Real Python Part 3: Advanced Web Development with Django 1.6

Jeremy Johnson

Contents

List of Figures

Chapter 1

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 :)

```python
# tinyp2p.py 1.0 (documentation at http://freedom-to-tinker.com/tinyp2p.html)
import sys, os, SimpleXMLRPCServer, xmlrpclib, re, hmac # (C) 2004, E.W. Felten
ar,pw,res = (sys.argv,lambda u:hmac.new(sys.argv[1],u).hexdigest(),re.search)
pxy,xs = (xmlrpclib.ServerProxy,SimpleXMLRPCServer.SimpleXMLRPCServer)
def ls(p=""):return filter(lambda n:(p=="")or res(p,n),os.listdir(os.getcwd()))
if ar[2]!="client": # license: http://creativecommons.org/licenses/by-nc-sa/2.0
  myU,prs,srv = ("http://"+ar[3]+":"+ar[4], ar[5:],lambda x:x.serve_forever())
  def pr(x=[]): return ([(y in prs) or prs.append(y) for y in x] or 1) and prs
  def c(n): return ((lambda f: (f.read(), f.close()))(file(n)))[0]
  f=lambda p,n,a:(p==pw(myU))and(((n==0)and pr(a))or((n==1)and [ls(a)])or c(a))
  def aug(u): return ((u==myU) and pr()) or pr(pxy(u).f(pw(u),0,pr([myU])))
  pr() and [aug(s) for s in aug(pr()[0])]
  (lambda sv:sv.register_function(f,"f") or srv(sv))(xs((ar[3],int(ar[4]))))
for url in pxy(ar[3]).f(pw(ar[3]),0,[]):
  for fn in filter(lambda n:not n in ls(), (pxy(url).f(pw(url),1,ar[4]))[0]):
    (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, the above is **NOT** what we are going to be doing. While it is a cool example of the power of Python, it's painful to try to understand what is going on with the code and nearly impossible to maintain without pulling your hair out. I'm being a bit unfair to whoever it is that wrote this, an Anonymous user on some Linux Forum, because it was intentionally written this way to fit the entire program in 15 lines... 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.

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

In fact, in 2013 enrollment in Computer Science degree programs for US Universities rose by 29% from the previous year. With everybody jumping in on the game, it may seem that the market will be saturated with would-be employees all vying for the same number of limited jobs. However, most high tech companies will tell a different story: the story of way too many résumés 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 book 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. That is, 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".

We will dive into this idea of well-factored code and quality starting from Chapter 2 which covers **Software Craftsmanship**, Chapter 3 which will focus on **Unit Testing** and Chapter 4 which covers **Test Driven Development**, Chapter 5 on **Git and Version Control** and Chapter 6 on **The Art of the Upgrade**.

It's important to start with 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 are the process of deliberately using a number of proven techniques and practices, and we are going to teach you those techniques and practices.

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

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

But in today's world of HTML5 and mobile browsers, JavaScript is everywhere, and Django just isn't enough. To develop a website from end to end, you must know about JavaScript frameworks, about REST-based APIs, about Bootstrap and responsive design, about NoSQL databases and all the other technologies that make it possible to compete at the global level. That is why this book will not just

cover Django exclusively, but it will cover all the pieces that you need to know to develop modern web applications from end to end, including not just development but testing, tooling, and deployment as well.

By the end of this book you're going to be able to develop a REAL web based application that might actually 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 book we will be focusing on a single application, so you can see how applications evolve and understand what goes into making a REAL application. We will be starting from the Django sample application introduced in the second course of this series, "Web Development with Python". If you haven't read that book, don't worry - there was only one chapter dedicated to the specific application, and all the code is in the associated repo of this book. To give you an idea of where we are going, at the start of this book your application will look like this:

Figure 1.1: Initial look

It's a nice start, but admittedly not something that you could take to yCombinator and get funding. By chapter 10 that same application will look something like:

Figure 1.2: Chapter 10

And by the end of the book... well, you're just going to have to read through the book and find out. :)

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 in this book and do my best to impart the most important and useful things I have learned from my Software Development career in the hopes that you can benefit from it and grow as a software developer. I appreciate any and all feedback, so don't be shy, send me your thoughts.

Thank you so much for purchasing this book, and a special thanks goes out to all the early Kickstarter supporters that had faith in the Real Python team and backed us even before this book was finished.

Now let's build something REAL!

License

This e-book is copyrighted and licensed under a Creative Commons Attribution- NonCommercial-NoDerivs 3.0 Unported License. This means that you are welcome to share this book and use it for any non-commercial purposes so long as the entire book 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 book 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 Alpha release, we do not have all the conventions in place. We are working on it. Patience.

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:

1. Notes:

   NOTE: This is a note filled in with bacon impsum 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.

2. Warnings:

   WARNING: This is a warning also filled in with bacon impsum. 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.

3. See Also:

   SEE ALSO: This is a see also box with more tasty impsum. 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.

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 in Mac OS X v. 10.8, Windows XP, Windows 7, and Linux Mint 14.

Chapter 2

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. A novice engineer 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. A true craftsman insists on quality throughout all aspects of a product; it is not good enough to just deliver requirements. McDonalds delivers requirements, but a charcoal-grilled, juicy burger cooked to perfection on a Sunday afternoon and shared with friends... that's craftsmanship.

In this book we are going to 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**
- 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 are going to start off with the most important of these practices and the first that any engineer looking to improve her craft should learn: *Automated Unit Testing*.

Automated unit testing is the foundation upon which good Software Craftsmanship practices are built. Since this is a book on Django development, and the 3rd in the Real Python series, we will start by taking a look at the Django MVP app from the last chapter of **Real Python Part 2: Intro to Web Development** by Michael Herman. By adding unit tests to this application we will start to see how unit testing is done and how it can make you a better software developer.

If you haven't read the **Real Python Part 2** book yet and you want to get a broad overview of several of the most popular web frameworks out there, you really should have a look at it here. If you just want to get started on this book, you can grab a copy of the Django MVP application by using the supplied git repo for this book.

The repo file should be called repo.zip. Go ahead and extract the repo and then switch to the ch1.1 tag by using a command like this:

```
$ unzip repo.zip repo
$ cd repo
$ git checkout tags/ch1.1
```

Unit Testing in Django

Django makes unit testing pretty straight-forward. Unit testing in Django is generally accomplished by using the 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 simple 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 the 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:

1. Receive a web request.
2. Route that request to the appropriate Django view
3. Do some business logic.
4. Including (maybe) accessing a datastore through a Django model.
5. Return an HTTP Response.

Thus, we want to test:

1. 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. By using the standard `unittest` style or the `DocTest` style.

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.

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 in-built tests. If you have worked through the previous book, you should have a virtual environment setup and ready to go for the MVP app. Make sure you have that activated and ready to go by running:

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

If you're starting from the downloaded repo do the following: First ensure you have virtualenvwrapper (this is a wrapper around virtualenv that makes it easier to use) installed:

```
$ pip install virtualenvwrapper
```

Then set up the virtual environment with virtualenvwrapper:

```
1  $ cd /django_ecommerce
2  $ mkvirtualenvwrapper django_mvp_app
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**: **Requirements.txt for dependency management**
>
> Using a requirements.txt file as part of a Python project is a great way to manage dependencies. A requirements.txt file is a simple text file that list a dependency per line of your project; pip will read this file and all the dependencies will then be downloaded and installed. After you have your virtual environment set up and all your dependencies installed, just type `pip freeze > requirements.txt` and that will create the `requirements.txt` file for you that can later be used to quickly install the project dependencies for you. Then check it 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

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  $ python manage.py test
```

Note: 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 :)

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

```
1  ...................E............................E................
2  ............................................s..........................
3  ...........................................................x.....
4  .......................................................................
5  .......................................................................
```

If you don't get any errors then everything is all good and we can move forward.

> **WARNING**: **Mac Users and the dreaded UTF / localization error**
>
> 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:

export LC_ALL=en_US.UTF-8

        export LANG=en_US.UTF-8

    Feel free to use another language if you prefer, and it's also probably easiest to put these
    two lines in your .bash_profile so you don't have to retype them all the time.

    With that out of the way, you should see the following output for your `./manage.py test` command

```
$ ./manage.py test
Creating test database for alias 'default'...
.............................................................................
....................................s...............................
..............................................................x......
.............................................................................
----------------------------------------------------------------------
Ran 529 tests in 12.625s

OK (skipped=1, expected failures=1)
Destroying test database for alias 'default'...
```

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

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`:

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

    url(r'^admin/', include(admin.site.urls)),
    url(r'^pages/', include('django.contrib.flatpages.urls')),
    url(r'^contact/', 'contact.views.contact', name='contact'),

    url(r'^sign_in$', views.sign_in, name='sign_in'),
    url(r'^sign_out$', views.sign_out, name='sign_out'),
    url(r'^register$', views.register, name='register'),
    url(r'^edit$', views.edit, name='edit'),

)
```

Our first set of tests will be to make sure this routing works correctly and 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). Furthermore, it 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.

Here is what our first test might look like:

main/tests.py

```
from django.test import TestCase
from django.core.urlresolvers import resolve
from .views import index


class MainPageTests(TestCase):

    def test_root_resolves_to_main_view(self):
        main_page = resolve('/')
        self.assertEqual(main_page.func, index)
```

Here we are using Django's own urlresolver to test that the / url resolves to the `main.views.index` function. This way we know that our routing is working correctly.

Now if we run:

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

We should see the test pass. Great! 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.tests.TestCase`. A basic test for this might look like:

```python
def test_returns_appropriate_html(self):
    index = self.client.get('/')
    self.assertEquals(index.status_code,200)
```

NOTE: **Note on Test Structure**

In Django 1.5 the default place to put tests is in a file called test.py inside the applicaiton directory. That is why these tests are in `main/tests.py`. For the remaining of this chapter we are going to stick to that standard and put all tests in `<app>/tests.py` files. In a later chapter we will discuss more on the advantages / disadvantages of this test structure. For now lets focus on testing.

Testing Templates and Views

The issue with this test is that even if you're returning the incorrect HTML, it will still pass. It's better to verify the actual template used:

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

assertTemplateUsed is one of those helper functions provided by Django. It just checks to see if an HTTPResponse object was generated by a specific template.

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

```
from django.shortcuts import render_to_response

def test_returns_exact_html(self):
    index = self.client.get("/")
    self.assertEquals(index.content,
            render_to_response("index.html").content)
```

In the above test, the render_to_response() function is used to run our index.html template through the Django template engine and ensure that the result is the same we get returned when calling the / url.

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. This is because the index function actually performs three separate functions:

1. Check the session to see if a user is logged in.
2. If the user is logged in, query the database to get the user information.
3. Return the appropriate HTTP Response.

Now is the time that we may want to argue for refactoring the code to make this function 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 System Test instead of a Unit Test, because we are now testing several parts of the system as opposed to just one individual unit. Forgetting about the distinction for the time being; let's look at how we might do that. We will break it down into three steps:

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....

Step 3. Verify that we get the appropriate response returned:

```python
def test_index_handles_logged_in_user(self):



    self.assertTemplateUsed(resp, 'user.html')
```

We have seen that before. All we want to do is verify that when there is a logged in user, we return the *user.html* template instead of the normal *index.html* template.

Step 2. Ensure there is a user in the database so our lookup works.

```python
def test_index_handles_logged_in_user(self):
    #create the user needed for user lookup from index page
    from payments.models import User
    user = User(
            name = 'jj',
            email = 'j@j.com',
        )
    user.save()

    ...snipped code....

    #verify it returns the page for the logged in user
    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 for the system. While we haven't mentioned it before, it's important to understand what exactly happens when you run a Django unittest. If you will allow a brief digression...

Behind the Scenes – Running unittests with Django

When running unit tests in Django, we have several classes we can inherit from to make our testing easier. They include:

1. `unittest.TestCase` - The default Python test case. Provides:

   - No association with Django. Which means 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'. Provides:

   - Asserting that a particular exception was raised.
   - Asserting on Form Fields, for example to check field validations.
   - Various HTML Asserts like `assert.contains` which check for an HTML fragment.
   - Loads Django settings and allows for loading custom Django settings.

3. `django.test.TestCase` or `django.test.TransactionTestCase` - These two test cases are similar accept with regards to how they deal with the database. They provide:

   - Automatic loading of fixtures (i.e. loading data prior to running tests).
   - Creates a TestClient Instance `self.client` useful for things like resolving URLS.
   - Provides a ton of Django-specific asserts.
   - `TransactionTestCase` - allows for testing database transactions and automatically resets database at the end of the test run by truncating all tables.
   - `TestCase` - wraps each test case in a transaction. This transaction is 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:

   - Does basically the same thing as `TransactionTestCase`
   - Also fires up an actual web server so you can perform GUI tests

With each one of these testing base classes you get more parts of Django loaded at a cost of test execution speed as well as test complexity. From a purely performance-oriented point of view you would want to 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.

Keep in mind that from a puritan point of view, unless your inheriting from `unittest.TestCase` or perhaps `django.test.SimpleTestCase` you're not really doing unit testing. *Django's "unit testing" framework is really a system testing framework, as it is very difficult to isolate different components when using* `django.test.TestCase` *for example.* Take the following tests:

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

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

- The Django Test Client is firing up
- The Django Router is called from `self.client.get`
- A Django request object is automatically created and passed on in `self.client.get`
- A Django response object is returned and `assertTemplateUsed` is dependent on that
- If there is any third party middleware it will be called
- App middleware will also be called
- Context Managers will be called

So really, there is a lot going on here, and because of that we can't really call this a unit test. We could however rewrite this test. Assuming we were inheriting from `django.test.SimpleTestCase`, the test case might look like:

```
from django.test import RequestFactory


def test_uses_index_html_template_2(self):
    #create a dummy request
    request_factory = RequestFactory()
    request = request_factory.get('/')
    request.session = {} #make sure it has a session associated

    #call the view function
    resp = index(request)

    #check the response
    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. 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. This has the following consequences:

- Tests will likely run faster, because there is less going on.
- 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!

Both system and unit tests have their merits, and you can use both types together, in the same test suite so it's not an all-or-nothing thing. 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, that 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.

Mocks, Fakes, Test Doubles, Stubs...

