_title="Sign Up Now"
41 ),
42 ]
43
44
45 def index(request):
46     uid = request.session.get('user')
47     # for now just hard-code all the marketing info stuff
48     # to see how this works
49     if uid is None:
50         return render_to_response(
51             'main/index.html',
52             {'marketing_items': market_items}
53         )
54     else:
55         return render_to_response(
56             'main/user.html',
57             {'marketing_items': market_items,
58             'user': User.get_by_id(uid)}

```

59

)

1. We created a dummy class called `market_item`. The class is used to make the variables easier to access in the template, and in fact later when we make the model class it is going to end up looking pretty similar to the dummy class.
2. Next we added dummy data. Normally you would read this from the database, but let's start quick and dirty and stick all the data in a list called `market_items`. Notice how we put the list at the module level namespace; this will make it easier to access from the unit tests (which are going to break as soon as we implement this). 1/ The final thing we did was pass the newly created list of marketing items to the template as the context:

```

1  if uid is None:
2      return render_to_response(
3          'main/index.html',
4          {'marketing_items': market_items}
5      )
6  else:
7      return render_to_response(
8          'main/user.html',
9          {'marketing_items': market_items,
10         'user': User.get_by_id(uid)}
11      )

```

Simply passing in the dictionary with key `marketing_items` set to the `market_items` list to the `render_to_response()` function will get the context set up so it's accessible to our templates. Then our inclusion tag, which now has access to the context, can pick it up and pass it to the template `main\templatetags\circle_item.html`. *First a look at the `marketing__circle_item` template.* It now does more or less nothing:

```

1  from django import template
2  register = template.Library()
3
4
5  @register.inclusion_tag(
6      'main/templatetags/circle_item.html',
7      takes_context=True
8  )
9  def marketing__circle_item(context):
10     return context

```

It does have to take the context as the first argument, and whatever it returns will be the context that `circle_item.html` template has access to. We pass the entire context. Finally our template can now loop through the list of `marketing_items` and display the nicely formatted marketing blurb:

```
1 {% load staticfiles %}
2
3 {% for m in marketing_items %}
4     <div class="col-lg-4">
5         
7         <h2>{{ m.heading }}</h2>
8         <p>{{ m.caption }}</p>
9         <p><a class="btn btn-default" href="{% url m.button_link %}"
10             role="button">{{ m.button_title }} &raquo;</a></p>
11     </div>
12 {% endfor %}
```

What we are doing here is looping through the `marketing_items` list (that we passed in from the `main.views.index` function) and created a new circle marketing HTML for each. This has the added advantage that it will allow us to add a variable number of marketing messages to our page!

Finally, make sure to update the `main/index.html` file:

```
1 <section class="container marketing">
2     <!-- Three columns of text below the carousel -->
3     <div class="row center-text">
4         {% marketing__circle_item %}
5     </div>
6 </section>
```

With all that done, fire up the browser and have a look at your site; it should all look as it did before.

Don't forget to check your unit tests. You should see a big ugly failure in `tests.main.testMainPageView.MainPageTests`. This is because your index page now requires the `marketing_info` context variable, and we are not passing it into our test. Remember earlier when we said we were putting the `marketing_items` list at the module level to aid with our testing? Well, let's fix `tests.main.testMainPageView.MainPageTests`:

1. First import the `marketing_items` into the test, so we can reuse the same data:

```
1 from main.views import index, market_items
```

2. Next update the test `test_returns_exact_html` to use `market_items`:

```
1 def test_returns_exact_html(self):
2     resp = index(self.request)
3     self.assertEqual(
4         resp.content,
5         render_to_response(
6             "main/index.html",
7             {"marketing_items": market_items}
8         ).content
9     )
```

Now rerun and all your tests should pass! Great work.

Bonus

Going for the bonus?

Question: ** For bonus points, create a `marketing_info` model. Read all the necessary data from the model in the index view function and pass it into the template.**

Essentially, you'll data drive the template from the database (as opposed to hardcoding a bunch of stuff in the view function). You're actually almost there already. All you need to do is create a model, then have your view read from the model as opposed to the hardcoded values. First let's add a new class to `main/models.py`:

```
1 from django.db import models
2
3
4 # Create your models here.
5 class MarketingItem(models.Model):
6     img = models.CharField(max_length=255)
7     heading = models.CharField(max_length=300)
8     caption = models.TextField()
9     button_link = models.URLField(null=True, default="register")
10    button_title = models.CharField(max_length=20, default="View
        details")
```

It's a simple model with a couple of default values. Next we can change our `main.views` to read from the model instead of from the hardcoded values like so:

```

1 from main.models import MarketingItem
2
3 def index(request):
4     uid = request.session.get('user')
5     market_items = MarketingItem.objects.all()
6     if uid is None:
7         return render_to_response(
8             'main/index.html',
9             {'marketing_items': market_items}
10        )
11    else:
12        return render_to_response(
13            'main/user.html',
14            {'marketing_items': market_items,
15            'user': User.get_by_id(uid)}
16        )

```

Here, we call `market_items = MarketingItem.objects.all()` and then whatever we have in our database will appear on the screen. Groovy!

Fire up the application and check to see if this works. You'll probably get some errors because you didn't run `syncdb` to create the new table. (Also note that we are going to stick with `syncdb` until we get to the migrations chapter, which will show us a better way of updating our models). But even if you did, you won't see any of the marketing items on your index page since there's nothing in your database. We need to load some data.

1. Create a directory "main/fixtures"
2. In that directory create a file called *initial_data.json* (Note: you could also use YAML or XML if you prefer)
3. Now chuck the data you want to load in the *initial_data.json*:

```

1 [
2   {
3     "model": "main.MarketingItem",
4     "pk": 1,
5     "fields":
6     {
7       "img": "yoda.jpg",
8       "heading": "Hone your Jedi Skills",

```

```

9         "caption": "All members have access to our unique training
           and achievements latters. Progress through the levels
           and show everyone who the top Jedi Master is!",
10        "button_title": "Sign Up Now"
11    }
12 },
13 {
14     "model": "main.MarketingItem",
15     "pk": 2,
16     "fields":
17     {
18         "img": "clone_army.jpg",
19         "heading": "Build your Clan",
20         "caption": "Engage in meaningful conversation, or
           bloodthirsty battle! If it's related to Star Wars, in
           any way, you better believe we do it.",
21         "button_title": "Sign Up Now"
22     }
23 },
24 {
25     "model": "main.MarketingItem",
26     "pk": 3,
27     "fields":
28     {
29         "img": "leia.jpg",
30         "heading": "Find Love",
31         "caption": "Everybody knows Star Wars fans are the best
           mates for Star Wars fans. Find your Princess Leia or
           Han Solo and explore the stars together.",
32         "button_title": "Sign Up Now"
33     }
34 }
35 ]

```

Once this is all in place, every time you run syncdb, the data from *initial_data.json* will automatically be put into the database. Re-run syncdb, fire up the Django development server, and you should now see your marketing info. Awesome!

Before you pack everything up, make sure to run the tests. They should all pass.

Another Migrations Note:

Please note that while syncdb will load the data in your fixtures, if you have created any migrations for the application this feature won't work. Again we will cover migrations in an upcoming chapter. So try not to jump ahead or this stuff might not work.

Building the Members Page

Exercise 1

Questions: **Our User Story** US3 Main Page** says that the members page is a place for announcements and to list current happenings. We have implemented user announcements in the form of status reports, but we should also have a section for system announcements/current events. Using the architecture described in this chapter, create an Announcements info_box to display system-wide announcements?**

The simplest way to implement this requirement is with a simple template. Let's call it `main/_announcements.html`:

```
1 <!-- system announcements -->
2 {% load staticfiles %}
3 <section class="info-box" id="announcements">
4   <h1>orders from the Council</h1>
5   <div class="full-image">
6     
7   </div>
8   <div >
9     <h3>April 1: Join us for our annual Smash Jar Jar bash</h2>
10    <p>Bring the whole family to MEC for a fun-filled day of
        smashing Jar Jar Binks!</p>
11  </div>
12 </section>
```

This uses the same `.info_box` class as all our other boxes on the `user.html` page. It includes a heading, image and some details about the announcement. We need a few CSS rules to control the size and position of the image.

Update `mec.css`:

```
1 .full-image {
2   overflow: hidden;
3 }
4
5 .full-image img {
6   position: relative;
7   display: block;
8   margin: auto;
9   min-width: 100%;
```

```

10 min-height: 100px;
11 }

```

This causes the image to center and autoscale to be the size of the info-box, minus the border. Also if the image gets too large, rather than trying to shrink the image to a tiny size (which will cause the image to look pretty bad) it will just be cropped.

Finally we can hook the image into our `user.html` page like so:

```

1 {% extends "__base.html" %}
2 {% load staticfiles %}
3 {% block content %}
4     <div class="row member-page">
5         <div class="col-sm-8">
6             <div class="row">
7                 {% include "main/_announcements.html" %}
8                 {% include "main/_lateststatus.html" %}
9             </div>
10        </div>
11        <div class="col-sm-4">
12            <div class="row">
13                {% include "main/_jedibadge.html" %}
14                {% include "main/_statusupdate.html" %}
15            </div>
16        </div>
17    </div>
18 {% endblock %}

```

It's just another include. We also moved the info-boxes around a bit so the announcements are the first thing the user will see. With this, you should have a page that looks like so:

[User.html with announcements](images/announcements.png)

The advantage to doing something simple like this is:

1. It doesn't take much time.
2. You are free to use whatever HTML you like for each announcement.

The disadvantages:

1. It's static; you need to update the template for each new announcement.

2. You are limited to one announcement, unless of course you update the template for more announcements.

Let's address the disadvantages by data driving the announcements info box from the database in the same way we did for `main_lateststatus.html`.

Step 1: Create a model.

Let's call it `main.models.Announcement`:

```
1 class Announcement(models.Model):
2     when = models.DateTimeField(auto_now=True)
3     img = models.CharField(max_length=255, null=True)
4     vid = models.URLField(null=True)
5     info = models.TextField()
```

We're allowing either an image or a video as the main part of the content, and then `info` will allow us to store arbitrary HTML in the database, so we can put whatever we want in there. We also time stamp the Announcements as we don't want to keep them on the site forever because it doesn't look good to have announcements that are several years old on the site.

Sync the database.

In order to support embedding videos, let's turn to a pretty solid third party app. It's called [django-embed-video](#). It's not very customizable, but it's pretty easy to use and does all the heavy lifting for us.

As always, install it with pip:

```
1 $ pip install django-embed-video
```

Don't forget to add it to the `requirements.txt` file as well as `embed_video` to the `INSTALLED_APPS` tuple in `django_ecommerce/settings.py`. Once done, update that `main/_announcements.html` template.