Coming back to our early example, we should now understand why we need to create a user to test if our `main.views.index` function is working. (Because the `django.test.TestCase` class that we are inheriting from will clear out the database prior to each run). We are testing against the actual database. But what about the Request object that is passed into the view? Here is the view code we were testing:

```python
def index(request):
    uid = request.session.get('user')
    if uid is None:
        return render_to_response('index.html')
    else:
        return render_to_response('user.html', {'user':
    User.objects.get(pk=uid)})
```

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

There is a technique in unit testing for doing this. There are many names for this technique; it can be called mocking, faking, 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 that simulates a real object for the purposes of ease of testing and test execution speed.*

Visually you can think of it as shown in figure 2.1:

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 run very frequently. I.e. 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`. Using the `RequestFactory` to create a mock request looks like this:

```python
#create a dummy request
from django.test import RequestFactory
request_factory = RequestFactory()
request = request_factory.get('/')
request.session = {} #make sure it has a session associated
```
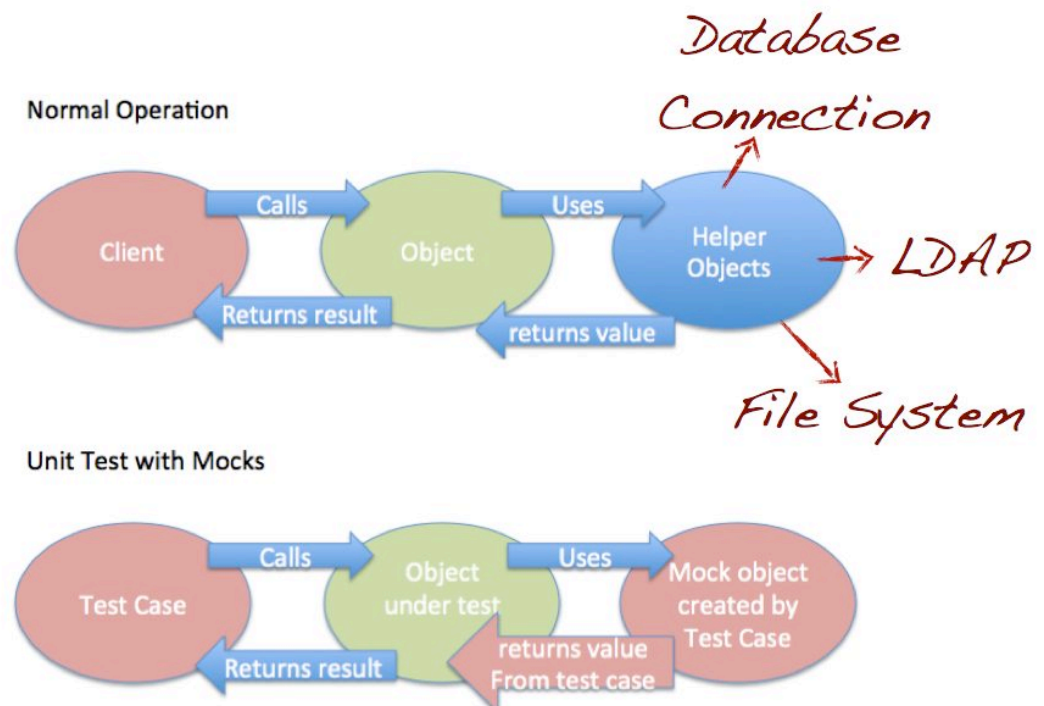
Figure 2.1: mocking example

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

```python
from payments.models import User
from django.test import RequestFactory

def test_index_handles_logged_in_user(self):
    #create the user needed for user lookup from index page
    user = User(
    name = 'jj',
    email = 'j@j.com',
    )
    user.save()

    #create a Mock request object, so we can manipulate the session
    request_factory = RequestFactory()
    request = request_factory.get('/')
    #create a session that appears to have a logged in user
    request.session = {"user" : "1"}

    #request the index page
    resp = index(request)

    #verify it returns the page for the logged in user
    self.assertEquals(resp.content,
                      render_to_response('user.html', {'user': user}).content)
```

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

At this point we have the test completed for our main app. You can check the code out from GitHub by doing:

```
$ git checkout tags/ch1.2
```

Looking back at the tests, we can see there are some discrepancies, as some of the tests use our RequestFactory mock and others don't. Let's clean up the tests a bit so they all use request mocks. This should make the tests run faster and allow them to be more isolated. There is however one test, `test_root_resolves_to_main_view`, that is intended to test the routing, and thus we don't want to mock out anything for that test. For that test we really do want to go through all the Django machinery and ensure it's wired up correctly.

To clean up the tests, the first thing we should do is create a setup function which will be run prior to the tests being executed. In the setup function we will create the mock. This way we don't have to

28

keep recreating it for each of our tests:

```python
@classmethod
def setUpClass(cls):
    from django.test import RequestFactory
    request_factory = RequestFactory()
    cls.request = request_factory.get('/')
    cls.request.session = {}
```

Notice the above function `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. See the Python docs for more info. If we instead want the setup code to run before each test, we could write it as:

```python
def setUp(self):
    from django.test import RequestFactory
    request_factory = RequestFactory()
    self.request = request_factory.get('/')
    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.

```python
class MainPageTests(TestCase):

    @classmethod
    def setUpClass(cls):
        from django.test import RequestFactory
        request_factory = RequestFactory()
        cls.request = request_factory.get('/')
        cls.request.session = {}

    def test_root_resolves_to_main_view(self):
        main_page = resolve('/')
        self.assertEqual(main_page.func, index)

    def test_returns_appropriate_html_respos_code(self):
        resp = index(self.request)
        self.assertEquals(resp.status_code,200)

    def test_returns_exact_html(self):
        resp = index(self.request)
        self.assertEquals(resp.content,
                          render_to_response("index.html").content)

    def test_index_handles_logged_in_user(self):
        #create the user needed for user lookup from index page
        from payments.models import User
        user = User(
```

```
27            name = 'jj',
28            email = 'j@j.com',
29        )
30        user.save()
31
32        #create a session that appears to have a logged in user
33        self.request.session = {"user" : "1"}
34
35        #request the index page
36        resp = index(self.request)
37
38        #ensure we return the state of the session back to normal so
39        #we don't affect other tests
40        self.request.session = {}
41
42        #verify it returns the page for the logged in user
43        expectedHtml = render_to_response('user.html', {'user':user}).content
44        self.assertEquals(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 url routing or middleware, so the test should be more robust and easier
to debug should there be a problem.

We still have one test, though, `test_index_handles_logged_in_user` that has a dependency on the
database. If we wanted to get rid of that dependency we could also mock out our call to the database.
First we will need to install the mock library:

```
1  $ pip install mock
```

Then our test might look like this:

```
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          #set the object function to our mock
10         user_mock.objects = mock.Mock()
11
12         #tell the mock what to do when called
13         config = {'get.return_value':mock.Mock()}
14         user_mock.objects.configure_mock(**config)
15
16
17         #run the test
```

```
18          resp = index(self.request)
19
20          #ensure we return the state of the session back to normal
21          self.request.session = {}
22
23          expectedHtml = render_to_response('user.html', {'user':user}).content
24          self.assertEquals(resp.content,expectedHtml)
```

Let's have a look at what we are doing.

- First `with mock.path('main.views.User') as user_mock:` says that when we come across a User objet in the `main.views` module, call our mock object instead of the actual user object.

- Then we mock out `User.objects` in doing so we can prevent any database access from happening.

- 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 a dummy mock.

- Finally we just run our test as normal and assert it does the correct thing.

This way our view method will get back the value it needs from the database and everything will be good - and we will never actually touch the database.

In fact, since we now have no reliance on the Django testing framework at all, we could inherit directly from unittest.TestCase. Many people in the Python community swear this is the way to go and you should mock out everything. I'm not so sure that is the best way to go, and we will talk a bit more about why in the next section on testing 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.

Testing Models

In the previous section we used the Mock library to mock out the Django ORM so we could test a Model without hitting the database. This can make tests faster, but I'm of the opinion that it is NOT a good approach to testing models. *Let's face it, ORM or no ORM, your models are tied to a database. Meaning they are intended to work with a database, and testing them without a database can often lead to a false sense of security.* It has been my experience that the vast majority of model-related errors have to do with the back-end database. Generally errors include:

- Database is out of date with Model.
- Storing incorrect datatypes, i.e. trying to store a string in a Number column.
- Referential / Integrity Constraints (i.e. storing two rows with same id).
- Join issues.

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, and these issues all come from the database. This is why I say testing models with mocks provides a false sense of security. It lets all of these issues slip through the cracks. I'm not saying you should never use mocks when testing models, I'm just saying be very careful you're not convincing yourself you have fully tested your model, when you haven't really tested it.

One thing to point out here though is that you may not be running your unit tests on the same database you are using in production. This can also be a source of errors slipping through the cracks. In a later chapter we will have a look at using Travis CI to automate testing after each commit including how run the test on multiple databases. For now just keep it in mind and let's look at some techniques to test a model without using mocks.

Technique 1 - Just don't test them

You may laugh, but I'm not joking. Many models are simply just a collection of fields and rely on standard Django functionality. We really shouldn't be testing standard Django functionality as we can always just run `manage.py test` and all the tests for standard Django functionality will be tested for us. Specifically if we look at the `payments.models.User` model from the last section:

```python
from django.db import models
from django.contrib.auth.models import AbstractBaseUser

class User(AbstractBaseUser):
    name = models.CharField(max_length=255)
    email = models.CharField(max_length=255, unique=True)
    #password field defined in base class
    last_4_digits = models.CharField(max_length=4, blank=True, null=True)
    stripe_id = models.CharField(max_length=255)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

```

```
13        USERNAME_FIELD = 'email'

14

15        def __str__(self):
16               return self.email
```

What do we really need to test here? A lot of the tutorials on the web will 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.assertEquals(users_in_db.count(), 1)
12
13          user_from_db = users_in_db[0]
14          self.assertEquals(user_from_db.email, "j@j.com")
15          self.assertEquals(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? Not much, actually. Remember we dont' really need to test Django's functionality.

Technique 2 - Create data on demand

Rather than spend a bunch of time testing stuff we don't really need to test, I 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 email as the USERNAME_FIELD just to make sure nobody changes that on us and also that the user prints out with the email. That's really the only custom functionality we have. But to make things a bit more interesting, let's add a custom function to our model.

Let's add a find_user() function. You could simply say User.objects.get(pk), but if I call that from somewhere else, like main.views.index for example, I'm causing my view code to know about the database, which breaks encapsulation. For small projects this doesn't really matter, but in large projects it can make the code difficult to maintain. *Imagine if you have 40 views and each one uses your User model in slightly different ways, by calling the ORM directly. Then you're stuck in a situation where 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.* Better to encapsulate all that in your User model and as a rule say everyone should only interact with the user model through its API. Then it becomes much easier to maintain. Without further ado, here is the change to our payments *models.py*:

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

Notice the `@classmethod` at the bottom of the code listing. It doesn't do much - just delegates to the Django ORM and returns the User from the database. I made it a `classmethod` because the method itself is stateless. Now we can write some tests for our model.

```
1  class UserModelTest(TestCase):
2
3      @classmethod
4      def setUpClass(cls):
5          cls.test_user = User(email = "j@j.com", name ='test user')
6          cls.test_user.save()
7
8      def test_user_to_string_print_email(self):
9          self.assertEquals(str(self.test_user), "j@j.com")
10
11     def test_get_by_id(self):
12         self.assertEquals(User.get_by_id(1), self.test_user)
```

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**. Just create the data as and when you need it for you test.

> **WARNING**: Be careful with `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 that user object it could cause the other tests to fail

in unexpected ways. This can lead to difficult debugging sessions and tests seemingly failing for no apparent reason. A safer strategy albeit a slower strategy would be to create the user in the `setup` method, which is not a `@classmethod` and is run before each test run. You may also want to look at the tearDown function and use it to delete the user model after each test is run if you decide to go the safer route. As with many 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.

This gives 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 and we don't need to write specific test for the functionality. In effect, testing the database appears sane is a side effect of our test here. Which means, by definition these are not unit tests, they are system tests, but for models the majority of your tests should really be system tests.

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

```python
def index(request):
    uid = request.session.get('user')
    if uid is None:
        return render_to_response('index.html')
    else:
        return render_to_response('user.html', {'user': User.get_by_id(uid)})
```

And update the associated test:

```python
import mock

def  test_index_handles_logged_in_user(self):
    #create a session that appears to have a logged in user
    self.request.session = {"user" : "1"}

    with mock.patch('main.views.User') as user_mock:

        #tell the mock what to do when called
        config = {'get_by_id.return_value':mock.Mock()}
        user_mock.configure_mock(**config)


        #run the test
        resp = index(self.request)

        #ensure we return the state of the session back to normal
        self.request.session = {}

        expected_html = render_to_response('user.html',{'user':
user_mock.get_by_id(1)})
```

```
21        self.assertEquals(resp.content, expected_html.content)
```

This small change actually simplifies our tests as before we had to mock out the ORM i.e. `User.objects`. 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; that's why we have models. And now we don't.

Technique 3 - Using fixtures

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

- It requires you to store you data in a separate file, which means additional places 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.

I hear slow performance often given as another reason a lot of people don't like it. But that can generally be rectified by using an in-memory database. And since I recommend always testing against a database when your testing your models, I'm not as concerned by the speed. Since our example application is using SQLite in-memory we don't have to worry so much about it. But if your application is using some other sql database and you want your tests to run in SQLite so they will run faster, add the following line to your `settings.py` file. Just make sure it comes after your actual database definition.

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

Then anytime you run Django tests it will use SQLite database and you won't notice much of a performance hit at all. Awesome!

> **NOTE**: Do keep in mind the warning early about testing against a different database than your using and 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. Model Mommy is one; another popular one is django-dynamic-fixtures. These are closer to **Technique 2- Create Data on Demand** than they are to using fixtures, 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. Honestly, I haven't heard a really convincing argument for why it's bad, but these models do make it super simple 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.

A couple of good write-ups on Django fixtures libraries can be found below:

- 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 usabilities 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 couple 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 you memory, it looks like this:

```python
from django import forms
from django.core.exceptions import NON_FIELD_ERRORS
from django.utils.translation import ugettext_lazy as _

class PaymentForm(forms.Form):
    def addError(self, message):
        self._errors[NON_FIELD_ERRORS] = self.error_class([message])

class SigninForm(PaymentForm):
    email = forms.EmailField(required = True)
    password = forms.CharField(required = True,
     widget=forms.PasswordInput(render_value=False))
```

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

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 populating data in the form and ensuring it is validated correctly. For our `SingingForm` we are interested in `email` and `password` only. Because most form validation is very similar, let's create a mixin to help us out. If you're unfamiliar with mixins stack overflow can help :

```python
class FormTesterMixin():

    def assertFormError(self, form_cls, expected_error_name, expected_error_msg,
                        data):

        from pprint import pformat
        test_form = form_cls(data=data)
        #if we get an error then the form should not be valid
        self.assertFalse(test_form.is_valid())

        self.assertEquals(test_form.errors[expected_error_name],
                        expected_error_msg,
                        msg= "Expected %s : Actual %s : using data %s" %
                        (test_form.errors[expected_error_name],
                        expected_error_msg, pformat(data)))
```

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. This little function will do all the appropriate validation and give a helpful error message that not only tells you what the failure was, 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 / fix the test failures. Here is an example of how you would use the mixin:

```
class FormTests(unittest.TestCase, FormTesterMixin):

    def test_signin_form_data_validation_for_invalid_data(self):
        invalid_data_list = [
            {'data': { 'email' : 'j@j.com'},
             'error': ('password' , [u'This field is required.'])},
            {'data': {'password' : '1234'},
             'error' : ('email' , [u'This field is required.'])}
        ]

        for invalid_data in invalid_data_list:
            self.assertFormError(SigninForm,
                                 invalid_data['error'][0],
                                 invalid_data['error'][1],
                                 invalid_data["data"])
```

The first thing we do here is use multiple inheritance to inherit from Python standard `unittest.TestCase` and our helpful `FormTesterMixin`. 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_lists` 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 list of field_name, error_message. With this setup we can quickly loop through many different data combination and check that each field in our form is validating correctly.

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 (everybody has got to get a few).

The other thing that Forms do is clean data. If you have customer data cleaning code, you should test that as well. Our `UserForm` does, so let's test that. **Please note: you'll have to inherit from SimpleTestCase to get the assertRaisesMessage function**

```
def test_user_form_passwords_match(self):
    form = UserForm({'name' : 'jj', 'email': 'j@j.com', 'password' : '1234',
                     'ver_password' : '1234', 'last_4_digits' : '3333',
                     'stripe_token': '1'})

    self.assertTrue(form.is_valid())
    #this will throw an error if it doesn't clean correctly
```

```
 8          self.assertIsNotNone(form.clean())

 9

10  def test_user_form_passwords_dont_match_throws_error(self):
11      form = UserForm({'name' : 'jj', 'email': 'j@j.com', 'password' : '234',
12                          'ver_password' : '1234', 'last_4_digits' : '3333',
13                          'stripe_token': '1'})

14

15      self.assertFalse(form.is_valid())

16

17      from django import forms
18      self.assertRaisesMessage(forms.ValidationError,
19                                  "Passwords do not match",
20                                  form.clean)
```

Conclusion

This chapter should have given you a solid overview of unit testing for Django applications. We talked about testing for Django url routing, views, models, and forms. We also talked about the various types of Django test classes, such as SimpleTestCase and TransactionalTestCase. Finally we covered the use of mocks and the mock library to make it easier to isolate your code.

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
System Testing	test integration between componenets
GUI Testing	test end to end from the website
Mocking	to make unit tests more independent
Fixtures	load neccessary data for a test

Testing Technique	Advantages
Unit Testing	test individual pieces of code, fast
System Testing	simple to write, test many things at one
GUI Testing	testing final product soup to nuts, test browser behavior
Mocking	seperate concerns and test one thing at a time
Fixtures	quick and easy data mangement

Testing Technique	Disadvantages
Unit Testing	may require mocking
System 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 comming from

This should provide you with the background you need to start unit testing your applications. While it may seem strange at first, writing your unit tests alongside your code, keep with 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 a quote from an anonymous comment from the internet.

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 concpets covered in the chapter. Working through them will improve your mastery of the subject. Also if you get stuck, answers can be found in Appendix A.

1. Our URL routing testing example only tested one route. Write a test to test the other routes. Where would you put the test to verify the `pages/` route? Do you need to / want 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 the `contact/models`. What do you really need to test? Do you need to use the database backend?

4. The QA teams I have worked with have been particularly keen on 'boundary checking'. Write some unit tests for the CardForm that ensure boundary checking is working correctly.

Chapter 3

Test Driven Development

Test Driven Development (TDD) is a software development practice that puts a serious emphasis on writing automated test 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. No seriously, 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 those 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:
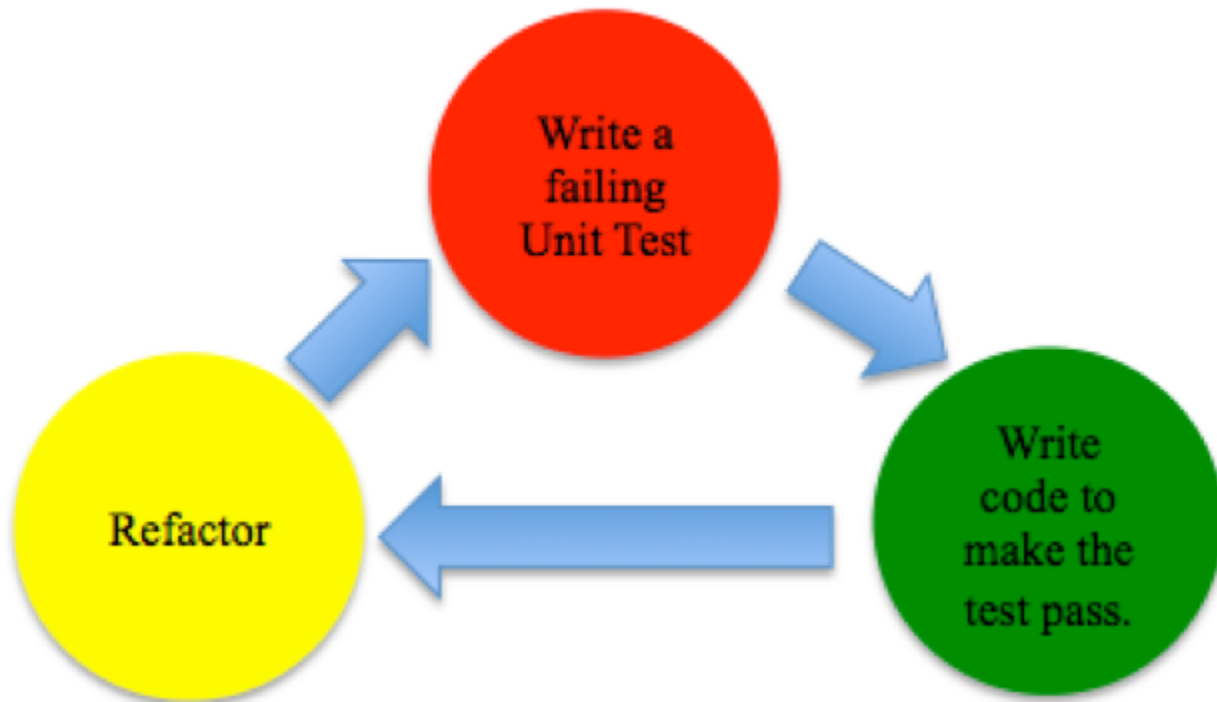
Figure 3.1: TDD workflow

RED LIGHT - The above workflow starts with writing a test **before** you write any code. The test will of course fail because the code it is supposed to verify doesn't exist yet. This may seem like a waste of time, writing a test before you even have code, 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:

- 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.

GREEN LIGHT - Once you have a basic test written, write **just enough** code to make it work. 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 oftentimes you won't. So don't write anything unless you really need it.

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 to 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.

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 has been 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!** But for most people reading this, 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 you next project to get started with TDD. You can start applying TDD to existing code, and in this section we are going to show you how to do that.

Before we jump into the code let's first talk briefly about where to start with TDD. There are 3 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. To refresh your memory the application we have been working on is a simple Django application that provides a registration function (that is integrated with stripe to take payments). As well as the basic login / logout and homepage 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 describing below. 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. Just type the following to checkout the appropriate tag:

```
$ git checkout tags/ch3
```

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

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

```
def register(request):
    user = None
    if request.method == 'POST':
        form = UserForm(request.POST)
        if form.is_valid():

            #update based on your billing method (subscription vs one time)
            customer = stripe.Customer.create(
```

```python
                email = form.cleaned_data['email'],
                description = form.cleaned_data['name'],
                card = form.cleaned_data['stripe_token'],
                plan="gold",
            )
            # customer = stripe.Charge.create(
            #    description = form.cleaned_data['email'],
            #    card = form.cleaned_data['stripe_token'],
            #    amount="5000",
            #    currency="usd"
            # )

            user = User(
                name = form.cleaned_data['name'],
                email = form.cleaned_data['email'],
                last_4_digits = form.cleaned_data['last_4_digits'],
                stripe_id = customer.id,
            )

            #ensure encrypted password
            user.set_password(form.cleaned_data['password'])

            try:
                user.save()
            except IntegrityError:
                form.addError(user.email + ' is already a member')
            else:
                request.session['user'] = user.pk
                return HttpResponseRedirect('/')

    else:
      form = UserForm()

    return render_to_response(
        'register.html',
        {
          'form': form,
          'months': range(1, 12),
          'publishable': settings.STRIPE_PUBLISHABLE,
          'soon': soon(),
          'user': user,
          'years': range(2011, 2036),
        },
        context_instance=RequestContext(request)
    )
```

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.py`. In the Chapter 2 exercises the first question is about creating tests for the various routes. In Appendix A we propose the answer of using a `ViewTesterMixin`. If you were to implement that solution you would probably come up with a call like `RegisterPageTests` that tests that the registration page returns the appropriate HTML / template data. The code for those tests are shown below.

```
import django_ecommerce.settings as settings
from .views import soon
class RegisterPageTests(TestCase, ViewTesterMixin):

    @classmethod
    def setUpClass(cls):
        html = render_to_response('register.html',
        {
          'form': UserForm(),
          'months': range(1, 12),
          'publishable': settings.STRIPE_PUBLISHABLE,
          'soon': soon(),
          'user': None,
          'years': range(2011, 2036),
        })
        ViewTesterMixin.setupViewTester('/register',
                                        register,
                                        html.content,
                                        )
```

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

```
git checkout tags/Ch3
```

That will get you working with the same set of code.

Now 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

49

4. `request.method == POST` and `user.save()` throws an `IntegretyError`:

and

Expected Results

1. return basic register.html

2. return register.html (possibly with an error msg)

3. set `session['user'] == user.pk` and return `HTTPResponseRedirect('/')`

4. return register.html with "user is already a member" error.

Our existing tests covers the first case, by using the `ViewTesterMixin`. Let's write tests for the rest. Case 2 would look like this:

```
from django.test import RequestFactory


def setUp(self):
    request_factory = RequestFactory()
    self.request = request_factory.get(self.url)


def test_invalid_form_returns_registration_page(self):

    with mock.patch('payments.forms.UserForm.is_valid') as user_mock:

        user_mock.return_value = False

        self.request.method = 'POST'
        self.request.POST = None
        resp = register(self.request)
        self.assertEquals(resp.content, self.expected_html)

        #make sure that we did indeed call our is_valid function
        self.assertEquals(user_mock.call_count, 1)
```

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

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.

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

The problem with this is it will actually try to go out to the Stripe server, which will reject the call
because the stripe_token is invalid. So let's mock out the Stripe server to get around that.

```
1   def test_registering_new_user_returns_succesfully(self):
2
3       self.request.session = {}
4       self.request.method='POST'
5       self.request.POST = {'email' : 'python@rocks.com',
6           'name' : 'pyRock',
7           'stripe_token' : '...',
8           'last_4_digits' : '4242',
9           'password' : 'bad_password',
10          'ver_password' : 'bad_password',
11        }
12
13      with mock.patch('stripe.Customer') as stripe_mock:
14
15          config = {'create.return_value': mock.Mock()}
16          stripe_mock.configure_mock(**config)
17
18          resp = register(self.request)
19          self.assertEquals(resp.content, "")
20          self.assertEquals(resp.status_code, 302)
21          self.assertEquals(self.request.session['user'], 1)
22
23          #verify the user was actually stored in the database.
24          #if the user is not there this will throw an error
25          db_user = User.objects.get(email='python@rocks.com')
```

So now we don't have to go out to 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:

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

> in this example we create a python dictionary called config and then pass it to the `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 get around it with argument unpacking. More details about this technique can be found here

Coming back to testing, you may be noticing that these tests are getting large and testing several different things. To me that's a code smell saying that we have a function that is doing too many things. Really we should be able to test most functions in three or four lines of code, plus maybe a few more for mocks when we have to use them, so as a rule of thumb if my test methods start getting longer than about 10 lines of code then I get concerned and I look for places to re-factor.

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

```python
def test_registering_user_twice_cause_error_msg(self):

    #create a user with same email so we get an integrity error
    user = User(name='pyRock', email='python@rocks.com')
    user.save()

    #now create the request used to test the view
    self.request.session = {}
    self.request.method='POST'
    self.request.POST = {'email' : 'python@rocks.com',
                         'name' : 'pyRock',
                         'stripe_token' : '...',
                         'last_4_digits' : '4242',
                         'password' : 'bad_password',
                         'ver_password' : 'bad_password',
                        }

    #create our expected form
    expected_form = UserForm(self.request.POST)
    expected_form.is_valid()
    expected_form.addError('python@rocks.com is already a member')

    #create the expected html
```

```
24        html = render_to_response('register.html',
25        {
26          'form': expected_form,
27          'months': range(1, 12),
28          'publishable': settings.STRIPE_PUBLISHABLE,
29          'soon': soon(),
30          'user': None,
31          'years': range(2011, 2036),
32        })
33
34        #mock out stripe so we don't hit their server
35        with mock.patch('stripe.Customer') as stripe_mock:
36
37            config = {'create.return_value': mock.Mock()}
38            stripe_mock.configure_mock(**config)
39
40            #run the test
41            resp = register(self.request)
42
43            #verify that we did things correctly
44            self.assertEquals(resp.status_code, 200)
45            self.assertEquals(self.request.session, {})
46
47            #assert there is only one record in the database.
48            users = User.objects.filter(email="python@rocks.com")
49            self.assertEquals(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 to me that maybe 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.assertEquals(resp.content, html.content)
```

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

Expected	Actual
`<a href="/sign_in">Login</a>`	`<a href="/sign_out">Logout</a>`

So it looks like even though you didn't register correctly the system thinks you are 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:

Figure 3.2: Registration Error

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 we find it than 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 IntegretyError: handler.

```
1  try:
2      user.save()
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('/')
```

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 re-factoring 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.*

I would start by pulling out the section of code that creates the new user, and putting it in the User model class. This way we leave all the User processing stuff to the User class. So to be clear, we will take this section:

```
user = User(
    name = form.cleaned_data['name'],
    email = form.cleaned_data['email'],
    last_4_digits = form.cleaned_data['last_4_digits'],
    stripe_id = customer.id,
)

#ensure encrypted password
user.set_password(form.cleaned_data['password'])

try:
    user.save()
```

And put it in the User model class so it looks like this:

```
@classmethod
def create(cls, name, email, password, last_4_digits, stripe_id):
    new_user = cls(name=name, email=email,
        last_4_digits=last_4_digits, stripe_id=stripe_id)
    new_user.set_password(password)

    new_user.save()
    return new_user
```

Using classmethods as alternative to constructors is a idiomatic way to do things in Python, so we are just following acceptable patterns here. The call to this function from the registration view function is below:

```
cd = form.cleaned_data
try:
    user = User.create(cd['name'], cd['email'], cd['password'],
        cd['last_4_digits'], customer.id)
except IntegrityError:
    form.addError(cd['email'] + ' is already a member')
else:
    request.session['user'] = user.pk
    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 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!

Taking a defensive approach, let's update some of our tests before moving on with the re-factoring. The first thing is to 'move' the testing of the user creation to the user model and test the create class method. In the `UserModelTest` class we can add the following two methods:

```python
def test_create_user_function_stores_in_database(self):
    user = User.create("test", "test@t.com", "tt", "1234", "22")
    self.assertEquals(User.objects.get(email="test@t.com"), user)

def test_create_user_allready_exists_throws_IntegrityError(self):
    from django.db import IntegrityError
    self.assertRaises(IntegrityError, User.create, "test user",
                        "j@j.com","jj","1234",89)
```

With these tests we know that the user creation stuff 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 have already tested that the `User.create` method functions as intended. Just to show how it works, let's go ahead and mock out the `test_registering_new_user_returns_succesfully()` function.