```
1 <!-- system announcements -->
2 {% load staticfiles %}
3 {% load embed_video_tags %}
4 <section class="info-box" id="announcements">
5     <h1>orders from the Council</h1>
6     {% for a in announce %}
7         <div class="full-image">
8             {% if a.vid %}
```

```

9         {% video a.vid "medium" %}
10     {% else %}
11         
12     {% endif %}
13 </div>
14 <div > {{ a.info | safe }}</div>
15 <br>
16 {% endfor %}
17 </section>

```

- **Line 3:** loads a new set of tags from django-embed-video
- **Line 6:** loops through all the announcements
- **Line 8:** gives videos priority; if we have a video entry in the database, show that...
- **Line 9:** load the video in an iframe using django-embed-video
- **Line 11:** if it's an image, show that instead
- **Line 14:** inserts the HTML from the database; the `safe` filter tells Django to not escape the data so it will be rendered as HTML instead of plain text

In `main/views.index` add another context variable. The relevant part is below:

```

1 def index(request):
2
3     ...snip...
4
5     else:
6         #membership page
7         status = StatusReport.objects.all().order_by('-when')[:20]
8         announce_date = date.today() - timedelta(days=30)
9         announce = (Announcement.objects.filter(
10             when__gt=announce_date).order_by('-when')
11         )
12
13         return render_to_response(
14             'main/user.html',
15             {
16                 'user': User.get_by_id(uid),
17                 'reports': status,
18                 'announce': announce
19             },

```

```

20         context_instance=RequestContext(request),
21     )

```

Update the imports as well:

```

1 from django.shortcuts import render_to_response, RequestContext
2 from payments.models import User
3 from main.models import MarketingItem, StatusReport, Announcement
4 from datetime import date, timedelta

```

Basically we are grabbing all the announcements in the last thirty days and ordering them, so the most recent will appear at the top of the page.

Run the tests. Make sure to manually test as well.

Exercise 2

Question: You may have noticed that in the Jedi Badge box there is a list achievements link. What if the user could get achievements for posting status reports, attending events, and any other arbitrary action that we create in the future? This may be a nice way to increase participation, because everybody likes badges, right? Go ahead and implement this achievements feature. You'll need a model to represent the badges and a link between each user and the badges they own (maybe a `user_badges` table). Then you'll want your template to loop through and display all the badges that the given user has.

There are several ways to do this. We'll look at the most straightforward.

First create the model `main.models.Badge`:

```

1 class Badge(models.Model):
2     img = models.CharField(max_length=255)
3     name = models.CharField(max_length=100)
4     desc = models.TextField()
5
6     class Meta:
7         ordering = ('name',)

```

Each user will have a reference to the badges and many users can get the same badge, so this creates a many-to-many relationship. Thus we will update `payments.models.User` adding the new relationship field:

```

1 badges = models.ManyToManyField(Badge)

```

The default for a `ManyToManyField` is to create a lookup table. After adding this code, drop the database, re-create it, and then run `syncdb` and in your database you will have a table called `payments_user_badges`. The `badges` field will manage all the “relationship” stuff for you. Of course we have to add `from main.models import Badge` for this to work. That causes a problem though because we already have `from payment.models import User` in `main.models` (because we are using it in the `StatusReport` model). This creates a circular reference and will cause the import to break.

NOTE Circular references don’t always cause imports to break, and there are ways to make them work, but it is generally considered bad practice to have circular references. You can find a good discussion on circular references [here](#).

We can remove this circular reference by changing our `StatusReport` model class so it doesn’t have to import `payments.Users`. We do that like so:

```
1 class StatusReport(models.Model):
2     user = models.ForeignKey('payments.User')
3     ...snip...
```

In the case of the `ForeignKey` field, Django allows us to reference a model by using its name as a string. This means that Django will wait until the `payments.User` model is created and then link it up with `StatusReport`. It also means we can remove our `from payments.main import User` statement, and then we don’t have a circular reference.

Next up is to return the list of badges as part of the request for the `user.html` page. Updating our `main.views.index()` function we now have:

```
1 def index(request):
2
3     ...snip...
4
5     else:
6         #membership page
7         status = StatusReport.objects.all().order_by('-when')[:20]
8
9         announce_date = date.today() - timedelta(days=30)
10        announce = (Announcement.objects.filter(
11            when__gt=announce_date).order_by('-when')
12        )
13
14        usr = User.get_by_id(uid)
```

```

15     badges = usr.badges.all()
16
17     return render_to_response(
18         'main/user.html',
19         {
20             'user': usr,
21             'badges': badges,
22             'reports': status,
23             'announce': announce
24         },
25         context_instance=RequestContext(request),
26     )

```

On **Line 13** we get all the badges linked to the current user by calling `user.badges.all()`. `badges` is the `ManyToManyField` we just created. The `all()` function will return a list of all the related badges. We just set that list to the context variable `badges` and pass it into the template.

Speaking of which, update the `user.html` template:

```

1  {% extends "__base.html" %}
2  {% load staticfiles %}
3  {% block content %}
4  <div id="achievements" class="row member-page hide">
5      {% include "main/_badges.html" %}
6  </div>
7  <div class="row member-page">
8      <div class="col-sm-8">
9          <div class="row">
10             {% include "main/_announcements.html" %}
11             {% include "main/_lateststatus.html" %}
12          </div>
13      </div>
14      <div class="col-sm-4">
15          <div class="row">
16             {% include "main/_jedibadge.html" %}
17             {% include "main/_statusupdate.html" %}
18          </div>
19      </div>
20  </div>
21  {% endblock %}

```

Lines 4-6 are the important ones here. Basically we are creating another info box that will stretch across the top of the screen to show all the badges. But we are adding the Bootstrap CSS class `hide` so the achievements row won't be shown.

The main/_badges.html template:

```
1 {% load staticfiles %}
2 <section class=" row info-box text-center" id="badges">
3   <h1 id="achieve">Achievements</h1>
4   {% for bdg in badges %}
5     <div class="col-lg-4">
6       <h2>{{ bdg.name }}</h2>
7       
10      <p>{{ bdg.desc }}</p>
11    </div>
12  {% endfor %}
13 </section>
```

Here, we loop through the badges and show them with a heading and description. Using `col-lg-4` means there will be three columns per row; if there are more than three badges, it will just wrap and add another row.

The final thing to do is to make the “Show Achievements” link work, since by default the Achievements info-box is hidden. Clicking the Show Achievements link should show them. And once they are visible, clicking the link again should hide them. The easiest way to do this is to use some JavaScript. We haven't really talked much about JavaScript in this course yet, but we will in an upcoming chapter. For now, just have a look at the code, which goes in `application.js`:

```
1 //show status
2 $("#show-achieve").click(function() {
3   a = $("#achievements");
4   l = $("#show-achieve");
5   if (a.hasClass("hide")) {
6     a.hide().removeClass('hide').slideDown('slow');
7     l.html("Hide Achievements");
8   } else {
9     a.addClass("hide");
10    l.html("Show Achievements");
11  }
```



```

12     return false;
13 });

```

This handles the click event that is called when you click on the “Show Achievements” link. Let’s walk through the code.

- **Line 2:** sets up the click even handler.
- **Line 3:** grabs a reference to the achievements info box.
- **Line 4:** grabs a reference to the “Show Achievements” link itself.
- **Line 5:** if a has the hide class (i.e. Bootstrap is currently hiding the info-box), then...
- **Line 6:** remove the class “hide”, which will make the info box visible, and then use a jQuery animation ‘slideDown’ to make it scroll in slowly.
- **Line 7:** changes the text of the “Show Achievements” link to “Hide Achievements”.
- **Line 9-11:** do the opposite of lines 5-7; hide the info box and change the link text to “Show Achievements”.
- **Line 12:** return false, which prevents the screen from redrawing.

Finally, update the `test_index_handles_logged_in_user()` test in `tests.main.testMainPageView.MainPageView` so that the foreign key relationship between users and badges is established:

```

1 def test_index_handles_logged_in_user(self):
2     #create a session that appears to have a logged in user
3     self.request.session = {"user": "1"}
4
5     #setup dummy user
6     #we need to save user so user -> badges relationship is created
7     u = User(email="test@user.com")
8     u.save()
9
10    with mock.patch('main.views.User') as user_mock:
11
12        #tell the mock what to do when called
13        config = {'get_by_id.return_value': u}
14        user_mock.configure_mock(**config)
15
16        #run the test
17        resp = index(self.request)
18
19        #ensure we return the state of the session back to normal
20        self.request.session = {}

```

```
21     u.delete()
22
23     #we are now sending a lot of state for logged in users,
        rather than
24     #recreating that all here, let's just check for some text
25     #that should only be present when we are logged in.
26     self.assertContains(resp, "Report back to base")
```

Be sure to add the import as well:

```
1 from payments.models import User
```

Finally, we need to update the database:

1. Enter the Postgres Shell:

```
1 DROP DATABASE django_db;
2 CREATE DATABASE django_db;
```

2. Exit the shell and run syncdb:

```
1 $ ./manage.py syncdb
```

That's it. Make sure to test, then commit your code.

REST

Exercise 1

Question: **Flesh out the unit tests. In the `JsonViewTests`, check the case where there is no data to return at all, and test a POST request with and without valid data.**

Expanding the tests, we can write a `test_get_member()`:

```
1 def test_get_member(self):
2     stat = StatusReport(user=self.test_user, status="testing")
3     stat.save()
4
5     status = StatusReport.objects.get(pk=stat.id)
6     expected_json = StatusReportSerializer(status).data
7
8     response = StatusMember.as_view()(self.get_request(),
9                                       pk=stat.id)
10
11     self.assertEqual(expected_json, response.data)
12
13     stat.delete()
```

This is very similar to the `test_get_collection()`. The main difference here is that we are saving a status report for our test user and then passing in the pk to our view. This is the same as calling the url `/api/v1/status_reports/1`.

Likewise, we can test other methods such as DELETE:

```
1 def test_delete_member(self):
2     stat = StatusReport(user=self.test_user, status="testing")
3     stat.save()
4
5     response = StatusMember.as_view()(
6         self.get_request(method='DELETE'), pk=stat.pk)
7
8     self.assertEqual(response.status_code,
9                       status.HTTP_204_NO_CONTENT)
10
11     stat.delete()
```

Now, in order to get this to pass, we need to update the `setUpClass()` method and add a `tearDownClass()` method.