```python
@mock.patch('stripe.Customer.create')
@mock.patch.object(User,'create')
def test_registering_new_user_returns_succesfully(self, create_mock,
    stripe_mock):

    self.request.session = {}
    self.request.method='POST'
    self.request.POST = {'email' : 'python@rocks.com',
                         'name' : 'pyRock',
                         'stripe_token' : '...',
                         'last_4_digits' : '4242',
                         'password' : 'bad_password',
                         'ver_password' : 'bad_password',
                            }

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

    resp = register(self.request)
```

```
21        self.assertEquals(resp.content, "")
22        self.assertEquals(resp.status_code, 302)
23        self.assertEquals(self.request.session['user'], new_user.pk)
24        #verify the user was actually stored in the database.
25
          create_mock.assert_called_with('pyRock','python@rocks.com','bad_password','4242',new_cust.i
```

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. We use them in these two assertions:

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

The first assertion just verifies that we are setting the session value to the id of the object returned from our `create_mock`. And the second assertion is an assertion that is built into the mock library, allowing you 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.

While mocks can be handy at cutting out part of the system or isolating a test, in this example they are probably more trouble then 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 server... NOT GOOD. But in terms of the database, I wouldn'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 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. As for general rules I generally think about it like this:

- When to Mock
- When a function is horribly slow and you don't want to run it each time.
- Similarly when the function needs to call something not on your development machine (think web APIs)
- When the function requires setup / state that is more difficult to manage than setting up the mock (think unique data generation)
- When the function is somebody else's code and you really don't trust it / don't want to be responsible for it (but see When not to Mock below)

- Time-dependent functionality is good to mock as often its difficult to recreate the dependency in a unit test.
- When not to Mock
- This is my default choice: basically, 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 unittest is paramount to anything else. While it is true mocking will generally speed up the execution time of your unittest suite, unless its a significant speed increase (like is the case for `stripe.Customer.create`) my advice is: *Mocking is a great way to kill brain cycles; 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. Then 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`. I'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:

- **RED LIGHT** - write a test that fails
- **GREEN LIGHT** - write just enough code to make it pass
- **YELLOW LIGHT** - re-factor your code / tests until you have something beautiful
- **REPEAT**

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

We also touched on applying TDD and unittesting 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, but first here are a couple of exercises to cement all this TDD stuff in your brain.

Exercises

1. If you really want to get your head around mocks, have a go at mocking out the `test_registering_user_twice` test. Start by mocking out the `User.create` function and have it throw the appropriate errors. HINT the documentation is a great resource and the best place to start. In particular have a search for `side_effect`. For extra credit mock out the UserForm as well.

2. As alluded to in the conclusion, pulling out the customer creation logic from `register()` into a separate CustomerManager class may be a good idea. Go ahead and re-read the first paragraph of the conclusion and do that. Don't forget to update the tests accordingly.

Chapter 4

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 1.6.

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 Real Python Part 2 book. 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 you 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 it make 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 1.6 or a 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 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 aren't getting access to 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 with 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 hotfixes.
- Feature Branches
- latest / development branches
- Easily handling UAT and last minute release type of work, before a branch of code makes it into production.

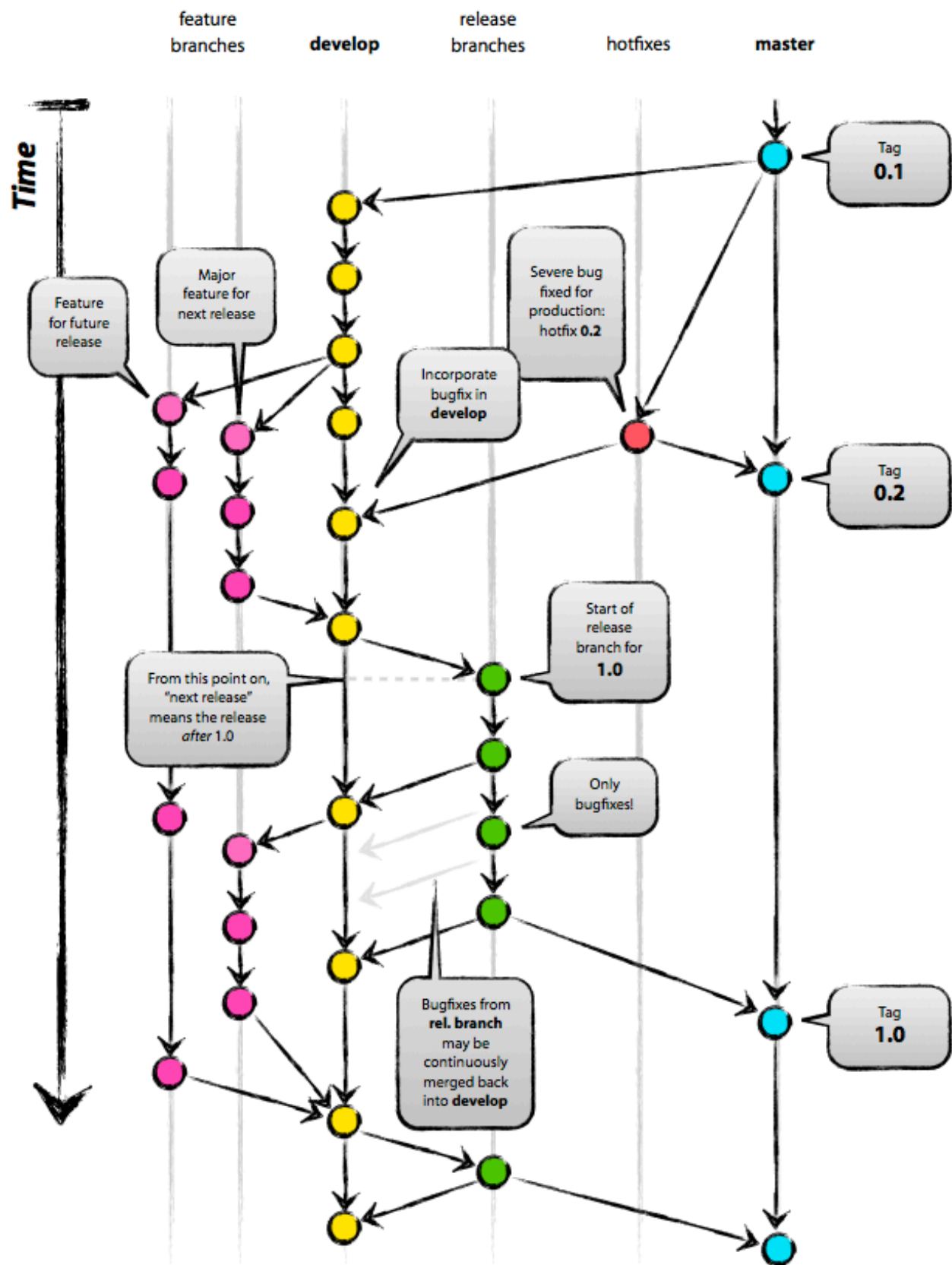A quick snapshot of what the branching model looks like can be seen below:

Figure 4.1: git-flow branching model

64

For large software projects, when working with a team, I highly recommend this model. It has served me quite well in the past. Since Vincent does such a good job of describing it on his website, I'm not going to go into detail about it here, but I will say if you are working on a large project this is a good branching model to use. For smaller projects or with less team members you may want to look at the model below. Also I have reports from a number of highly skilled developers that they 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 which is a collection of scripts that make using this branching model pretty straight-forward, and they save a lot on the typing. 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 I like to call **The Github Pull-Request Model**. I 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 seeming simplicity, this model works pretty effectively. Let's go though 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 Branch from Master

For this model to work effectively the Master branch must always be deployable. That means keep it clean at all times and 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
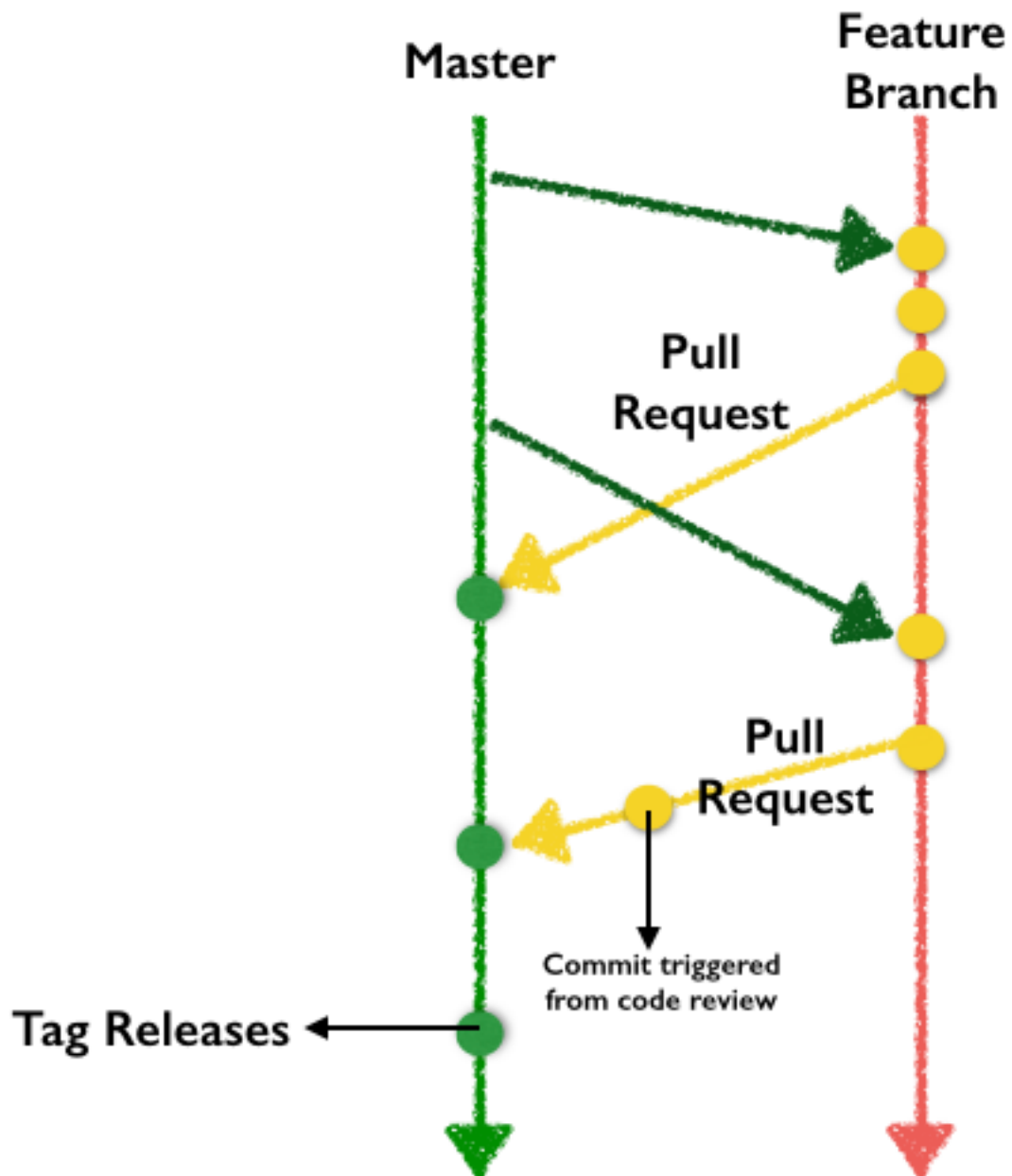
Figure 4.2: Github Pull-Request Model

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 it to go through a code review before being merged back into master. Create the branch by simply typing:

```
$ 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 your 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 commits / 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:

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

Note the -i 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. Here is an example:

```
pick 7850c5d tag: Ch3_before_exercises
pick 6999b75 Ch3_ex1
pick 6598edc Ch3_ex1_extra_credit
pick 8779d40 Ch3_ex2

# Rebase 7850c5d..8779d40 onto dd535a5
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  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 bottom.
17   #
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 and 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 and I find 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, apply 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.

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:

```
$ 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:

```
$ git checkout feature_branch_name
$ 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:

```
$ 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 If you're using the GitHub web GUI and you use the Merge automatically button Github actually does:

```
$ git checkout master
$ git pull origin master
$ 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:

```
*   5340d4b Merge branch 'new_feature_branch'
|\
| * 4273443 file2
| * 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 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:

```
1   $ 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 the 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 1.6.

Exercise

The exercise for this chapter is pretty fun.

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/

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 5

Upgrade, Upgrade, and Upgrade some more

Django 1.6

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 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.

```
$ git checkout -b django1.6
```

Then we also need to create a separate virtualenv and upgrade that virtualenv to Django 1.6:

```
$ mkvirtualenv django1.6
$ pip install -r requirements.txt
$ pip install django -U
```

We have created a new virtualenv called django1.6, installed all of our dependencies in that environment, and upgraded to Django1.6. Now to see if everything works, we want to run all of our tests. There is a nice option you can use when running your tests, `-Wall`, that will show all warnings in addition to errors; its a good idea to turn that on now as well so we can spot any other potential gotchas. Cross your fingers and type:

```
$ python -Wall manage.py test
```

and the result....

```
=========================================================================
ERROR: test_signin_form_data_validation_for_invalid_data
    (payments.tests.FormTests)
-------------------------------------------------------------------------
Traceback (most recent call last):
  File
    "/Users/j1z0/Code/RealPython/mvp_for_Adv_Python_Web_Book/django_ecommerce/payments/tests.py
    line 65, in test_signin_form_data_validation_for_invalid_data
     invalid_data["data"])
  File
    "/Users/j1z0/.virtualenvs/django_1.6/lib/python2.7/site-packages/django/test/testcases.py",
    line 398, in assertFormError
     contexts = to_list(response.context)
AttributeError: type object 'SigninForm' has no attribute 'context'


-------------------------------------------------------------------------
```

#FAIL - Would you look at that. Django 1.6 decided to take the name of our function `assertFormError`. Guess the Django folks also thought we needed something like that. OK, let's just change the name of our function to something like `shouldHaveFormError()` so it won't get confused with internal Django stuff.

Rerun the tests and now they are all at least passing. Nice. But we still have a warning to deal with:

```
1 PendingDeprecationWarning: 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. `PendingDeprecationWarning` refers to functionality that you will have to change for the next version, e.g. 1.7. 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`. It will now look like:

```
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 rerun our unit tests - no errors and no warnings. Let's just add one more test to make sure that we are displaying the expected fields for our ContactForm like so:

```
1 from .forms import ContactView
2
3 class ContactViewTests(SimpleTestCase):
4
5    def test_displayed_fields(self):
6        expected_fields = ['name','email','topic','message']
7        self.assertEquals(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 teammates 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.

New in Django 1.6 is the `check` management command. This will check our configuration and make sure it is compatible with Django 1.6, so let's go ahead and run that as well, just to make sure that our *settings.py* is in order. By the way the `check` command won't show any output if there are no issues, so if you see a blank screen after running the command, that's a good thing. :)

```
1 $ ./manage.py check
```

```
 2
 3  UserWarning: Django 1.6 introduced a new default test runner
       ('django.test.runner.DiscoverRunner') 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-any-test-module
       for more information.
 4    warnings.warn(message)
```

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 ok here. 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.