```
1 @classmethod
2 def setUpClass(cls):
3     cls.factory = APIRequestFactory()
4     cls.test_user = User(id=2222, email="test@user.com")
5     cls.test_user.save()
6
7 @classmethod
8 def tearDownClass(cls):
9     cls.test_user.delete()
```

Why do you think we need to add these?

Run the tests. All 38 should pass.

Exercise 2

Questions: **Extend the REST API to cover the `user.models.Badge`.**

To create a new endpoint you need to update three files:

1. `main/serializers.py` - create the JSON serializer
2. `main/json_views.py` - create the DRF views
3. `main/urls.py` - add the REST URIs (endpoints)

First, create the serializer:

```
1 class BadgeSerializer(serializers.ModelSerializer):
2
3     class Meta:
4         model = Badge
5         fields = ('id', 'img', 'name', 'desc')
```

Nothing to it. Just use the handy `serializer.ModelSerializer`. Make sure to update the imports as well:

```
1 from payments.models import User, Badge
```

Now let's create the Collection and Member views:

```

1 class BadgeCollection(
2     mixins.ListModelMixin, mixins.CreateModelMixin,
3     generics.GenericAPIView
4 ):
5     queryset = Badge.objects.all()
6     serializer_class = BadgeSerializer
7     permission_classes = (permissions.IsAuthenticated,)
8
9     def get(self, request):
10         return self.list(request)
11
12     def post(self, request):
13         return self.create(request)
14
15
16 class BadgeMember(
17     mixins.RetrieveModelMixin, mixins.UpdateModelMixin,
18     mixins.DestroyModelMixin, generics.GenericAPIView
19 ):
20
21     queryset = Badge.objects.all()
22     serializer_class = BadgeSerializer
23     permission_classes = (permissions.IsAuthenticated,)
24
25     def get(self, request, *args, **kwargs):
26         return self.retrieve(request, *args, **kwargs)
27
28     def put(self, request, *args, **kwargs):
29         return self.update(request, *args, **kwargs)
30
31     def delete(self, request, *args, **kwargs):
32         return self.destroy(request, *args, **kwargs)

```

It's a bit of copy and paste from the StatusMember and StatusCollection, but it's pretty clear exactly what is going on here.

Again, update the imports:

```

1 from main.serializers import StatusReportSerializer, BadgeSerializer
2 from main.models import StatusReport, Badge

```

Finally we need to wire up our URI's in `main/urls.py`:

```
1 urlpatterns = patterns(
2     'main.json_views',
3     url(r'^$', 'api_root'),
4     url(r'^status_reports/$', json_views.StatusCollection.as_view(),
5         name='status_reports_collection'),
6     url(r'^status_reports/(?P<pk>[0-9]+)/$',
7         json_views.StatusMember.as_view()),
8     url(r'^badges/$', json_views.BadgeCollection.as_view(),
9         name='badges_collection'),
10    url(r'^badges/(?P<pk>[0-9]+)/$',
11        json_views.BadgeMember.as_view()),
12)
```

Don't forget to add the unit tests as well. This is all you since you should be an expert now after doing all the testing from exercise 1. If you'll notice, though, once you do all your tests for Badges, they are probably pretty similar to your tests for 'StatusReport'. Can you factor out `arest_api_test_case`? Have a look back at the testing mixins in Chapter 2 and see if you can do something similar here.

Exercise 3

Question: Did you know that the browsable API uses Bootstrap for the look and feel? Since we just learned Bootstrap, update the browsable API Template to fit with our overall site template.

This is actually pretty easy to do once you know how. In case your "Googling" didn't find it, the DRF documentation tells you how [on this page](#).

Basically, all you have to do is create a template `rest_framework/api.html` that extends the DRF template `rest_framework/base.html`. The simplest thing we can do is to change the CSS to use the CSS we are using for the rest of our site. To do that, make a `rest_framework/api.html` template look like this:

```
1 {% extends "rest_framework/base.html" %}
2 {% load staticfiles %}
3 {% block bootstrap_theme %}
4     <link href="{% static "css/bootstrap.min.css" %}"
5         rel="stylesheet">
6     <!-- HTML5 Shim and Respond.js IE8 support of HTML5 elements
7         and media queries -->
```

```

6      <!-- WARNING: Respond.js doesn't work if you view the page via
          file:// -->
7      <!--[if lt IE 9]>
8          <script
              src="https://oss.maxcdn.com/libs/html5shiv/3.7.0/html5shiv.js"></script>
9          <script
              src="https://oss.maxcdn.com/libs/respond.js/1.4.2/respond.min.js"></script>
10         <![endif]-->
11         <link href="{% static 'css/mec.css' %}" rel="stylesheet">
12     {% endblock %}

```

Here we added the `bootstrap_theme` block and set it to use the Bootstrap CSS that we are using and then also use our custom `mec.css` stylesheet, mainly to get our cool `star_jedi` font. Super simple.

Go ahead. Test it out. Fire up the server, and then navigate to http://127.0.0.1:8000/api/v1/status_reports/.

Again [the docs](#) have a lot more information about how to customize the look and feel of the browsable API, so have a look if you're interested.

Exercise 4

Question: We don't have permissions on the browsable API. Add them in.

Right now, since our REST API requires authenticated users, the browsable API doesn't work very well unless you can log in. So we can use the built-in DRF login and logout forms so that a user can login and use the browsable API. To do that, all we need to do is update our `main/urls.py` file by adding this line at the end of the `urlpatterns` member:

```

1 url(r'^api-auth/', include(
2     'rest_framework.urls', namespace='rest_framework')),

```

With this, you will now be able to browse to <http://localhost:8000/api/v1/api-auth/login/> where you can log in as the superuser, for example. Then you can browse the API.

Another part of the browsable API that is missing is a “main page” where we can see all of the URIs that the API contains. We can create one by updating our `main/json_views.py` file as such:

```

1 from rest_framework.decorators import api_view
2 from rest_framework.response import Response
3 from rest_framework.reverse import reverse

```

```

4
5
6 @api_view(('GET',))
7 def api_root(request):
8     return Response({
9         'status_reports': reverse(
10             'status_reports_collection', request=request),
11         'badges': reverse('badges_collection', request=request),
12     })

```

This view function returns a list of the two URIs that we currently support. This way the user can start here and click through to view the entire API. All we need to do now is add the URL to our `main/urls.py`:

```

1 url(r'^$', 'api_root'),

```

Now as an API consumer we don't have to guess what the available resources are in our REST API - just browse to <http://localhost:8000/api/v1/> to view them all.

Django Migrations

Exercise 1

Question: At this point if you drop your database, run migrations and then run the tests you will have a failing test because there are no MarketItems in the database. For testing you have two options:

- Load the data in the test (or use a fixture).
- Load the data by using a datamigration.

The preferred option for this case is to create a data migration to load the MarketingItems. Can you explain why?. Create the migration.

First, create a new migrations file. Use the following command to help out:

```
1 $ ./manage.py makemigrations --empty main
```

This will create a new migration file in main\migrations, which should be called 0003_auto_<some ugly date string>.py. I don't particularly care for that name so let's rename it to data_load_marketing_itmes_0003.py. (Later it will become clear why we put the number at the end). Then we can update the contents of the file to create the marketing items that we need. In total the file should now look like:

```
1 # -*- coding: utf-8 -*-
2 from __future__ import unicode_literals
3
4 from django.db import models, migrations
5
6
7 #start with a list of all the data we want to add.
8 init_marketing_data = [
9     {
10         "img": "yoda.jpg",
11         "heading": "Hone your Jedi Skills",
12         "caption": "All members have access to our unique"
13         " training and achievements latters. Progress through the "
14         "levels and show everyone who the top Jedi Master is!",
15         "button_title": "Sign Up Now"
16     },
17     {
```

```

18         "img": "clone_army.jpg",
19         "heading": "Build your Clan",
20         "caption": "Engage in meaningful conversation, or "
21         "bloodthirsty battle! If it's related to "
22         "Star Wars, in any way, you better believe we do it.",
23         "button_title": "Sign Up Now"
24     },
25     {
26         "img": "leia.jpg",
27         "heading": "Find Love",
28         "caption": "Everybody knows Star Wars fans are the "
29         "best mates for Star Wars fans. Find your "
30         "Princess Leia or Han Solo and explore the "
31         "stars together.",
32         "button_title": "Sign Up Now"
33     },
34 ]
35
36 def create_marketing_items(apps, schema_editor):
37
38     MarketingItem = apps.get_model("main", "MarketingItem")
39
40     #store data in database
41     [MarketingItem(**d).save() for d in init_marketing_data]
42
43
44 class Migration(migrations.Migration):
45
46     dependencies = [
47         ('main', '0002_statusreport'),
48     ]
49
50     operations = [
51         migrations.RunPython(create_marketing_items)
52     ]

```

This follows the same pattern as the user creation data migration we did in the *Data Migrations* section of this chapter. The main differences are as follows:

1. The top part of the file just lists all the data that we are going to load. This makes it

easy to see what “system data” we will be loading.

2. In our data creation function `create_marketing_items()` notice the following line:

```
1 MarketingItem = apps.get_model("main", "MarketingItem")
```

This uses the [app registry](#) that is provided by the migration framework so we can get the version of the `MarketingItem` that corresponds to our particular migration. This is important in case the `MarketingItem` model later changes. By using the app registry we can be sure we are getting the correct version of the model.

Once we have that correct version we just create the data using a list comprehension.

But why use a funny name for the file?

Remember that we are trying to load the data so that our `testmain.testMainPageView.test_returns_exact_html` will run correctly. If you recall the test itself looks like:

```
1 ```python
2 def test_returns_exact_html (self):
3     market_items = MarketingItem.objects.all()
4     resp = index(self.request)
5     self.assertEqual(
6         resp.content,
7         render_to_response("main/index.html",
8             {"marketing_items":market_items}
9         ).content)
10 ```
```

So we are comparing the html rendered in `index.html` to the html rendered when we pass in `market_items`. If you recall from earlier chapters, `market_items` is a list that contains all the items that we want to pass into our template to render the template. This was left over from before we switch the view function to read from the database. So this leaves us in a state where we have duplication. We have two lists of system data:

1. `data_load_marketing_items_0003.py` - the data we used to load the database
2. `main.views.market_items` - a left over from when we refactored the code to load marketing items from the database.

So to remove that duplication and only have one place to store / load the initial system data we can change our test as follows:

```

1 from main.migrations.data_load_marketing_items_0003 import
   init_marketing_data
2
3
4 def test_returns_exact_html(self):
5     data = [MarketingItem(**d) for d in init_marketing_data]
6     resp = index(self.request)
7     self.assertEqual(
8         resp.content,
9         render_to_response(
10             "main/index.html",
11             {"marketing_items": data}
12         ).content
13     )

```

NOTE: Don't forget to remove the `market_items` import at the top of this file.

What we have done here is load the `init_marketing_data` list from our migration directly into our test, so we don't need to keep the list in `main.views.market_items` around anymore and we remove the code duplication!

Again, so why the funny name for the migration file?

Because python doesn't allow you to import a file name that starts with a number. In other words-

```

1 from main.migrations.0003_data_load_marketing_items import
   init_marketing_data

```

-will fail. Of course as with all things in python you can get around it by doing something like:

```

1 init_marketing_data =
   __import__('main.migrations.0003_data_load_marketing_items.init_marketing_data')

```

But you really should avoid calling anything with `__` if you can. So we renamed the file instead.

Exercise 2

Question We have a new requirement for **two-factor authentication**. Add a new field to the user model called `second_factor`. Run `./manage.py makemigration payments`. What did it create? Can you explain what is going on in each line of

the migration? Now run `./manage.py migrate` and check the database to see the change that has been made. What do you see in the database? Now assume management comes back and says two-factor is too complex for users; we don't want to add it after all. List two different ways you can remove the newly added field using migrations.

We are not going to show the code in the first part; just try it out and see what happens. For the question at the end - "list two different ways you can remove the newly added field using migrations" - the answers are:

1. You can migrate backwards to the previous migration. So assuming this was migration 0004 that came right after migration 0003 you could simply run:

```
1 $ ./manage.py migrate payments 0003
```

That will return you to before the field was created. Then just drop the field from your model and continue.

2. The second way is to just drop the field from the model and then run `makemigrations` again, which will then drop the field. At that point, since your migrations are going around in a circle you may want to look at [Squashing Migrations](#). Do keep in mind though that squashing migrations is completely optional and not at all required.

Exercise 3

Question: Let's pretend that MEC has been bought by a big corporation - we'll call it BIGCO. BIGCO loves making things complicated. They say that all users must have a bigCoID, and that ID has to follow a certain formula. The ID should look like this: <first_two_digits_in_name><1-digit-Rank_code><sign-up-date>

- 1-digit-Rank_code = 'Y' for youngling, 'P' for padwan, 'J' for Jedi
- sign-up-date = Use whatever date format you like
- Since this is an ID field we need to ensure that it's unique.

Now create the new field and a migration for the field, then manually write a data migration to populate the new field with the data from the pre-existing users.

First add the field to `payments.models.User`:

```
1 bigCoID = models.CharField(max_length=50)
```

Then create the migration with:

```
1 $ ./manage.py makemigrations payments
```

Which will give you a warning about non-nullable fields:

```
1 You are trying to add a non-nullable field 'bigCoID' to user
  without a default;
2 we can't do that (the database needs something to populate existing
  rows).
3 Please select a fix:
4 1) Provide a one-off default now (will be set on all existing rows)
5 2) Quit, and let me add a default in models.py
6 Select an option:
```

Just select 1 and put in 'foo'. We are going to change it in a second.

This will create a migration with a name like 0004_<some ugly string of numbers and dashes>, which will house your migration.

It should include this operation (notice we have changed the default value):

```
1 migrations.AddField(
2     model_name='user',
3     name='bigCoID',
4     field=models.CharField(max_length=50, default='foo'),
5     preserve_default=False,
6 ),
```

There maybe another AlterField statement for the last_notification field, which you can ignore or delete. Now just apply the migration with:

```
1 $ ./manage.py migrate
```

This should create the field for you.

Next we need to create the data migration, which we have done a couple of times now already so you should be getting good at this now. Create a file called 0005_bigcoId_migration.py:

```
1 # -*- coding: utf-8 -*-
2 from __future__ import unicode_literals
3
4 from django.db import migrations
5
6
7 def migrate_bigcoId(apps, schema_editor):
8
```

```

9     User = apps.get_model('payments', 'User')
10
11     for u in User.objects.all():
12         bid = ("%s%s%s%s" % (u.name[:2],
13                               u.rank[:1],
14                               u.created_at.strftime("%Y%m%d%H%M%S%f"),
15                               ))
16         u.bigCoID = bid
17         u.save()
18
19
20 class Migration(migrations.Migration):
21
22     dependencies = [
23         ('payments', '0004_auto_20141001_0546'),
24     ]
25
26     operations = [
27         migrations.RunPython(migrate_bigcoid)
28     ]

```

That should do it. Apply the migration.

What about the unique field?

You may be thinking that bigCoID should be a unique field. And you're right. The problem is if we try to add a unique field, to a table, we need to ensure the data is unique so we have to do that as a three step process:

1. Create a new field without unique constraint (Schema Migration)
2. Update the data in the field to make it unique (Data Migration)
3. Add a unique constraint to the field (Schema Migration)

The only other way to do it would be to create a custom migration using the [RunSQL operation](#). But that means you have to write the SQL by hand.

Coming back to our specific example. We have already completed step 1 and 2. So all we have to do now is edit `payments/models.py` and add `unique=True` to `bigCoID`. Then run `create` and apply the migration as usual. And that will apply the unique constraint in the database and you are good to go.

And creating new Users

One last thing we need to handle: Creating a new user. If we stop here, it will become difficult to create new users without getting a unique key constraint error. So let's update the create function in `payments.models.User` so that it will generate the bigCoID for us. The create function will now look like this:

```
1 @classmethod
2 def create(cls, name, email, password, last_4_digits, stripe_id=''):
3     new_user = cls(name=name, email=email,
4                     last_4_digits=last_4_digits, stripe_id=stripe_id)
5     new_user.set_password(password)
6
7     #set bigCoID
8     new_user.bigCoID = ("%s%s%s" % (new_user.name[:2],
9                                     new_user.rank[:1],
10                                    datetime.now().strftime("%Y%m%d%H%M%S%f"),
11                                    ))
12     new_user.save()
13     return new_user
```

Notice that we are using `datetime.now().strftime("%Y%m%d%H%M%S%f")`. This is a date-time string accurate to the micro second. Otherwise known as a “poor man’s unique id”. Since we are accurate down to the micro second there is an extremely low probability that there will ever be two users created in the same micro second and thus this should basically always produce a unique value. You could also use a database sequence to ensure uniqueness. Take a look at [this django snippet](#), for example.

Also, we need to update a few test in `tests.payments.testUserModel` to use the create method (as we didn’t strictly require it until now):

```
1 @classmethod
2 def setUpClass(cls):
3     cls.test_user = User.create(email="j@j.com", name='test user',
4                                 password="pass",
5                                 last_4_digits="1234")
6
7 def test_create_user_function_stores_in_database(self):
8     self.assertEqual(User.objects.get(email="j@j.com"),
9                       self.test_user)
```

Here we simply update the `setUpClass()` and the `test_user_function_stores_in_database()` function to use `User.create()`. Also to be extra safe you may want to create another test

called something like `test_create_two_users_each_user_has_unique_bigCoID()` to ensure that our create function is indeed generating unique values for `bigCoID`.

AngularJS Primer

Exercise 1

Question: **Our User Poll example uses progress bars, which are showing a percentage of votes. But our vote function just shows the raw number of votes, so the two don't actually match up. Can you update the vote() function to return the current percentage of votes. (HINT: You will also need to keep track of the total number of votes.)**

The answer to this question relies on storing a bit more information in the controller. Since we are updating percentages, we will have to update every item in the list each time a vote is placed. Here's a look at the controller in its entirety, and then we will break it down line by line:

```
1  meApp.controller('UserPollCtrl', function($scope) {
2    $scope.total_votes = 0;
3    $scope.vote_data = {}
4
5    $scope.vote = function(voteModel) {
6      if (!$scope.vote_data.hasOwnProperty(voteModel)) {
7        $scope.vote_data[voteModel] = {"votes": 0, "percent": 0};
8        $scope[voteModel]=$scope.vote_data[voteModel];
9      }
10     $scope.vote_data[voteModel]["votes"] =
11       $scope.vote_data[voteModel]["votes"] + 1;
12     $scope.total_votes = $scope.total_votes + 1;
13     for (var key in $scope.vote_data) {
14       item = $scope.vote_data[key];
15       item["percent"] = item["votes"] / $scope.total_votes * 100;
16     }
17   };
18 });
```

Okay. Let's break it down line by line:

- **Line 1** - No change here; just the definition of our controller.
- **Line 2** - `$scope.total_votes = 0;`. We are keeping track of the total number of votes so we can calculate percentages.

- **Line 3** - `$scope.vote_data = {}`. This creates an empty JavaScript object (which in practice functions much like a Python dictionary). We will use this as the data structure to store information about the votes. We are not defining the data structure yet, but when we do, it will look like this:

```

1 { 'votes_for_yoda':
2   { 'votes': 0,
3     'percent': 0'
4   },
5   'votes_for_qui':
6   { 'votes': 0,
7     'percent': 0'
8   },
9 }

```

Of course the structure will continue for each Jedi we add to the list. But the basic idea is that we have a key with `votes_for_` + the jedi name, and that key returns the number of votes and the percentage of all votes.

- **Line 6 and 7** - Since we have not defined any data in our `votes_data` data structure, we need to dynamically add it each time the `vote()` function is called. `hasOwnProperty` checks to see if we have a key equal to the `viewModel` string passed in, which should be something like `votes_for_yoda`. If we don't, then this is the first time that particular Jedi has been voted for, so we will create a new data structure for the Jedi and initialize everything to 0.
- **Line 8** - This line is completely optional, and we're adding it to save typing in the view. Instead of typing something like `{{ vote_data.votes_for_yoda.percent }}`, we can simply say - `{{ votes_for_yoda.percent }}`.
- **Line 10** - Increase the number of votes for the current Jedi by one.
- **Line 11** - Keep track of the total number of votes.
- **Lines 12-14** - Loop through the data structure of all Jedi's and recalculate their associated percentage of votes.

That gets us through the controller, which will automatically update all associated vote percentages.

Now we do need to make some updates to the view as well. To save space, let's just look at Yoda:

```

1 <span ng-click='vote("votes_for_yoda")' class="glyphicon
  glyphicon-plus"></span>
2 <strong>Yoda</strong>
3 <span class="pull-right">{{ votes_for_yoda.percent | number:0
  }}%</span>
4 <div class="progress">
5   <div class="progress-bar progress-bar-danger" role="progressbar"
     aria-value="{{ votes_for_yoda.percent }}"
6     aria-valuemin="0" aria-valuemax="100" style="width: {{
       votes_for_yoda.percent }}%;">
7   </div>
8 </div>

```

- **Line 1** - The call to the controller - e.g., `ng-click='vote("votes_for_yoda")'` - is unchanged.
- **Line 3** - `{{ votes_for_yoda.percent | number:0 }}`. Here the expression being evaluated asks for the percentage attached to `votes_for_yoda` and uses the Angular number filter to format the number with 0 decimal places. Remember: We can use the shorthand to get the model because of what we did in Line 8 of the controller.
- **Lines 3, 5, and 6** - Again, we just access the percent for the appropriate votes item from the controller.

Likewise, we repeat the same structure for each vote item, and now you have a set of progress bars that always total up to 100 and all dynamically update each time you cast a vote!

Curious to see the vote count for each individual Jedi? Update the HTML like so:

```

1 <strong>Yoda (votes: {{ votes_for_yoda.votes }})</strong>

```

Be sure to do this for all Jedi's.

Djangular: Integrating Django and Angular

Exercise 1

Question: For the first exercise let's explore storing state in our factory. Change the `pollFactory.getPoll` function to take no parameters and have it return the single most recent poll. Then cache that poll's id, so next time `getPoll()` is called you have the id of the poll to retrieve.

Before we even begin to element caching, we need to make sure our API returns a `poll.id`. Let's test it out. Fire up the server, then navigate to <http://localhost:8000/api/v1/polls/?format=json>. You should see something like this:

```
1  [
2  [
3      {
4          "id": 1,
5          "title": "Who is the best Jedi?",
6          "publish_date": "2014-10-21T04:05:24.107Z",
7          "items": [
8              {
9                  "id": 1,
10                 "poll": 1,
11                 "name": "Yoda",
12                 "text": "Yoda",
13                 "votes": 1,
14                 "percentage": 50
15             },
16             {
17                 "id": 2,
18                 "poll": 1,
19                 "name": "Vader",
20                 "text": "Vader",
21                 "votes": 1,
22                 "percentage": 50
23             },
24             {
25                 "id": 3,
26                 "poll": 1,
27                 "name": "Luke",
```

```

28         "text": "Luke",
29         "votes": 0,
30         "percentage": 0
31     }
32 ],
33     "total_votes": 2
34 }
35 ]

```

So, yes - there is a `poll.id`. Where is the code for this? Open `django_polls/serializers.py`. It's the line:

```

1 fields = ('id', 'title', 'publish_date', 'items', 'total_votes')

```

Okay. Now to update the controller. We could change the JSON API to provide a way to return only the latest poll, but let's learn a bit more about Angular instead. Update the `pollFactory`:

```

1 pollsApp.factory('pollFactory', function($http, $filter) {
2
3     var baseUrl = '/api/v1/';
4     var pollUrl = baseUrl + 'polls/';
5     var pollItemsUrl = baseUrl + 'poll_items/';
6     var pollId = 0;
7     var pollFactory = {};
8
9     pollFactory.getPoll = function() {
10         var tempUrl = pollUrl;
11         if (pollId != 0) { tempUrl = pollUrl + pollId; }
12
13         return $http.get(pollUrl).then(function(response)
14         {
15             var latestPoll = $filter('orderBy')(response.data,
16                 '-publish_date')[0];
17             pollId = latestPoll.id;
18             return latestPoll;
19         });
20
21     pollFactory.vote_for_item = function(poll_item) {
22         poll_item.votes += 1;
23         return $http.put(pollItemsUrl + poll_item.id, poll_item);

```

```

24 }
25
26 return pollFactory;
27 });

```

- **Line 5** - `var pollId = 0;` This is our state variable to store the poll `id`.
- **Line 8** - Notice we aren't taking in any parameters (before the function took an `id`).
- **Lines 9-10** - If we don't have a `pollId` cached then just call `/api/v1/polls/`, which returns the complete list of polls. Otherwise pass the `id`, so we would be calling `/api/v1/polls/<pollId>`.
- **Line 11** Notice we are calling `return` on our `$http.get().then()` function. `then()` returns a promise. So, we will be returning a promise, but only after we first call `$http.get`, then we call our `then` function (i.e., lines 12-16), then the caller or controller gets the return value from line 13.
- **Line 13** - This may be something new to you. It's the `$filter` function from Angular, which is the same function that we use in the Angular templates - i.e., `[[ model | filter ]]`. Here we are just calling it from the controller.

Let's digress for a second...

To access the `$filter` function we need to inject it into the top of our factory like so:

```

1 pollsApp.factory('pollFactory', function($http, $filter) {

```

Then a general form of the `filter` function looks like this:

```

1 $filter(filterName)(array, expression, reverse)

```

- `filterName` is the name of the filter to use. In our case we are using the `orderBy` filter.
- `array` is the array to filter. For us, it's `response.data` from our `$http.get` - i.e., our list of Polls.
- `expression` generally takes the name of the attribute on the object you want to order by. You can also prepend `+` or `-` to the front of the attribute to control the order to be ascending or descending. `+` is default.
- `reverse` - instead of prepending `+` or `-`, you can pass `reverse`, which is a boolean value.

So this line orders the list for us, and we take the first element in the list after it is ordered.

Back to the `pollFactory` code:

- **Line 14** - grab the `id` from the poll item we are working with and
- **Line 15** - return the poll item.

Finally, there is a slight change we need to make in our controller. Because our `getPoll` function is now returning a Poll as the promise instead of the response as the promise, we need to change the `setPoll` function. It should now look like this:

```
1 function setPoll(promise){
2   $scope.poll = promise;
3 }
```

And that's it. You will now get the whole list of polls for the first call, and after that only get the poll you are interested in.

Exercise 3

Question: Currently our application is a bit of a mismatch, we are using jQuery on some parts - i.e., showing achievements - and now Angular on the user polls. For practice, convert the showing achievement functionality, from *application.js*, into Angular code.

Let's handle this in four parts...

Part 1

The first thing we need to do is initialize our ng-app somewhere more general. I'm a big fan of DRY, so let's just declare the ng-app on the `<html>` tag in `__base.html`.

```
1 <html lang="en" ng-app="mecApp">
```

We'll call it `mecApp` since that's more fitting for our application. This also means we need to remove it from `__polls.html`. Also, let's move the declaration of the `angular.module` to `application.js` (so it will be the first thing created). The top of `application.js` should now look like:

```
1 var mecApp = angular.module('mecApp', []);
2
3 mecApp.config(function($interpolateProvider) {
4   $interpolateProvider.startSymbol('[[')
5   .endSymbol(']]');
6 });
```


And, of course, in `userPollCtrl.js` we no longer need the `angular.module` declaration:

```
1 var pollsApp = angular.module('pollsApp', []);
2
3 pollsApp.config(function($interpolateProvider){
4     $interpolateProvider.startSymbol('[[')
5     .endSymbol(']]');
6 });
```

Then everywhere that we referenced `pollsApp` we need to replace with `mecApp`. With that change, everything should continue to work fine. It will allow us to use Angular throughout our entire application as opposed to just for the poll app.

Part 2

Now to make the BIG changes.

Let's start with declaring a `LoggedInCtrl` in a new file called `static/js/loggedInCtrl.js`:

```
1 mecApp.controller('LoggedInCtrl', function($scope) {
2     $scope.show_badges = false ;
3     $scope.show_hide_label = "Show";
4
5     $scope.show = function() {
6         $scope.show_badges = ! $scope.show_badges;
7         $scope.show_hide_label = ($scope.show_badges) ? 'Hide': 'Show';
8     }
9 });
```

The above is a very simple controller where we define two models, `$scope.show_badges` and `$scope.show_hide_label`. We also define a function called `show()`, which will be called when the user clicks on the “Show Achievements” link. So let's look at that link in `templates/main/_jedibadge.html`...

Change this:

```
1 <li><a id="show-achieve" href="#">Show Achievements</a></li>
```

To:

```
1 <li ng-click="show()"><a href="#">[[ show_hide_label ]]  
    Achievements</a></li>
```

We've got two bits of Angular functionality tied into the list item:

1. `ng-click="show()"` calls the `$scope.show()` function from our `LoggedInCtrl`, which toggles the value of `$scope.show_badges` between `true` and `false`. The function also sets the model `$scope.show_hide_label` to either `Hide` (if `show_badges` is `true`) or `Show`.
2. `[[ show_hide_label ]]` substitutes the value of `$scope.show_hide_label` so that the link text will read “Show Achievements” or “Hide Achievements” appropriately.

Part 3

The next piece of the puzzle is the updated template `templates/main/user.html`:

```
1 {% extends "__base.html" %}
2 {% load staticfiles %}
3 {% block content %}
4 <div ng-controller="LoggedInCtrl">
5 <div id="achievements" class="row member-page"
6   ng-class="{hide: show_badges==false}">
7   {% include "main/_badges.html" %}
8 </div>
9 <div class="row member-page">
10   <div class="col-sm-8">
11     <div class="row">
12       {% include "main/_announcements.html" %}
13       {% include "main/_lateststatus.html" %}
14     </div>
15   </div>
16   <div class="col-sm-4">
17     <div class="row">
18       {% include "main/_jedibadge.html" %}
19       {% include "main/_statusupdate.html" %}
20       {% include "djangular_polls/_polls.html" %}
21     </div>
22   </div>
23 </div>
24 </div>
25 {% endblock %}
26 {% block extrajs %}
27   <script src="{% static 'js/userPollCtrl.js' %}"
    type="text/javascript"></script>
```

```

28     <script src="{% static 'js/loggedInCtrl.js' %}"
        type="text/javascript"></script>
29 {% endblock %}

```

The interesting part is lines 4-8 of the template:

```

1 <div ng-controller="LoggedInCtrl">
2 <div id="achievements" class="row member-page"
3   ng-class="{hide: show_badges==false}">
4   {% include "main/_badges.html" %}
5 </div>

```

- **** Line 1**** - Here we declare our controller.
- **** Line 3**** - `ng-class` is a core Angular directive used to add/remove classes dynamically. So, if the value of `$scope.show_badges` is false, then we add the class `hide` to our div; otherwise, remove the class from our div. This is all we need to do to show/hide the div.

Part 4

Finally, we need to update our scripts in `__base.html`:

```

1 <!-- jQuery (necessary for Bootstrap's JavaScript plugins) -->
2 <script src="{% static 'js/jquery-1.11.1.min.js' %}"
    type="text/javascript"></script>
3 <!-- Include all compiled plugins (below), or include individual
    files as needed -->
4 <script src="{% static 'js/bootstrap.min.js' %}"></script>
5 <script src="{% static 'js/angular.min.js' %}"
    type="text/javascript"></script>
6 <script src="{% static 'js/application.js' %}"
    type="text/javascript"></script>
7 <script src="{% static 'js/userPollCtrl.js' %}"
    type="text/javascript"></script>
8 <script src="{% static 'js/loggedInCtrl.js' %}"
    type="text/javascript"></script>

```

And then we can remove the following from `templates/main/user.html`:

```

1 <script src="{% static 'js/userPollCtrl.js' %}"
    type="text/javascript"></script>

```

```
2 <script src="{% static 'js/loggedInCtrl.js' %}"  
   type="text/javascript"></script>
```

That's everything to convert our “Show Achievements” functionality to Angular, so we can remove the corresponding jQuery code from `application.js`.

Angular Forms

Exercise 1

Question: We are not quite done yet with our conversion to Angular, as that register view function is begging for a refactor. A good way to organize things would be to have the current register view function just handle the GET requests and return the register.html as it does now. As for the POST requests, I would create a new users resource and add it to our existing REST API. So rather than posting to /register, our Angular front-end will post to /api/v1/users. This will allow us to separate the concerns nicely and keep the code a bit cleaner. So, have a go at that.

The most straight forward way to do this is to leverage Django Rest Framework. The first thing to do is to create a new file payments/json_views.py. Then add a post_user() function to that file:

```
1 from payments.models import User, UnpaidUsers
2 from payments.forms import UserForm
3 from payments.views import Customer
4 from rest_framework.response import Response
5 from django.db import transaction
6 from django.db import IntegrityError
7 from rest_framework.decorators import api_view
8
9
10 @api_view(['POST'])
11 def post_user(request):
12     form = UserForm(request.DATA)
13
14     if form.is_valid():
15         try:
16             #update based on your billing method (subscription vs
17             #one time)
18             customer = Customer.create(
19                 "subscription",
20                 email=form.cleaned_data['email'],
21                 description=form.cleaned_data['name'],
22                 card=form.cleaned_data['stripe_token'],
23                 plan="gold",
```

```

23         )
24     except Exception as exp:
25         form.addError(exp)
26
27     cd = form.cleaned_data
28     try:
29         with transaction.atomic():
30             user = User.create(
31                 cd['name'], cd['email'],
32                 cd['password'], cd['last_4_digits'])
33
34             if customer:
35                 user.stripe_id = customer.id
36                 user.save()
37             else:
38                 UnpaidUsers(email=cd['email']).save()
39
40     except IntegrityError:
41         form.addError(cd['email'] + ' is already a member')
42     else:
43         request.session['user'] = user.pk
44         resp = {"status": "ok", "url": '/'}
45         return Response(resp, content_type="application/json")
46
47     resp = {"status": "fail", "errors": form.non_field_errors()}
48     return Response(resp)
49 else: # for not valid
50     resp = {"status": "form-invalid", "errors": form.errors}
51     return Response(resp)

```

After all the imports, we declare our view function (on **Line 10**) using the `api_view` decorator and passing in the list of http methods we support - just POST, in our case. The `post_user()` function is basically the same as it was in the `register` view but we can take a few shortcuts because we are using DRF:

- **Line 10** - No need to convert the request to JSON; DRF does that for us; we just load up the `UserForm`.
- **Lines 40-41, 43-44, 46-47** - No need to convert the response back to JSON; just use `rest_framework.response.Response` and it converts for us.

Now we just need to update our url config, so let's add a file `payments/urls.py`

```
1 from django.conf.urls import patterns, url
2
3
4 urlpatterns = patterns(
5     'payments.json_views',
6     url(r'^users$', 'post_user'),
7 )
```

Let's update our main url file, `django_ecommerce/urls.py`, the same way we did when we added `django_polls`...

Add the import:

```
1 from payments.urls import urlpatterns as payments_json_urls
```

And code:

```
1 main_json_urls.extend(payments_json_urls)
```

The file should now look like:

```
1 from django.conf.urls import patterns, include, url
2 from django.contrib import admin
3 from payments import views
4 from main.urls import urlpatterns as main_json_urls
5 from django_polls.urls import urlpatterns as
6     django_polls_json_urls
7 from payments.urls import urlpatterns as payments_json_urls
8
9 admin.autodiscover()
10 main_json_urls.extend(django_polls_json_urls)
11 main_json_urls.extend(payments_json_urls)
12
13 urlpatterns = patterns(
14     '',
15     url(r'^admin/', include(admin.site.urls)),
16     url(r'^$', 'main.views.index', name='home'),
17     url(r'^pages/', include('django.contrib.flatpages.urls')),
18     url(r'^contact/', 'contact.views.contact', name='contact'),
19     url(r'^report$', 'main.views.report', name="report"),
```

```

20     # user registration/authentication
21     url(r'^sign_in$', views.sign_in, name='sign_in'),
22     url(r'^sign_out$', views.sign_out, name='sign_out'),
23     url(r'^register$', views.register, name='register'),
24     url(r'^edit$', views.edit, name='edit'),
25
26     # api
27     url(r'^api/v1/', include('main.urls')),
28 )

```

Finally, we have one last tiny change in our `static/js/RegisterCtrl.js` in order to point it to the new url:

```

1  mecApp.factory("UserFactory", function($http) {
2      var factory = {}
3      factory.register = function(user_data) {
4          return $http.post("/api/v1/users",
5                          user_data).then(function(response)
6                          {
7                              return response.data;
8                          });
9      }
10     return factory;
11 });

```

Easy, right?

Exercise 2

Question: As I said in the earlier part of the chapter, I'm leaving the form validation for `_cardform.html` to the user. True to my word, here it is as an exercise. Put in some validation for credit card, and CVC fields.

Lets look at a simple way to add field validation for our credit card number and CVC fields. Update `django_ecommerce/templates/payments/_cardform.html`:

```

1  <div class="clearfix">
2      <label for="credit_card_number">Credit card number</label>
3      <div class="input">
4          <input class="field" id="credit_card_number" name="cc_num"
5              type="text"
6              ng-model="card.number" ng-minlength="16" required>

```



```

6 </div>
7 <div class="custom-error"
8   ng-show="user_form.cc_num.$dirty && user_form.cc_num.$invalid"
   style="display: none;">
9   Credit card number is invalid:<span
   ng-show="user_form.cc_num.$error.required">value is
   required</span>
10  <span ng-show="user_form.cc_num.$error.minlength">min length of
    sixteen</span>
11 </div>
12 </div>
13
14 <div class="clearfix">
15   <label for="cvc">Security code (CVC)</label>
16   <div class="input">
17     <input class="small" id="cvc" type="text" name="cvc"
        ng-model="card.cvc" required
18     ng-minlength="3">
19   </div>
20   <div class="custom-error" ng-show="user_form.cvc.$dirty &&
    user_form.cvc.$invalid" style="display: none;">
21     CVC is invalid:<span
        ng-show="user_form.cvc.$error.required">value is
        required</span>
22     <span ng-show="user_form.cvc.$error.minlength">min length of
        three</span>
23   </div>
24 </div>

```

This is similar to how we added field validation to our `user_form`. We add the `ng-minlength` attribute and the `required` attribute to each of our input field and then have a custom error div (**Line 7-11, 20-23**) that displays an error message based upon the error that was raised by angular.

SEE ALSO: If you wanted to validate to ensure an input is a number you would think you could use the attribute `type=number` but there apparently is a bug in Angular for those types of validations, so you would have to create your own custom directive to get around that. There is a good explanation of how to do that on blakeembrey.com.

This validation is a bit weak, as you might still get an invalid credit card that is 16 digits long. The Stripe API actually provides validation methods that you can call. Like the link above you would need to create a custom directive or filter to do the validation for you. But there is already an angular-stripe library that does most of that heavy lifting for you. Have a look at the library [on Github](#)

Exercise 3

Question: While the code we added to templates/payments/_field.html is great for our register page, it also affects our sign in page, which is now constantly displaying errors. Fix it!

Remember that _field.html uses the template expression-

```
1 ng-show="{{form.form_name}}.{{field.name}}.$dirty &&
   {{form.form_name}}.{{field.name}}.$invalid"
```

-which relies on the form having a form_name attribute which our SigninForm doesn't have. So edit payments/forms.SigninForm and add the attribute just like you did previously for payments.forms.UserForm. So now your SigninForm should look like:

```
1 class SigninForm(PaymentForm):
2     email = forms.EmailField(required=True)
3     password = forms.CharField(
4         required=True,
5         widget=forms.PasswordInput(render_value=False)
6     )
7     form_name = 'signin_form'
8     ng_scope_prefix = 'signinform'
```

Then update templates/payments/sign_in.html to use that form name, by modifying the `from` tag so it looks like this:

```
1 <form accept-charset="UTF-8" action="{% url 'sign_in' %}"
2     name="{{form_name}}" class="form-signin" role="form"
   method="post">{% csrf_token %}
```

Now your error messages will be gone!

Bonus points if you can find one more form that also needs the same treatment. Give up? Take a look at `contact.forms.ContactView`. It needs the same treatment as our `SigninForm` got.

MongoDB Time!

Exercise 1

Question: Write a MongoTestCase that clears out a MongoDB database prior to test execution/between test runs.

To accomplish this we just need to extend Django's built-in testing capabilities. Let's use `django.test.runner.DiscoverRunner` for that:

```
1 from django.test.runner import DiscoverRunner
2 from mongoengine import connect
3 from mongoengine.connection import disconnect
4 from pymongo import Connection
5 from django.conf import settings
6
7
8 class MongoTestRunner(DiscoverRunner):
9
10     def setup_databases(self, **kwargs):
11         # since we connect to the database in settings.py
12         disconnect()
13
14         db_name = "test_%s" % settings._MONGO_NAME
15         connect(db_name, alias='default')
16         print('Creating test-database:', db_name)
17
18         return db_name
19
20     def teardown_databases(self, db_name, *args):
21         conn = Connection()
22         conn.drop_database(db_name)
23         print('Dropping test-database:', db_name)
```

In the above code example we are overwriting the `setup_databases()` and `teardown_databases()` methods from the parent `DiscoverRunner`.

In the `setup_databases()` method we use `mongoengine` to create a new MongoDB database. The test database has its name derived from the word `test_` and the name defined in our `settings.py` file. Which in our case is `mec-geodata`.

Then in the `teardown_databases()` method we use pymongo to drop all collections in the database.

We can also make a `MongoTestCase` if we need to do any setup/configuration per test. It would look like this:

```
1 from django.test import TestCase
2
3
4 class MongoTestCase(TestCase):
5
6     def _fixture_setup(self):
7         pass
8
9     def _fixture_teardown(self):
10        pass
```

There is a skeleton to use. Feel free to insert your own functionality.

The final step is to get django to use our new `MongoTestRunner`. Which can be done through `settings.py`, by setting `TEST_RUNNER` to be what you want. For us, it looks like this:

```
1 TEST_RUNNER = 'django_ecommerce.tests.MongoTestRunner'
```

This will tell Django to use your runner always. Of course you can use a temporary settings file for this as well, if you want to switch back and forth between database types:

```
1 from django_ecommerce.settings import *
2
3 TEST_RUNNER = 'django_ecommerce.tests.MongoTestRunner'
```

And that can be called from the command line by using something like:

```
1 $ ./manage.py test ../tests --settings=mongo_settings
```

Thats all there is to it. Now you have a way to test MongoDB as well. So get cracking - write some tests!

One Admin to Rule Them All

Exercise 1

Question We didn't customize the interfaces for several of our models, as the customization would be very similar to what we have already done in this chapter. For extra practice, customize both the list view and the change form for the following models:

- **Main. Announcements**
- **Main. Marketing Items**
- **Main. Status reports**

Note both Announcements and Marketing Items will benefit from the thumbnail view and ImageField setup that we did for Badges.

Let's go through the models one at a time. All the below code will go into `main/admin.py`

First up `Main. Announcements`

Let's create the `ModelAdmin` and get the list view working correctly.

```
1 @admin.register(Announcement)
2 class AnnouncementAdmin(admin.ModelAdmin):
3
4     list_display = ('when', 'img', 'vid', 'info_html')
5
6     def info_html(self, announcement):
7         return format_html(announcement.info)
8
9     info_html.short_description = "Info"
10    info_html.allow_tags = True
```

Now we are going to fix the `img` field in the same way we did for badges by making it an `ImageField` and adding a thumbnail view. After doing that our model should now look like this:

```
1 class Announcement(models.Model):
2
3     when = models.DateTimeField(auto_now=True)
```

```

4     img = models.ImageField(upload_to="announce/", null=True,
        blank=True)
5     vid = models.URLField(null=True, blank=True)
6     info = models.TextField()
7
8     def thumbnail(self):
9         if self.img:
10             return u'' %
                (self.img.url)
11         else:
12             return "no image"
13
14     thumbnail.allow_tags = True

```

And we need to update our `AnnouncementAdmin.list_display` to use `thumbnail` instead of `img`. Then you can add / view images from the admin view. Also don't forget to run `./manage.py makemigrations` and `./manage.py migrate` to update the database accordingly.

Then of course just like for Badges we will have to update our main site to point to the new images. So we update `templates/main/_announcements.html` to load the image from the media URL by changing this line:

```

1 

```

to

```

1 

```

With that images should display nicely on the front end.

How about the videos? Well if you remember from way back early in the book, we used a library called 'django-embed-video' to add the links to youtube videos to our user page. Well as it turns out that same library offers Admin support. Bonus! We need to do a couple of things.

First, update our model again to use the `EmbedVideoField` which is just a lightweight wrapper around a `URLField`:

```

1 from embed_video.fields import EmbedVideoField
2
3 class Announcement(models.Model):
4
5     when = models.DateTimeField(auto_now=True)

```

```

6     img = models.ImageField(upload_to="announce/", null=True,
7                             blank=True)
8     vid = EmbedVideoField(null=True, blank=True)
9     #----- snip ---- #

```

This is necessary so we can get the correct widget displayed in our Admin View. Also, this will require you to make and run migrations again so go ahead and do that.

Now let's modify our AnnouncementAdmin to use `embed_video.admin.AdminVideoMixin`. All we have to change is the class definition:

```

1 from embed_video.admin import AdminVideoMixin
2
3 @admin.register(Announcement)
4 class AnnouncementAdmin(AdminVideoMixin, admin.ModelAdmin):

```

Once that Mixin is in place, it will take care of finding any fields of type `EmbedVideoField` and will create a custom widget that displays the video and the url for the video in the change form. So if you now look go to <http://127.0.0.1:8000/admin/main/announcement/2/> you will see a playable thumbnail of the video plus a text box to input the url for the video.

There you go. That should do it for Announcements. Feel free to tweak anything else you like, but that should get you most of the functionality you need.

Main.Marketing Items

Marketing Items will end up being very similar to Announcements. Let's start off with the model updates, since we now have three models that will likely use the thumbnail view, it time to refactor.

First let's create a ThumbnailMixin:

```

1 class ThumbnailMixin(object):
2     '''use this mixin if you want to easily show thumbnails for an
3     image field
4     in your admin view
5     '''
6
7     def thumbnail(self):
8         if self.img:
9             return u'' %
10                (self.img.url)
11         else:

```

```

10         return "no image"
11
12     thumbnail.allow_tags = True

```

Then we can add the ThumbnailMixin to the models MarketingItem, Announcement, and Badge. With that here is what MarketingItem will look like:

```

1 class MarketingItem(models.Model, ThumbnailMixin):
2     img = models.ImageField(upload_to="marketing/")
3     heading = models.CharField(max_length=300)
4     caption = models.TextField()
5     button_link = models.CharField(null=True, blank=True,
6                                   max_length=200, default="register")
7     button_title = models.CharField(max_length=20, default="View
8                                   details")

```

As per usual, don't forget to run migrations:

```

1 $ ./manage.py makemigrations
2 $ ./manage.py migrate

```

For the Admin view, let's do something a little different. If you remember from the Bootstrap chapter we created a template tag marketing__circle_item to display marketing items. So why not use that tag in the admin view so Administrators can immediately see what the marketing item will look like on the "live" site. Below is the admin class that will make that happen:

```

1 from django.template.loader import render_to_string
2
3 @admin.register(MarketingItem)
4 class MarketingItemAdmin(admin.ModelAdmin):
5
6     list_display = ('heading', 'live_view')
7
8     def live_view(self, mi):
9         return
10         render_to_string("main/templatetags/circle_item.html",
11                        {'marketing_items': (mi,)})
12
13     live_view.short_description = "Rendered Template"
14     live_view.allow_tags = True

```


The “magic” is in the `live_view` function which simply renders the template associated with the `marketing__circle_item` template tag and passes in the `MarketItem` `mi` as the context for the template.

Try it out, you’ll see the fully rendered marketing item in the admin list view!

One more to go...

Main.StatusReports

This one is even simpler. No need to change the model. Let’s just create an Admin class:

```
1 from django.forms import Textarea
2 from django.db import models
3
4 @admin.register(StatusReport)
5 class StatusReportAdmin(admin.ModelAdmin):
6
7     list_display = ('status', 'user', 'when')
8
9     formfield_overrides = {
10         models.CharField: {'widget': Textarea(attrs={'rows': 4,
11             'cols': 70})},
12     }
```

The `formfield_overrides` is the only thing new here. What it says is to change the widget used to edit any field that is a `models.CharField` to a `Textarea` and pass `attrs` to `Textarea.__init__()`. This will give us a large `Textarea` to edit the status message. We want to do this since the `max_length` for the field is 200 and it’s just easier to type all that in a `Textarea`.

Okay. That’s it for the models. Things should be looking better now.

Exercise 2

Question: For our `Payments.Users` object in the change form you will notice that badges section isn’t very helpful. See if you can change that section to show a list of the actual badges so it’s possible for the user to know what badges they are adding / removing from the user.

There are a few ways to do this. Probably the simplest is to use `filter_horizontal` or `filter_vertical` which will give you a nice jQuery select that shows what you have chosen. To do that just add a single line to the bottom of your `UserAdmin`. Here is the full `UserAdmin`:

```

1 @admin.register(User)
2 class UserAdmin(admin.ModelAdmin):
3
4     list_display = ('name', 'email', 'rank', 'last_4_digits',
5                     'stripe_id')
6     ordering = ('-created_at',)
7     fieldsets = (
8         ('User Info', {'fields': ('name', 'email', 'rank',)}),
9         ('Billing', {'fields': ('stripe_id',)}),
10        ('Badges', {'fields': ('badges',)}),
11    )
12     filter_horizontal = ('badges',)

```

That will give us a nice select, but all options will just say badge `object`. We can change this like we changed the `SelectWidget` by adding a `__str__` function to our `Badge` model. Such as:

```

1 def __str__(self):
2     return self.name

```

Testing, Testing, and More Testing

Exercise 1

Question: Write a Page Object for the Registration Page, and use it to come up with test for successful and unsuccessful registrations. You'll need to use the LiveServerTestRunner for the tests to execute successfully.

We'll start by creating our page object which is very similar to the page object we created for the login page:

```
1 class RegisterPage(SeleniumPage):
2
3     name_textbox = SeleniumElement((By.ID, 'id_name'))
4     email_textbox = SeleniumElement((By.ID, 'id_email'))
5     pwd_textbox = SeleniumElement((By.ID, 'id_password'))
6     ver_pwd_textbox = SeleniumElement((By.ID, 'id_ver_password'))
7     cc_textbox = SeleniumElement((By.ID, 'credit_card_number'))
8     cvc_textbox = SeleniumElement((By.ID, 'cvc'))
9     expiry_month_dd = SeleniumElement((By.ID, 'expiry_month'))
10    expiry_year_dd = SeleniumElement((By.ID, 'expiry_year'))
11    register_title = SeleniumElement((By.CSS_SELECTOR,
12                                     '.form-signin-heading'))
13    register_button = SeleniumElement((By.ID, 'user_submit'))
14    errors_div = SeleniumElement((By.CSS_SELECTOR, ".alert"))
```

Nothing special here; we are just defining our elements like we did with SignInPage.

Next are a few properties:

```
1 @property
2 def error_msg(self):
3     '''the errors div has a 'x' to close it
4     let's not return that
5     '''
6     return self.errors_div.text[2:]
7
8 @property
9 def rel_url(self):
10    return '/register'
```

Again, similar to 'SignInPage' do note for the error_msg property we trim off the first two characters since they are used to show the X a user can click on to hide the errors.

Continuing with the rest of the class...

```
1 def go_to(self):
2     self.driver.get('%s%s' % (self.base_url, self.rel_url))
3     assert self.register_title.text == "Register Today!"
4
5 def mock_geoloc(self, lat, lon):
6     self.driver.execute_script('''angular.element($('#id_email'))
7         .scope().geoloc = {
8             'coords': {'latitude': '%s', 'longitude': '%s'}};
9     angular.element(document.body).injector().get('$rootScope').$apply();
10    ''' % (lat, lon))
```

Okay. Let's look at that `mock_geoloc()`. We talked about this in the *You need to run some JavaScript code* section of the chapter. Basically, we are setting the value of the `$scope.geoloc` to whatever the user passes in. This way we can test that geolocation is working, because with the default browser you get when you run selenium, the geolocation functionality probably won't work correctly.

```
1 def set_expiry_month(self, month_as_int):
2     selector = "option[value='%s']" % (month_as_int)
3     self.expiry_month_dd.find_element_by_css_selector(selector).click()
4
5 def set_expiry_year(self, year_as_int):
6     selector = "option[value='%s']" % (year_as_int)
7     self.expiry_year_dd.find_element_by_css_selector(selector).click()
```

What are we doing here?

1. Setting our drop downs, by clicking on the option tag in the drop down. Note here that normally we say `webdriver.find_element...`, but now we are in effect saying `webelement.find_element...`. When we do this, our find will be restricted to only search through the child DOM objects of the web element. This is how we can ensure that we are finding only the option tags that belong to our drop down. These functions could be refactored up in the `SeleniumElement` class... consider that extra credit.

```
1 def do_reg(self, name, email, pwd, pwd2, cc, cvc,
2     expiry_month, expiry_year, lat=1, lon=2):
3
4     self.name_textbox.send_keys(name)
5     self.email_textbox.send_keys(email)
```

```

6     self.pwd_textbox.send_keys(pwd)
7     self.ver_pwd_textbox.send_keys(pwd2)
8     self.cc_textbox.send_keys(cc)
9     self.cvc_textbox.send_keys(cvc)
10    self.set_expiry_month(expiry_month)
11    self.set_expiry_year(expiry_year)
12    self.mock_geoloc(lat, lon)
13    self.register_button.click()

```

This mimics the `doLogin()` function we have on the `SignInPage`. It just fills out all the necessary fields, and, as a bonus, mocks out the geolocation value as well.

Now onto the actual test.

```

1  class RegistrationTests(TestCase):
2
3      @classmethod
4      def setUpClass(cls):
5          from selenium.webdriver.firefox.firefox_profile import
             FirefoxProfile
6          profile = FirefoxProfile()
7          profile.set_preference('geo.prompt.testing', True)
8          profile.set_preference('geo.prompt.testing.allow', True)
9          cls.browser = webdriver.Firefox(profile)
10
11         cls.browser.implicitly_wait(10)
12         super(RegistrationTests, cls).setUpClass()

```

This is the hardest part of the test, and probably sent you googling for a bit. That's all part of the programming game. If you ran the test as you may have noticed the browser kept asking you if it was okay to give your location. (Whether or not the browser asks you this is controlled by a setting in your firefox profile). Luckily with selenium we have access to the Firefox profile and we can set whatever values we want. These settings above will ensure that you are not prompted with that annoying popup anymore.

```

1  @classmethod
2  def tearDownClass(cls):
3      cls.browser.quit()
4      super(RegistrationTests, cls).tearDownClass()
5
6  def setUp(self):

```

```
7 self.reg = RegisterPage(self.browser, "http://" +
    127.0.0.1:9001)
```

And now finally the actual tests:

```
1 def test_registration(self):
2     self.reg.go_to()
3     self.reg.do_reg(name="somebodynew", email="test@newtest.com",
4                     pwd="test", pwd2="test", cc="4242424242424242",
5                     cvc="123", expiry_month="4", expiry_year="2020")
6     self.assertTrue(
7         self.browser.find_element_by_id("user_info").is_displayed())
```

Nothing special here we are just filling out the form, using our PageObject and checking to make sure the `user_info` element is displayed (which should only be on the logged in members page).

```
1 def test_failed_registration(self):
2     self.reg.go_to()
3     self.reg.do_reg(name="somebodynew", email="test@newtest2.com",
4                     pwd="test", pwd2="test2", cc="4242424242424242",
5                     cvc="123", expiry_month="4", expiry_year="2020")
6     self.assertIn("Passwords do not match", self.reg.error_msg)
```

And for our final test we fill out all the information (with passwords that don't match) and check to see that we get the right error message. We could also come up with other combinations of this test to check the various error messages one might encounter.

One final note, even though `test_failed_registration()` comes later in the file than `test_registration()`, it runs first. Why? by default the test runner will run the tests in alphabetical order. This useful tidbit is helpful if you want your test to run in a certain order. For example, if you run all your failed registrations before your successful registration, this will improve the execution speed of the tests.

While this is the only exercise in this chapter, feel free to continue the testing and see if you can come up with test cases for the major functionality pieces. It is good practice. Cheers!

Deploy

Exercise 2

Question: **Didn't think you were going to get out of this chapter without writing any code, did you? Remember back in the Fabric section when we talked about creating another function to update all of your configuration files? Well, now is the time to do that. Create a function called `update_config()` that automatically updates your Nginx, Supervisor and Django config / setting files for you.**

When the function is complete you should be able to execute:

```
1 $ fab integrate update_app update_config
```

Bonus points if you can get it all to work so the user only has to type:

```
1 $ fab ci
```

For the first part - e.g, the `update_config()` function - our function should look something like this:

```
1 def update_config():
2     with cd("/opt/mec_env/mec_app/deploy"):
3         run("cp settings_prod.py
4             ../django_ecommerce/django_ecommerce/")
5         run("cp supervisor/mec.conf /etc/supervisor/conf.d/")
6         run("cp nginx/sites-avaliable/mec
7             /etc/nginx/sites-available/")
8         run("/etc/init.d/supervisor restart")
9         run("/etc/init.d/nginx restart")
```

Not a lot new above. Basically all I did was start off in the deploy directory, copy the three files to their necessary places (unix will automatically overwrite on copy), and then I restarted both supervisor and nginx to make sure any configuration changes get updated. With that you can now run

```
1 $ fab integrate update_app update_config
```

For the bonus section... well there is really nothing to it. Keep in mind that fabric is just python so you can structure it anyway you like. Which means you just need to create a function called `ci` that in turn calls the other three functions.

Below is a listing of the entire `fabfile.py` including the new `ci` function.

```

1 from fabric.api import env, cd, run, prefix, lcd, settings, local
2
3 env.hosts = ['<< server ip>>']
4 env.user = 'root'
5
6
7 def ci():
8     integrate()
9     update_app()
10    update_config()
11
12
13 def update_app():
14     with cd("/opt/mec_env/mec_app"):
15         run("git pull")
16     with cd("/opt/mec_env/mec_app/django_ecommerce"):
17         with prefix("source /opt/mec_env/bin/activate"):
18             run("pip install -r ../requirements.txt")
19             run("./manage.py migrate --noinput")
20             run("./manage.py collectstatic --noinput")
21
22
23 def update_config():
24     with cd("/opt/mec_env/mec_app/deploy"):
25         run("cp settings_prod.py
26             ../django_ecommerce/django_ecommerce/")
27         run("cp supervisor/mec.conf /etc/supervisor/conf.d/")
28         run("cp nginx/sites-avaliable/mec
29             /etc/nginx/sites-available/")
30         run("/etc/init.d/supervisor restart")
31         run("/etc/init.d/nginx restart")
32
33 def integrate():
34     with lcd("../django_ecommerce/"):
35         local("pwd")
36         local("./manage.py test ../tests/unit")
37
38     with settings(warn_only=True):
39         local("git add -p && git commit")

```



```
39
40     local("git pull")
41     local("./manage.py test ../tests")
42     local("git push")
```