In Django 1.5 you were "strongly encouraged" to put you test code in the same folder as your production code in a module called *test.py*. While this is ok 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 that verifies every route in the project points to the correct view. In Django 1.6 you can do 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 a src and a test top level directory. The test directory mirrors the source directory and only contains tests; that way you can easily find what tests apply to what part of the application. Graphically that structure would look like this:

```
 1  |-- django_ecommerce
 2    |-- contact
 3    |-- django_ecommerce
 4    |-- main
 5    |-- payments
 6  |-- tests
 7    |-- contact
 8      |-- userModelTests.py
 9      |-- contactViewTests.py
10    |-- django_ecommerce
11    |-- main
12      |-- mainPageTests.py
13    |-- 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 having separate files for helper test functions; for example, the `ViewTesterMixin` from Chapter 3 could be refactored out into a `tests\testutils.py` so it's easily shared amongst the various tests classes)
- Simple to see the linkage between the test cases and the code it's testing

- Make sure that test code isn't accidentally copied over to a production environment

    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 make the changes. You can find them here:

**********Put in a reference to the github commit number a9b81a1710edfd5774f******

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

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

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

Once everything is working, remember that we need to issue a pull request so the code can be reviewed and eventually merged back into our main line. To do that, we issue the following commands from the terminal:

First to show we have made the commits:

```
$ git log --oneline
a9b81a1 moved all tests out to a separate tests directory
f5dab51 updated ContactView to explicitly list fields to be displayed as per Dja
7cda748 wouldn't you know it, Django 1.6 took my assertFormError name... changed
58f1503 yes we are upgrading to Django 1.6
```

The above shows the four commits that we have made as part of our upgrade process. Now all we need to do is issue a pull request. If you have installed hub from the last chapter, you can just type:

```
$ git pull-request "upgraded to django 1.6 fixed all errors" -b
    repo_owner:master -h repo_owner:django_1.6
```

(Keep in mind that hub must also be aliased to git using a line line `alias git=hub` in your .bash_profile for the previous line to work).

This will issue the merge request which can be committed on and reviewed from the GitHub UI. After the merge request has been reviewed / approved then it can be merged from the command line with:

```
$ git checkout master
$ git pull origin master
$ git merge --no-ff django_1.6
```

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.

That's it for the upgrade. 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.

Well that's almost true but not entirely. Of course you need to make sure there are no errors, but it's also a good idea to read through 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 1.6 officially supports Python 3. In the docs they highly recommend 3.2.x or 3.3.x. So let's go ahead and upgrade straight to 3.3.4, which is the latest version as of this writing. Although 3.4 will probably be out by the time you read this, the instructions below should remain the same.

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 I feel I wouldn't be doing my job if I didn't give you both sides of the story.

First off, the negative side of things:

Armin Ronacher has one of the most popual "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 was in Python 2.x. To be fair, he isn't wrong and some of the issues still exist, but a good amount of the issues he is talking about are now fixed in 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. So my thought is just to get everybody to 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 do 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.

Here's a short list: Unicode everywhere, virtualenv builtin, 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 more...

While I don't intend to 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 there, and throughout the rest of this book I'll introduce several of them to you. So now you know both sides of the story. Let's look at how to do the upgrade.

Final Checks before the upgrade

The first thing to do is to make sure that all of the third party Django applications and tools that you are using support Python 3. These days there are a large number of apps that do. But you can find the list on PyPI Python 3 packages list. This lists all packages hosted by PyPI that support Python 3.

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 (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 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, we'll examine how you might support both 2 and 3 and the same time.

1. *Drop support for python 2.5* - while it is possible to support from python 2.5 -> 3.3 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 I say just drop the support and make your life easier. :)

2. *Set up a Python 3 virtual env* – 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 and 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.

5. A couple of resources that may help.

   - 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 Python 3 and Python 2 on the same code base, you can have a look at the pull request I did for Flask-FlatPages.

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

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

   - *Create a virtualenv for Python 3.* We will have to both create the virtualenv as well as pip install all our dependencies in the upgraded environment.

```
$ cd <project_dir>
$ mkvirtualenv -p /usr/local/bin/python3 py3
$ pip install -r requirements.txt
```

These commands will create a new virtualenv with Python 3, switch you to that environment, and install all the project dependencies.

   - *run 2to3* - Now let's see what we need to change to support Python 3. If we run 2to3 against our Django app like this:

```
$ 2to3 django_ecommerce
```

It will return a diff listing of what needs to change. The listing should look like:

```
1   --- django_ecommerce/payments/forms.py    (original)
2   +++ django_ecommerce/payments/forms.py   (refactored)
3   @@ -17,8 +17,8 @@
4    class UserForm(CardForm):
5        name = forms.CharField(required = True)
6        email = forms.EmailField(required = True)
7   -    password = forms.CharField(required = True, label=(u'Password'),
        widget=forms.PasswordInput(render_value=False))
8   -    ver_password = forms.CharField(required = True, label=(u' Verify
        Password'), widget=forms.PasswordInput(render_value=False))
9   +    password = forms.CharField(required = True, label=('Password'),
        widget=forms.PasswordInput(render_value=False))
10  +    ver_password = forms.CharField(required = True, label=(' Verify Password'),
        widget=forms.PasswordInput(render_value=False))
11
12       def clean(self):
13           cleaned_data = self.cleaned_data
14  --- django_ecommerce/payments/views.py  (original)
15  +++ django_ecommerce/payments/views.py  (refactored)
16  @@ -33,7 +33,7 @@
17       else:
18         form = SigninForm()
19
20  -    print form.non_field_errors()
21  +    print(form.non_field_errors())
22
23       return render_to_response(
24           'sign_in.html',
25  @@ -87,11 +87,11 @@
26           'register.html',
27           {
28             'form': form,
29  -          'months': range(1, 12),
30  +          'months': list(range(1, 12)),
31             'publishable': settings.STRIPE_PUBLISHABLE,
32             'soon': soon(),
33             'user': user,
34  -          'years': range(2011, 2036),
35  +          'years': list(range(2011, 2036)),
36           },
37           context_instance=RequestContext(request)
38         )
39  @@ -127,8 +127,8 @@
40             'form': form,
41             'publishable': settings.STRIPE_PUBLISHABLE,
42             'soon': soon(),
43  -          'months': range(1, 12),
```

```
44  -            'years': range(2011, 2036)
45  +            'months': list(range(1, 12)),
46  +            'years': list(range(2011, 2036))
47          },
48          context_instance=RequestContext(request)
49      )
```

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

```
1  -     password = forms.CharField(required = True, label=(u'Password'),
       widget=forms.PasswordInput(render_value=False))
2  -     ver_password = forms.CharField(required = True, label=(u' Verify
       Password'), widget=forms.PasswordInput(render_value=False))
3  +     password = forms.CharField(required = True, label=('Password'),
       widget=forms.PasswordInput(render_value=False))
4  +     ver_password = forms.CharField(required = True, label=(' Verify Password'),
       widget=forms.PasswordInput(render_value=False))
```

In the first change it found above we can see that all it has done is put the `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 explicitly required; 2to3 puts it in so you can support both 2 and 3 at the same time, which we are not doing. So if you put in the `u` or leave it out, it will still be ok.

On to the next issue:

```
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 is done just to standardize things. The next few issues are all the same issue.

```
1  -            'months': range(1, 12),
2  +            'months': list(range(1, 12)),
```

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 is effectively the same as what xrange was in Python 2. (Meanwhile xrange was removed in Python 3.) The switch to returning generators 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 covert 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. However if we try to run our tests now we will get a bunch of errors because the tests also need to be converted to Python 3. Running 2to3 on our tests directory will give us the following errors.

```
1    --- tests/payments/testForms.py (original)
2    +++ tests/payments/testForms.py (refactored)
3    @@ -23,9 +23,9 @@
4            def test_signin_form_data_validation_for_invalid_data(self):
5                invalid_data_list = [
6                    {'data': { 'email' : 'j@j.com'},
7    -                'error': ('password' , [u'This field is required.'])},
8    +                'error': ('password' , ['This field is required.'])},
9                    {'data': {'password' : '1234'},
10   -                'error' : ('email' , [u'This field is required.'])}
11   +                'error' : ('email' , ['This field is required.'])}
12               ]
13
14               for invalid_data in invalid_data_list:
15   @@ -37,9 +37,9 @@
16           def test_card_form_data_validation_for_invalid_data(self):
17               invalid_data_list = [
18                   {'data': {'last_4_digits' : '123'},
19   -                'error' : ('last_4_digits', [u'Ensure this value has at least 4
         characters (it has 3).'])},
20   +                'error' : ('last_4_digits', ['Ensure this value has at least 4
         characters (it has 3).'])},
21                   {'data': {'last_4_digits' : '12345'},
22   -                'error' : ('last_4_digits', [u'Ensure this value has at most 4
         characters (it has 5).'])}
23   +                'error' : ('last_4_digits', ['Ensure this value has at most 4
         characters (it has 5).'])}
24               ]
25
26               for invalid_data in invalid_data_list:
27   --- tests/payments/testUserModel.py (original)
28   +++ tests/payments/testUserModel.py (refactored)
29   @@ -53,9 +53,9 @@
30           def test_signin_form_data_validation_for_invalid_data(self):
31               invalid_data_list = [
32                   {'data': { 'email' : 'j@j.com'},
33   -                'error': ('password' , [u'This field is required.'])},
34   +                'error': ('password' , ['This field is required.'])},
35                   {'data': {'password' : '1234'},
36   -                'error' : ('email' , [u'This field is required.'])}
37   +                'error' : ('email' , ['This field is required.'])}
38               ]
39
40               for invalid_data in invalid_data_list:
41   @@ -67,9 +67,9 @@
42           def test_card_form_data_validation_for_invalid_data(self):
43               invalid_data_list = [
```

```
44              {'data': {'last_4_digits' : '123'},
45 -                 'error' : ('last_4_digits', [u'Ensure this value has at least 4
      characters (it has 3).'])},
46 +                 'error' : ('last_4_digits', ['Ensure this value has at least 4
      characters (it has 3).'])},
47              {'data': {'last_4_digits' : '12345'},
48 -                 'error' : ('last_4_digits', [u'Ensure this value has at most 4
      characters (it has 5).'])}
49 +                 'error' : ('last_4_digits', ['Ensure this value has at most 4
      characters (it has 5).'])}
50              ]
51
52          for invalid_data in invalid_data_list:
53 --- tests/payments/testViews.py (original)
54 +++ tests/payments/testViews.py (refactored)
55 @@ -75,11 +75,11 @@
56          html = render_to_response('register.html',
57          {
58            'form': UserForm(),
59 -          'months': range(1, 12),
60 +          'months': list(range(1, 12)),
61            'publishable': settings.STRIPE_PUBLISHABLE,
62            'soon': soon(),
63            'user': None,
64 -          'years': range(2011, 2036),
65 +          'years': list(range(2011, 2036)),
66          })
67          ViewTesterMixin.setupViewTester('/register',
68                                          register,
69 @@ -178,11 +178,11 @@
70          html = render_to_response('register.html',
71          {
72            'form': self.get_MockUserForm(),
73 -          'months': range(1, 12),
74 +          'months': list(range(1, 12)),
75            'publishable': settings.STRIPE_PUBLISHABLE,
76            'soon': soon(),
77            'user': None,
78 -          'years': range(2011, 2036),
79 +          'years': list(range(2011, 2036)),
80          })
```

Looking through the output we can see the same basic errors that we had in our main code line. One is the suggestion to include u before each string literal, the second is converting the generator returned by a range to a list, and the third is calling 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:

```
$ cd django_ecommerce
$ ./manage.py tests ../tests
```

Whoa... Errors all over the place. :(. 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 off, you should have gotten a bunch of DepreciationWarnings that say something about getting rid of assertEquals in favor of assertEqual. Kind of annoying but it makes sense to keep the API clean, and it's nothing a search and replace can't find. So let's do a search and replace in our tests directory and drop the 's' off the end of all the assetEquals statements.

Now if we run the tests again we still get some errors, but at least all those DepreciationWarnings are gone. Here is the first remaining error:

```
=======================================================================
FAIL: test_contactform_str_returns_email
    (tests.contact.testContactModels.UserModelTest)
-----------------------------------------------------------------------
Traceback (most recent call last):
  File
    "/Users/j1z0/Code/RealPython/mvp_for_Adv_Python_Web_Book/tests/contact/testContactModels.py
    line 19, in test_contactform_str_returns_email
     self.assertEqual("first@first.com", str(self.firstUser))
AssertionError: 'first@first.com' != 'ContactForm object'
- first@first.com
+ 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:

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

In Python 3 the unicode() built-in function has gone away and str() always returns a unicode string. Therefore the function **unicode** is just ignored. Change the name of the function as shown below and that test will pass.

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

Keep in mind, in Python 3 for you models you only need the **str** function. You can forget about the **unicode**, as it's not needed. This and various other Python 3-related errata for Django can be found here.

Three more errors to go, but lucky for us they all have the same solution. First the error:

```
1  ========================================================================
2  FAIL: test_returns_correct_html (tests.payments.testViews.EditPageTests)
3  ------------------------------------------------------------------------
4  Traceback (most recent call last):
5    File
        "/Users/j1z0/Code/RealPython/mvp_for_Adv_Python_Web_Book/tests/payments/testViews.py",
        line 39, in test_returns_correct_html
6      self.assertEqual(resp.content, self.expected_html)
7  AssertionError: b'' != ''
```

In python2 a bytestring (`b'somestring'`) is effectively the same as a unicode string (`u'somestring'` as long as 'somestring' 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'somestring' != u'somestring'`.

BUT, and the big but here, is that according to PEP 3333, input and output streams are always `byte` objects - and 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 i.e. 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:

test/payments/testViews.py - SignOutPageTests

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

All we need to do is change the third argument to `setupViewTester` from `''` to `b''`, because remember: response.context is now a bytestring, so our expected_html must also be a bytestring. We make the same change to `EditPageTests` and then the two errors go away.

There is one more error, but it's the same type of bytestring error, with the same type of fix. This fix is shown below:

```
1  __test/payments/testViews.py -
      RegistrationPageTests.test_registering_new_user_returns_succesfully__
2   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. 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 1.6 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.

```
$ ls -al django_ecommerce/django_ecommerce/__pycache__
total 24
drwxr-xr-x   5 j1z0   staff    170 Dec 20 17:52 .
drwxr-xr-x  11 j1z0   staff    374 Dec 29 01:24 ..
-rw-r--r--   1 j1z0   staff    196 Dec 20 17:52 __init__.cpython-33.pyc
-rw-r--r--   1 j1z0   staff   3932 Dec 20 17:52 settings.cpython-33.pyc
-rw-r--r--   1 j1z0   staff    997 Dec 20 17:52 urls.cpython-33.pyc
```

There are a few things to notice 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 unclutters your code directory.

2. Further notice the format: `__init__.cpython-33.pyc`. Pyc files are now in the format .-.pyc. This allow for storing multiple pyc files; if you're testing against 3.3 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 and multiple vms for projects that need to do that. For projects that don't... at least it gets the pyc files out of your code directories.

As I said before, there are a ton of new features in Python 3 and we will see several of them as we progress throughout the book. 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 sqllite, which is OK for testing and descriptive purposes, but you would probably not want to use sqllite in production. (At least not for a high traffic site. Have 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:

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

This is a built-in Django command that will export all the data using the JSON format. The nice thing about this is that JSON is database-independent, so we can easily pass this data to other databases (in our case to Postgres). The downside of this approach is that you won't get things like primary and foreign keys. 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 PostgreSQL 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. I'm not going to go into the details of installation. Instead I'll just point you to the official installation page and you can go from there.

PostgreSQL Installation Page

For Debian-based systems the following procedure should work:

1. Install postgres `shell`   `$ sudo apt-get install postgresql`

2. Verify installation is correct: `shell`   `$psql --version`

   You should get something like the following:

   ```
   psql (PostgreSQL) 9.1.11
   contains support for command-line editing
   ```

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.

   ```
   $ sudo su postgres
   ```

4. Now postgres creates a Django user in the database.

```
1  $ createuser -P djangousr
```

Enter the password twice and remember it, because 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 #**

```
1  $ psql
2  #once in the shell type the following to create the database
3  $ CREATE DATABASE django_db OWNER djangousr ENCODING 'UTF8';
4  #then to quit the shell type
5  $ \q
```

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

```
1  local    django_db        djangousr     md5
```

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

7. Now 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 djangousr
```

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.

Step 2 Once postgres is installed, you will want to install the Python bindings. Ensure that you are in the virtualenv that we created earlier, and then install the postgresql binding 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.

Try this:

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

then re-run 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
```

> **NOTE**: Note for OSX Hombrew Users:
>
> If you have install python or postgress through homebrew and the `pip install` fails ensure that you're using a version of Python installed by homebrew and that you install 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.

For 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 (be sure to get the Python 3 version!) is here: Windows prebuilt installer

To be clear installing postgres globally is not an issue as far as this book is concerned. It can however become an issue if you are working on multiple projects each with 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, this 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. Go ahead and add a line to that file that simply says `psycopg2`. Thus the *requirments.txt* file should now look something like this:

```
1  Django
2  mock
3  requests
4  stripe
5  psycopg2
```

That's all the dependencies we need for now. Also note that in the requirements.txt file it is perfectly valid to put a version number in such as *Django=1.6* . If you don't put a version number in pip will search for the latest version and install that one. Since we have built up a good set of test cases and because

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:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'django_db',
        'USER': 'djangousr',
        'PASSWORD': 'djangousr', #or whatever you password was
        'HOST': 'localhost', #or an ip addr
        'PORT': '5432',
    }
}
```

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

```
$ grep postgres /etc/services
```

Once that is done just sync up the db:

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

The final thing is to import the `backup.json` file we created earlier. `shell $ ./manage.py loaddata backup.json`

Of course this isn't essential if you're following this course strictly as you won't yet have much (if any) data to backup / 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:

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

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. Well, 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, my guess is 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.

Exercise

There are some really good videos out there on upgrading to Python 3 and Django 1.6. So for this chapter's exercise, we will point you to a few videos to watch to give you 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 6

Graceful Degradation and Database Transactions with Django 1.6

Back in my day, we used to have to walk to school barefoot, in the snow and uphill (both ways). :) But seriously, back when I was getting started in Computer Science (a long long time ago...) Graceful Degradation was a hot topic. Today you don't hear about it too much, at least not in the startup world, because everybody is rushing to get their MVP out the door and get sign-ups. "We'll fix it later, right?" With Django 1.6 and the drastic simplification it has brought to transactions, we can actually apply a little bit of Graceful Degradation to our sign-in process without having to spend too much time on it.

We should start with a quick definition of Graceful Degradation:

Graceful Degradation - the property that enables a system to continue operating properly in the event of the failure of some of its components.

– [Wikipedia](#)

If something breaks, we just keep on ticking. A common example of this in web development is IE support. :) Most cool stuff doesn't work in IE, right? (Well, it is getting better.) To compensate, we write frameworks with a bunch of "if IE do something basic" because the cool stuff (WebGL for example) is broken. Let's try to apply this thinking to our registration function. If Stripe is down (hey, it could happen), you still want your users to be able to register, right? Maybe even take their info and just re-verify when Stripe is back up. Let's look at how we can do that.

Following what we learned before, let's first create a branch with `git checkout -b transactions`. That should put us in a clean environment to work on this change. Now let's write a unit test describing what we are looking for:

```
1  def test_registering_user_when_stripe_is_down(self):
2
3      #create the request used to test the view
4      self.request.session = {}
5      self.request.method='POST'
6      self.request.POST = {'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
15      #mock out Stripe and ask it to throw a connection error
16      with mock.patch('stripe.Customer.create', side_effect =
17                     socket.error("Can't connect to Stripe")) as stripe_mock:
18
19          #run the test
20          resp = register(self.request)
21
22
23          #assert there is a record in the database without Stripe id.
24          users = User.objects.filter(email="python@rocks.com")
25          self.assertEquals(len(users), 1)
26              self.assertEquals(users[0].stripe_id, '')
```

That should do it. We have just more or less copied the test for `test_registering_new_user_returns_successfull`
but removed all the databases 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. But that's exactly what we want (remember the TDD Fail,
Pass, Refactor loop). Don't forget to commit that change.

Ok, how can we get it to work? First thing is to update the `Customer.create` method to handle that
pesky socket.error so we don't have to deal with it.

```
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:
11              return None
```

This way when Stripe is down, our call to `Customer.create` will just return None. I prefer this type
of design so I don't have to put try-except blocks everywhere.

Rerun our test; it will still fail. So now we need to modify the registration function. Basically all we
have to do is change the part of the `register` function that saves the user.

We will change:

```
1  try:
2      user = User.create(cd['name'], cd['email'], cd['password'],
3                          cd['last_4_digits'], customer.id)
4  except IntegrityError:
```

to:

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

Basically, we break the single insert on the database into an insert and then an update. Running the test still fails, and in fact we will now have a failure with the `test_registering_new_user_returns_succesfully` test failing as well. But not to worry; it's just a simple change to the `User.create()` function providing a default value for 'stripe_id“:

```
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
```

All we did there was to give `stripe_id` the default value of an empty string; that way, we don't get an error for passing it in. Since we aren't passing in the `stripe_id` when we initially create the user anymore, we have to change our `test_registering_new_user_returns_succesfully` 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.

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 DB mocks out of `test_registering_new_user_returns_succesfully` the test looks like this:

```
1  def get_mock_cust():
2      class mock_cust():
```

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

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. Probably time for a commit. :)

At this point we now let a user register 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.

> **Note:** The solution I'm about to propose isn't the most elegant, but I need an example to use for database transactions that fits into the running example! Plus, I'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 the credit cards... umm I mean, 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*:

```
class Unpaid_users(models.Model):
    email = models.CharField(max_length=255, unique=True)
    last_notification = models.DateTimeField(default=datetime.now())
```

> **Note:** I'm intentionally leaving off foreign key constraints for now... we may come back to this later.

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

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

In other words, make sure 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 (note: I'll just show the important parts of the function).

```
cd = form.cleaned_data
try:
    user = User.create(cd['name'], cd['email'], cd['password'],
                       cd['last_4_digits'])

    if customer:
        user.stripe_id = customer.id
        user.save()
    else:
        UnpaidUsers(email=cd['email']).save()

except IntegrityError:
    form.addError(cd['email'] + ' is already a member')
else:
    request.session['user'] = user.pk
    return HttpResponseRedirect('/')
```

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

But are we...?

Improved Transaction Management

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

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_ma` just to name a few. Fortunately, with Django 1.6 that all goes out the door. You only really need to know about a couple functions now, and we will get to those in just a second. 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:

```
# START
note1 = Note(title="my first note", text="Yay!")
note2 = Note(title="my second note", text="Whee!")
note1.save()
Note2.save()
# 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?

Yeah, 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?

Now, welcome to Django 1.6. Simply remember that in Django 1.6, 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!

Enough theory. Let's code.

Coming back to our earlier question, is our registration function really Golden? 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.

```
cd = form.cleaned_data
try:
    user = User.create(cd['name'], cd['email'], cd['password'],
                       cd['last_4_digits'])

    if customer:
        user.stripe_id = customer.id
        user.save()
    else:
        UnpaidUsers(email=cd['email']).save()

except IntegrityError:
    form.addError(cd['email'] + ' is already a member')
```

Did you spot the issue? What happens if the `UnpaidUsers(email=cd['email']).save()` line fails? Well, then you will 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 all or nothing. 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 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:

```
@mock.patch('payments.models.UnpaidUsers.save', side_effect =
            IntegrityError)
def test_registering_user_when_strip_is_down_all_or_nothing(self, save_mock):

```

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

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

If we run the test:

```
1  $ ./manage.py test ../tests
2  Creating test database for alias 'default'...
3  ...................F....
4  .
5  ............
6  ======================================================================
7  FAIL: test_registering_user_when_strip_is_down_all_or_nothing
8      (tests.payments.testViews.RegisterPageTests)
8  ----------------------------------------------------------------------
9  Traceback (most recent call last):
10   File
      "/Users/j1z0/.virtualenvs/django_1.6/lib/python2.7/site-packages/mock.py",
      line 1201, in patched
11     return func(*args, **keywargs)
12   File
      "/Users/j1z0/Code/RealPython/mvp_for_Adv_Python_Web_Book/tests/payments/testViews.py",
```

```
     line 266, in test_registering_user_when_strip_is_down_all_or_nothing
13      self.assertEquals(len(users), 0)
14 AssertionError: 1 != 0
15
16 ----------------------------------------------------------------------
17 Ran 38 tests in 0.636s
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....

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

1. The recommended way.

According to Django 1.6 documentation

> Django provides a single API to control database transactions.
>
> atomic(using=None, savepoint=True)
>
> Atomicity is the defining property of database transactions. atomic allows us to create a block of code within which the atomicity on the database is guaranteed. If the block of code is successfully completed, the changes are committed to the database. If there is an exception, the changes are rolled back.

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:

```python
from django.db import transaction

try:
    with transaction.atomic():
        user = User.create(cd['name'], cd['email'], cd['password'],
                           cd['last_4_digits'])

        if customer:
            user.stripe_id = customer.id
            user.save()
        else:
            UnpaidUsers(email=cd['email']).save()

except IntegrityError:
    form.addError(cd['email'] + ' is already a member')
```

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!

2. Using a decorator

We can also trying 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 (i.e., 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:

```
from django.db import transaction

try:
    with transaction.atomic():
        user = User.create(cd['name'], cd['email'], cd['password'],
                           cd['last_4_digits'])

        if customer:
            user.stripe_id = customer.id
            user.save()
        else:
            UnpaidUsers(email=cd['email']).save()

except IntegrityError:
    form.addError(cd['email'] + ' is already a member')
```

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 (i.e., the `form.addError` line), the rollback will be done 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.

3. 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 / 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 fail 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 was raised and not caught), then all savepoints will get rolled back as well.

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

```python
@transaction.atomic()
def save_points(self,save=True):

    user = User.create('jj','inception','jj','1234')
    sp1 = transaction.savepoint()

    user.name = 'starting down the rabbit hole'
    user.stripe_id = 4
    user.save()

    if save:
        transaction.savepoint_commit(sp1)
    else:
        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:

```python
user.name = 'starting down the rabbit hole'
user.stripe_id = 4
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:

```python
def test_savepoint_rollbacks(self):

    self.save_points(False)

```

```
5        #verify that everything was stored
6        users = User.objects.filter(email="inception")
7        self.assertEquals(len(users), 1)
8
9        #note the values here are from the original create call
10       self.assertEquals(users[0].stripe_id, '')
11       self.assertEquals(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.assertEquals(len(users), 1)
20
21       #note the values here are from the update calls
22       self.assertEquals(users[0].stripe_id, '4')
23       self.assertEquals(users[0].name, 'starting 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 = 'starting 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 rolledback.

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:

```python
@transaction.atomic()
def save_points(self,save=True):

    user = User.create('jj','inception','jj','1234')
    sp1 = transaction.savepoint()

    user.name = 'starting down the rabbit hole'
    user.save()

    user.stripe_id = 4
    user.save()

    if save:
        transaction.savepoint_commit(sp1)
    else:
        transaction.savepoint_rollback(sp1)

    try:
        with transaction.atomic():
            user.create('limbo','illbehere@forever','mind blown',
                '1111')
            if not save: raise DatabaseError
    except DatabaseError:
        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 rolledback upon exiting the context manager.

Thus the following two tests describe their behavior here:

```python
def test_savepoint_rollbacks(self):

    self.save_points(False)

    #verify that everything was stored
    users = User.objects.filter(email="inception")
    self.assertEquals(len(users), 1)
```

```python
         #savepoint was rolled back so we should have original values
         self.assertEquals(users[0].stripe_id, '')
         self.assertEquals(users[0].name, 'jj')

         #this save point was rolled back because of DatabaseError
         limbo = User.objects.filter(email="illbehere@forever")
         self.assertEquals(len(limbo),0)


     def test_savepoint_commit(self):
         self.save_points(True)

         #verify that everything was stored
         users = User.objects.filter(email="inception")
         self.assertEquals(len(users), 1)

         #savepoint was commited
         self.assertEquals(users[0].stripe_id, '4')
         self.assertEquals(users[0].name, 'starting down the rabbit hole')

         #save point was commited by exiting the context_manager without an
     exception
         limbo = User.objects.filter(email="illbehere@forever")
         self.assertEquals(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.

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 statment..

```
START TRANSACTION
SELECT * FROM DEV_JOBS WHERE PRIMARY_SKILL = 'PYTHON'
END TRANSACTINO
```

** atomic Contect Manager **

Django 1.6 based transaction management has one main API which is atomic. Using atomic wraps a code block in a db transaction.

```
with transaction.atomic():
    user1.save()
    unpaidUser1.save()
```

** atomic Decorator **

Django 1.6 based transaction management has one main API which is atomic. Using atomic as a decorator wraps a function in db transaction.

```
@transaction.atomic()
def saveUser(user1, unpaidUser1)

    user1.save()
    unpaidUser1.save()
```

** Transaction per http Request **

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`

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        '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. The following example will

```
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 = 'starting 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? Verify your expectation with a test a 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.

    For reference, the code I'm referring to is below (and I've already removed 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 = 'starting 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
```

2. Build your own transaction management system. :) No just joking, but you could read through the complete Django documentation on the new transaction management features here.

Chapter 7

Building a Membership Site

It's time to make something cool

Up until now we have covered the nitty gritty of unit testing; how to do TDD; git branching for teams; upgrading to Django 1.6, 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.

That's all about to change. In my mind, 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, I'm a firm believer in the practice of "full-stack" development. Full-stack developers can develop complete features from database design to Javascript animation. About the only thing I don't include in the repertoire of a "full-stack" developer is design. I don't expect everybody to be artistic (I know I'm not).

The ability to develop both in a back-end language such as Python as well as THE front end language of Javascript 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 are still focusing 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. I'm not going to go into a whole spiel about the benefits of agile and user stories, because there are plenty of excellent books devoted to that. But I want to at least document a few of the user stories that we are going to implement in the coming chapters so as to paint the picture of where we are headed.

A Membership Site

What do we have so far? Currently we have the following implemented:

- Static homepage
- User login
- User 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 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 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 meetups, 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.

I could go on for days, but that's enough to fill a book I think. :) 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. I'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 **US 1**. See you on the next page. :)

Chapter 8

Bootstrap 3 and Best Effort Design

I said in the previous chapter that I don't expect most full-stack developers to be designers. 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 like myself, with absolutely zero artistic ability, can make a site look presentable. I'm not saying the site is 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 is the first User Story that we will be working 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 signup button right smack dab in the middle of the page. Also, towards the bottom of the page there should be an "about" link and a "contact us" link so younglings 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.

That said, we need something that looks cool and gets the "youngling" to click on the signup button. If I 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. 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 Michael created in the second course, "Real Python Part 2". He created the application with Bootstrap. So we are already using Bootstrap, but it's version 2. Currently Bootstrap is on version 3 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, since we are going to redesign anyway, I'm going to start over (more or less) with the front end. If you have a large site with a lot of pages, I wouldn't recommend starting from scratch, but as we only have a few pages and it simplifies the explanation, we're going to go ahead and do that.

Installing Bootstrap 3

The first thing to do is grab Bootstrap. It could be served from a CDN; we will look at how to do that later we we get to the chapter on deployment. For now, let's download it locally and chuck it in our static directory.

1. Download bootstrap distribution files from here.

2. Unzip the files and put them in the *django_ecommerce/static* file directory. I like to remove the *dist* folder so I end up with a folder structure like:

```
static
--css
  --bootstrap.min.css
  ...
--fonts
  --glyphicons-halflings-regular.ttf
  --glyphicons-halflings-regular.svg
--js
  --jquery-1.11.0.min.js
  --bootstrap.min.js
  --application.js
```

I've left out some of the files, but I think you get the idea. One critical file that we must not forget is jQuery. As of this writing 1.11.0 is the most recent version in the 1.x branch, which is what Bootstrap uses. Download that from the jQuery download page, and add it to the *js* folder.

```
    A jQuery Upgrade:
    On a side note, do keep in mind that if you have a "real" site using a
    version of jQuery earlier than
    1.9, you should take the time to look through the release notes and also
    look at the jQuery Migrate
    plugin.  Because our example site only makes very simple use of jQuery, it's
    not much of an issue for
    us, but in a production site (as always) make sure you approach the upgrade
    carefully and test
    thoroughly.
```

3. Once you have Bootstrap in the static folder then you can 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. It should now look like:

```
<!DOCTYPE html>
<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, initial-scale=1">
 7     <title>Bootstrap 101 Template</title>
 8
 9     <!-- Bootstrap -->
10     <link href="css/bootstrap.min.css" rel="stylesheet">
11
12     <!-- HTML5 Shim and Respond.js IE8 support of HTML5 elements and media
       queries -->
13     <!-- WARNING: Respond.js doesn't work if you view the page via file:// -->
14     <!--[if lt IE 9]>
15       <script
     src="https://oss.maxcdn.com/libs/html5shiv/3.7.0/html5shiv.js"></script>
16       <script
     src="https://oss.maxcdn.com/libs/respond.js/1.4.2/respond.min.js"></script>
17     <![endif]-->
18   </head>
19   <body>
20     <h1>Hello, world!</h1>
21
22     <!-- jQuery (necessary for Bootstrap's JavaScript plugins) -->
23     <script src="https://code.jquery.com/jquery.js"></script>
24     <!-- Include all compiled plugins (below), or include individual files as
       needed -->
25     <script src="js/bootstrap.min.js"></script>
26   </body>
27 </html>
```

4. Once you have that in place, from the *django_ecommerce* directory run `./manage.py` `runserver` and navigate to the url shown. You should see 'Hello, world!'

Installation done.

Making Bootstrap your own

Some of the paths in the above template aren't necessarily correct, so we will fix that first, then we will put our Star Wars theme on the main page.

The first thing to do is use the Django `{% load static %}` directive 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 to lines such as:

```
<link href= "{% static "css/bootstrap.min.css" %}" rel="stylesheet">
<script src="{% static "js/jquery-1.11.0.min.js" %}"></script>
<script src="{% static "js/bootstrap.min.js" %}"></script>
```

With that out of the way, our *base.html* should have the basic setup we need. Of course we probably want to change the title and things to make it look nice. And since this is a Star Wars site, let's 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; 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. To do this for our case, let's add a new file called *static/css/mec.css*. To start with, it should look like this:

```
@font-face {
  font-family: 'starjedi';
  /*IE9 */
  src: url('../fonts/Starjedi.eot');
  /* Chrome, FF, Opera */
  src: url('../fonts/Starjedi.woff') format('woff'),
  /* android, safari, iOS */
  url('../fonts/Starjedi.ttf') format('truetype'),
  /* legacy safari */
  url('../fonts/Starjedi.svg') format('svg');
  font-weight: normal;
  font-style: normal;
}

h1 {
  font-family: 'starjedi', sans-serif;
}
```

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 directives like `static` available to us.

Also make sure that you have copied the actual font files into your `static/fonts` directory. Alterna-

tively you can grab the `jedi-font` tag from the git repo like:

```
$ git checkout tags/jedi-font
```

This will get our `h1` to use the cool startjedi font so our site can look more authentic. Now let's work on a nice layout. This is where we are going:
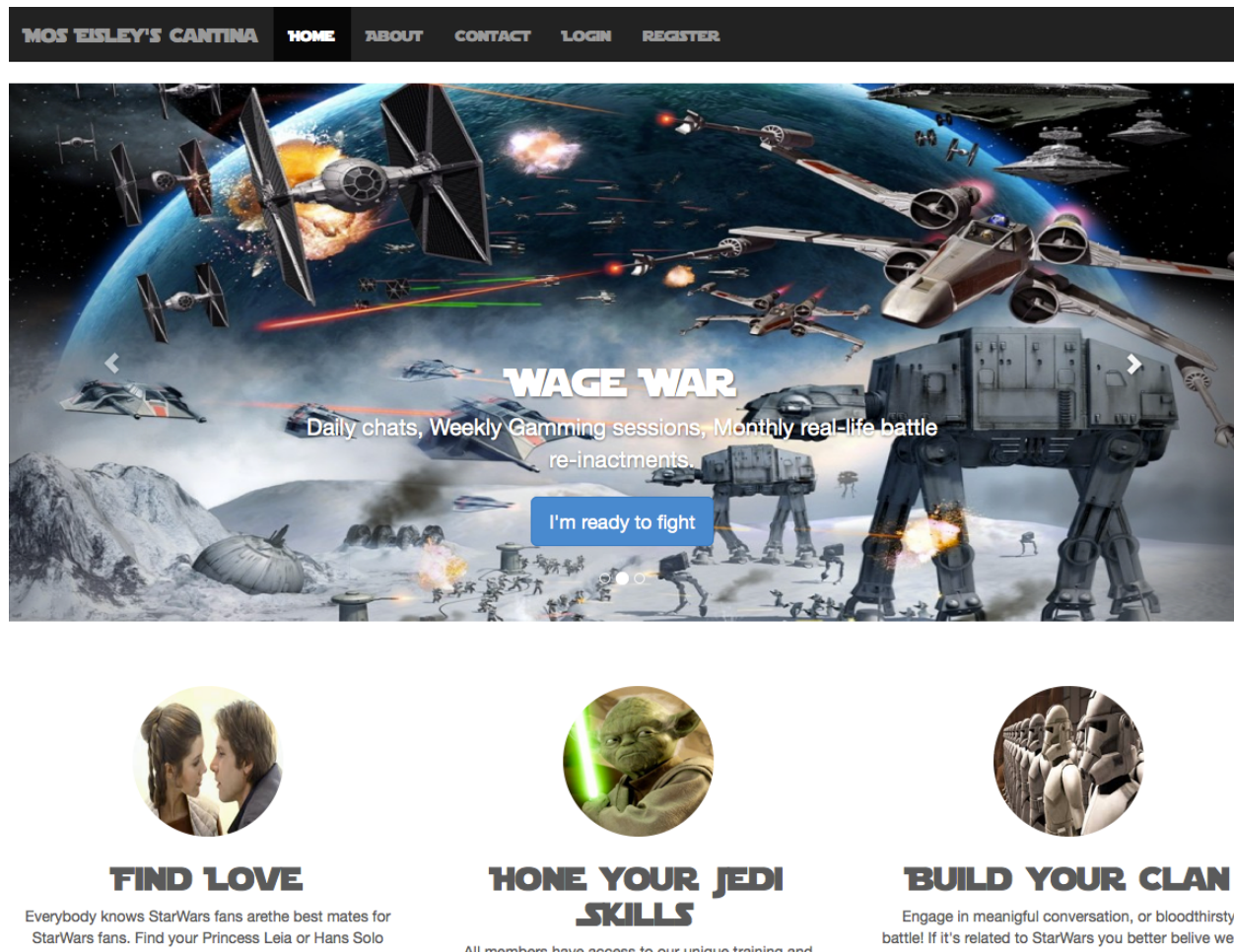
Figure 8.1: main page

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 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     <div class="container">
4
5      <div class="navbar navbar-inverse navbar-static-top" role="navigation">
6        <div class="container">
7          <div class="navbar-header">
8            <button type="button" class="navbar-toggle" data-toggle="collapse"
     data-target=".navbar-collapse">
9        <span class="sr-only">Toggle navigation</span>
10       <span class="icon-bar"></span>
11       <span class="icon-bar"></span>
12       <span class="icon-bar"></span>
13           </button>
14           <a class="navbar-brand" href="#">Mos Eisley's Cantina</a>
15         </div>
16         <div class="navbar-collapse collapse">
17           <ul class="nav navbar-nav">
18       <li class="active"><a href="{% url 'home' %}">Home</a></li>
19       <li><a href="/pages/about">About</a></li>
20       <li><a href="{% url 'contact' %}">Contact</a></li>
21       {% if user %}
22         <li><a href="{% url 'sign_out' %}">Logout</a></li>
23             {% else %}
24       <li><a href="{% url 'sign_in' %}">Login</a></li>
25       <li><a href="{% url 'register' %}">Register</a></li>
26       {% endif %}
27           </ul>
28         </div> <!-- end navbar links -->
29       </div> <!-- end container -->
30     </div> <!-- end navbar -->
31
32   </div> <!-- end navbar container -->
33 </div><! -- end navbar-wrapper -->
```

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 important 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.
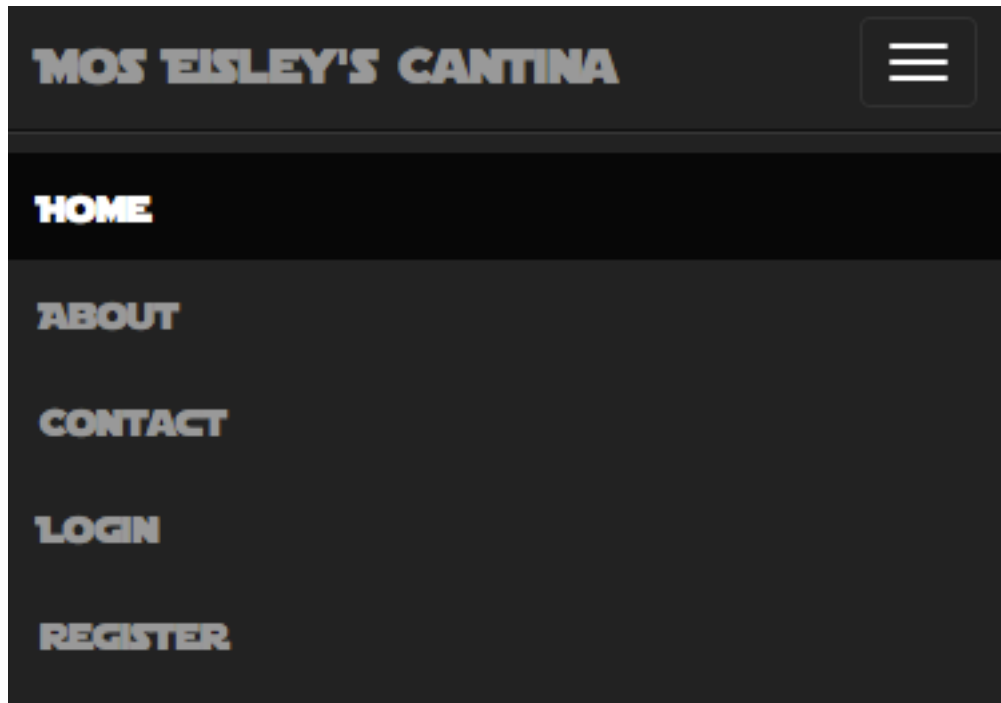
Figure 8.2: responsive dropdown

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:

```
$ ./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 *nix 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. My IP address was 192.168.1.12, so on my phone I just put in 192.168.1.12:8000 and then I could see the site.

I always get a certain sense of satisfaction out of seeing my websites on my own phone. :) There are also simpler ways to test out the responsiveness of bootstrap. The simplest being simply resize your browser and what bootstrap automatically adjust. Also you can use an online tool like viewpoint-resizer.

To make the navbar use our cool star_jedi font, update *mec.css* like this:

```css
h1,
.nav li,
.navbar-brand
{
  font-family: 'starjedi', sans-serif;
}
```

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 startjedi font. This can be done by adding the following to `mec.css`:

```
.navbar-brand
{
  height : 40px;
}
```

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:

```
<footer>
  <p class="pull-right"><a href="">Back to top</a></p>
  <p class="pull-left">&copy; 2014 <a href="http://j1z0.com">The Jedi
    Council</a></p> <p class="text-center"> &middot;  Powered by about 37 AT-AT
    Walkers, Python 3 and Django 1.6 &middot; </p>
</footer>
```

More or less the same, right? But you may be saying to yourself, "What is this `<footer>` thing? Why shouldn't we just use a `<div>`? Well I'm glad you asked... (or did I ask for you?) Because that brings us into our next topic.

HTML5 Sections and the Semantic Web

You may have 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. HTML5 is trying to make the web much more semantic. Before we get into the HTML5 part though, let's define "Semantic Web". Let's take the first two paragraphs from the Wikipedia page verbatim:

> 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 the DATA on the web more accessible, specifically more accessible programmatically. There are a number of techniques 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 I just want to 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. Here it is again:

```
1  <div class="navbar-wrapper">
2    <div class="container">
3
4      <div class="navbar navbar-inverse navbar-static-top" role="navigation">
5        <div class="container">
6          <div class="navbar-header">
7            <button type="button" class="navbar-toggle" data-toggle="collapse"
    data-target=".navbar-collapse">
8        <span class="sr-only">Toggle navigation</span>
9        <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 container -->
32 </div><! -- end navbar-wrapper -->
```

Now imagine you were a computer program trying to figure out the *meaning* of that section of HTML. It's 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, but that is actually part of the HTML5 semantic web specification (so you can't use that for now); specifically it is to provide a way for assistive technology to better understand the interface.

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 step we can implement to start to make our 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. In HTML5 the following section (or section-like) tags are defined:

- **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. Maybe an author's bio could be sectioned off from the blog post in a section tag.

- **article** - Same as a section tag, but specifically for content that should be syndicated, a blog post being the best example.

- **aside** – Used for somewhat related content that isn't necessarily critical to the rest of the content. There are several places in this book where I use asides. For example, there is an aside just a few paragraphs up that start with "The astute reader". Keep in mind oftentimes asides appear indented (such as is the case with pullquotes). However, the aside tag itself says nothing about placement; it describes the data only.

- **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.

- **nav** – Basically for site navigation, like our navbar above (which we are about to rewrite using some of these tags. :)

- **footer** – Doesn't need to be at the bottom of the page. Footer elements contain information about the containing section they are in. Oftentimes (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.

So with our new understanding of some of the HTML section elements, let's rewrite our navbar using them:

```
<div class="container">
  <nav class="navbar navbar-inverse navbar-static-top" role="navigation">
    <header class="navbar-header">
      <button type="button" class="navbar-toggle" data-toggle="collapse"
    data-target=".navbar-collapse">
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="{% url 'home' %}">Mos Eisley's Cantina</a>
    </header>
    <div class="navbar-collapse collapse">
      <ul class="nav navbar-nav">
        <li class="active"><a href="{% url 'home' %}">Home</a></li>
        <li><a href="/pages/about">About</a></li>
        <li><a href="{% url 'contact' %}">Contact</a></li>
        {% if user %}
          <li><a href="{% url 'sign_out' %}">Logout</a></li>
        {% else %}
          <li><a href="{% url 'sign_in' %}">Login</a></li>
          <li><a href="{% url 'register' %}">Register</a></li>
        {% endif %}
      </ul>
    </div>
  </nav>
</div>
```

There are still a lot of divs there, but we have replace two of them with more meaningful section elements. 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**. 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 pretend you were a program (like a web scrapper or an automated testing tool) 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 jump through a whole bunch of loops and special cases just to visit all the pages on the site. Now you can basically have the following pseudo-code:

```
access site

for <a> in <nav>:
    click <a>
    check for interesting sections
    extract data

#now return to main landing page
<nav>.<header>.<a>.click()
```

Making your site more accessible to scrappers or such may seem like not much of a big deal, but think of several potential applicaitons. Such as:

- Specialized interfaces * : Imagine tring 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. No imagine if that smart watch could determine the conent and respone 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 divs.

- 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 or writeups 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 writting maintainable code is a huge part of Software Craftmanship. If you go back and look at our previous example with all divs vs the semantic markup I think you'll see it's much easier to see whats going on at a glance. This may not seem like much, but the simpler things are to understand the easier it is to maintain, and every bit counts. Further this is a pretty small example, 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`. This makes it tought 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. We now have a navigation structure that not only looks cool, 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 the HTML5 section tag that is intended for 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 book, but if you want to find out more information (it's a huge topic), 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 to do is put 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:

```
<section id="myCarousel" class="carousel slide" data-ride="carousel">
  <!-- Indicators -->
  <ol class="carousel-indicators">
    <li data-target="#myCarousel" data-slide-to="0" class="active"></li>
    <li data-target="#myCarousel" data-slide-to="1"></li>
    <li data-target="#myCarousel" data-slide-to="2"></li>
  </ol>
  <div class="carousel-inner">
    <figure class="item active">
          <img src="{% static 'img/darth.jpg' %}" alt="Join the Dark Side">
        <figcaption class="carousel-caption">
          <h1>Join the Dark Side</h1>
          <p>Or the light side.  Doesn't matter.  If your into Star Wars then
    this is the place for you.</p>
          <p><a class="btn btn-lg btn-primary" href="{% url 'register' %}"
    role="button">Sign up today</a></p>
        </figcaption>
    </figure>
    <figure class="item">
          <img src="{% static 'img/star-wars-battle.jpg' %}" alt="Wage War">
        <figcaption class="carousel-caption">
          <h1>Wage War</h1>
          <p>Daily chats, Weekly Gaming sessions, Monthly real-life battle
    re-inactments.</p>
          <p><a class="btn btn-lg btn-primary" href="#" role="button">I'm ready
    to fight</a></p>
        </figcaption>
    </figure>
    <figure class="item">
          <img src="{% static 'img/star-wars-cast.jpg' %}" alt="Meet People">
        <figcaption class="carousel-caption">
          <h1>Meet fellow Star Wars Fans</h1>
          <p>Join forces with fellow padwans and Jedi who lives near you.</p>
          <p><a class="btn btn-lg btn-primary" href="#" role="button">Check the
    Members Map</a></p>
        </figcaption>
```

```
32        </figure>
33      </div>
34      <a class="left carousel-control" href="#myCarousel" data-slide="prev">
35          <span class="glyphicon glyphicon-chevron-left"></span></a>
36      <a class="right carousel-control" href="#myCarousel" data-slide="next">
37          <span class="glyphicon glyphicon-chevron-right"></span></a>
38  </section>
```

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>
```

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.

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          <img src="{% static 'img/darth.jpg' %}" alt="Join the Dark Side">
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>
```

`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, that 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. Official W3C spec

Just like with the HTML5 section tags, we are replacing the overused `div` element with an element `figure` that provides some meaning.

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 an 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">
2    <span class="glyphicon glyphicon-chevron-left"></span></a>
3  <a class="right carousel-control" href="#myCarousel" data-slide="next">
4    <span class="glyphicon glyphicon-chevron-right"></span></a>
```

And that's it. You now have a cool carousel telling visiting younglings about the benefits of joining the site. I've included some of the pictures in the repo so you can try it out for yourself, or you can use your own.

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 dump a bunch of extra HTML after the carousel like so:

```
1   <section class="container marketing">
2    <!-- Three columns of text below the carousel -->
3    <div class="row">
4      <div class="col-lg-4">
5          <img class="img-circle" src="{% static 'img/yoda.jpg' %}" width=140
      height=140" alt="Generic placeholder image">
6        <h2>Hone your Jedi Skills</h2>
7        <p>All members have access to our unique training and achievements
      ladders.  Show off your Star Wars skills, progress through the levels and
      show everyone who the top Jedi Master is! </p>
8        <p><a class="btn btn-default" href="#" role="button">View details
      &raquo;</a></p>
9      </div><!-- /.col-lg-4 -->
10     <div class="col-lg-4">
11         <img class="img-circle" src="{% static 'img/clone_army.jpg' %}"
      width=140 height=140 alt="Clone Army">
12        <h2>Build your Clan</h2>
13        <p>Engage in meaningful conversation, or bloodthirsty battle!  If it's
      related to Star Wars you better believe we do it here. :)</p>
14        <p><a class="btn btn-default" href="#" role="button">View details
      &raquo;</a></p>
15     </div><!-- /.col-lg-4 -->
16     <div class="col-lg-4">
```

```
17        <img class="img-circle" src="{% static 'img/leia.jpg' %}" width=140
      height=140 alt="Leia">
18        <h2>Find Love</h2>
19        <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>
20        <p><a class="btn btn-default" href="#" role="button">Sign Up Now
      &raquo;</a></p>
21      </div><!-- /.col-lg-4 -->
22    </div> <!-- /.row -->
23  </section>
```

This gives us three sections each with an image in a circular border (because circles are hot these days), 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">`) 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. In Bootstrap each row has a total of 12 columns. You can break up the page how you want. 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. Now add to that a further identifier, in our case `lg`, but there is also `xsm`, `sm`, and `md` for extra-small, small and medium.

Bootstrap 3, being responsive by default, has the notion of extra-small, small, medium, and large screen sizes. So you can say I want columns only if the screen size is large, which is what the class `col-lg-4` says. 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:
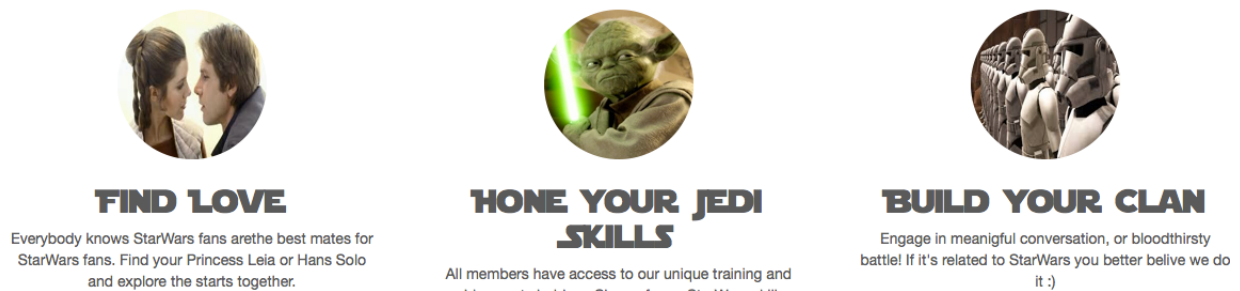
Figure 8.3: large screen size

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 re-sizing 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?

137

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 achievemnts ladders. Show of your StarWars skills, progress through the levels and show everyone who the

Figure 8.4: small screen size

The best place to get all the information about the Bootstrap grid system is in the [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 code, and we aren't really taking advantage of Django's templating system. Let's do that.

Custom template 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 allready familiary with:

- `{% static %}` - provides a way to reference the location of your static folder without hardcoding
- `{% extend 'base.html' %}` - allows for the extending of a parent template
- `{% csrf %}` - Django's 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 loged in" }}` - if value evaluates to False, use the given default i.e. Not logged in

There are tons more builting filters and template tags. You can go through to documentation [here]((https://docs.djangoproject.com/en/1.6/ref/templates/builtins/).

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 are repeating over and over. For example this section of code:

```
<div class="col-lg-4">
    <img class="img-circle" src="{% static 'img/yoda.jpg' %}" width=140
    height=140" alt="Generic placeholder image">
  <h2>Hone your Jedi Skills</h2>
  <p>All members have access to our unique training and achievement ladders.
    Show off your Star Wars skills, progress through the levels and show everyone
    who the top Jedi Master is! </p>
  <p><a class="btn btn-default" href="#" role="button">View details
    &raquo;</a></p>
</div>
```

Could be easily re-factored into a template tag so you wouldn't have to type so much. :) Let's do it.

1. First thing to do is find a place to put the template tags. They need to be inside a Django app in a folder called *templatetags*. If we use our *main* Django app, then (after we create the new *templatetags*) the folder structure should look like:

```
1  main/
2      models.py
3      templatetags/
4          __init__.py
5          marketing.py
6      views.py
```

2. In the `marketing.py` file we will put the following code to define our custom template tag:

```python
1  from django import template
2  register = template.Library()
3
4  @register.inclusion_tag('circle_item.html')
5  def circle_header_item(img_name="yoda.jpg", heading="yoda", caption="yoda",
6                         button_link="register", button_title="View details"):
7      return  {'img' : img_name,
8               'heading'      : heading,
9               'caption'      : caption,
10              'button_link' : button_link,
11              'button_title' : button_title}
```

The first two lines just register the `circle_header_item` as a template tag so you can use it in a template with syntax like:

```
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.

We will skip the `@register` line for just a section. The 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.

Coming back to this line:

```
1  @register.inclusion_tag('circle_item.html')
```

This declares an HTML fragment that will be 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.

The `circle_item.html` file looks like this:

```
1  {% load staticfiles %}
2
3  <div class="col-lg-4">
```

```
4      <img class="img-circle" src="{% static 'img/'|add:img %}" width=140
5      height=140" alt="{{img}}">
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>
```

We have taken our original HTML to define the structure of the item and then used several template variables (the ones we returned from `main.templatetags.marketing.circle_header_item`) to allow for dynamically populating 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 <img class="img-circle" 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` This is the add filter and it's pretty cool. Not only can it add numbers, i.e.:

```
1 {{ num|add:'1' }}
```

-which would produce 5 if num is 4. It can also concatenates lists if the two parameters are lists or strings if the two parameters are strings. Pretty handy filter to have.

In terms of our inclusion tag, when we call it from our index.html it might look like this:

```
1  <section class="container marketing">
2    <!-- Three columns of text below the carousel -->
3    <div class="row">
4      {% circle_header_item img_name='yoda.jpg' heading='Hone your Jedi Skills'
5      caption='All members have access to our unique training and achievements
       ladders.
6          Show off your Star Wars skills, progress through the levels and show
       everyone who
7            the top Jedi Master is!' %}
8
9      {% circle_header_item img_name='clone_army.jp' heading='Build your Clan'
10     caption='Engage in meaningful conversation, or bloodthirsty battle! If it's
11     related to Star Wars you better believe we do it. :)' %}
12
13     {% circle_header_item img_name="leia.jpg" heading="Find Love"
14         caption="Everybody knows Star Wars fans are the best mates for Star Wars
15     fans.  Find your Princess Leia or Han Solo and explore the stars
16     together." button_title="Sign Up Now" %}
17    </div>
18 </section>
```

There you have it! Just three calls to our new template tag and nowhere near as much HTML all over the place. I did however cheat a bit.

Django template tags are not allowed to span multiple lines like I have done in the sample above, so you actually have to string everything together in one super long line. (But if I did that you couldn't read it in the book.) For this reason a lot of people don't use inclusion tags as I have done in this example, but for me I'm always using word-wrap on my editor, so it's not that big of a deal. Also, I like the fact that I don't have to repeat bits of HTML over and over.

Another solution to the long argument list is to give you template tag access to the context with a registration line like the following:

```
@register.inclusion_tag('circle_item.html', takes_context=True)
```

Then you just have to make sure the data that you need is in the context (which can be set in your view function), and then you're done. Doing this will be left as an exercise at the end of the chapter.

A Note on Structure

Bootstrap is a great thing and pairing it with Django templates can make design a breeze while at the same time greatly reducing 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 with 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. *The best advice I can offer to combat this issue is to choose a meaningful structure to organize your templates and stick to it religiously.* It will make your debugging and maintenance much easier.

The default "structure" and what we are currently using in our examples up to this point is just to throw 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 each other.

Let's first address that issue by putting templates in a subfolder of the template directory. The name of the subfolder should correspond with the app that is using the template. Reorganizing our template folder as such would produce a listing like the one below:

```
./templates
|--  contact
|    |--contact.html
|-- flatpages
|    |--default.html
|-- main
|    |--index.html
|    |--user.html
|    |--templatetags
|        |-- circle_item.html
|-- payments
|    |--cardform.html
```

143

```
13   |    |--edit.html
14   |    |--field.html
15   |    |--register.html
16   |    |--sign_in.html
17   |-- contact
18   |    |--contact.html
19   |-- base.html
```

With this type of 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.

In a template you would write something like this:

```
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 *view.py* file, or anywhere in Python, it's the same idea:

```
1   return render_to_response(
2       'payments/sign_in.html',    (1)
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.

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.

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 know the purpose of the template with a simple `ls` command (in Unix), without

144

even having to open up the template. You can use any naming convention you like as long as it works for you.

I like the following convention:

case	naming convention	example
1	start name with __	*__base.html*
2	start name with _	*payments/_cardform.html*
3	put in templatetag sub directory	*@register.inclusion_tag('main/templatetags/circle_item.html')*

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.

Updating the files based upon the above naming convention will give us a file listing like the one below:

```
1   ./templates
2   |-- contact
3   |    |--contact.html
4   |-- flatpages
5   |    |--default.html
6   |-- main
7   |    |--index.html
8   |    |--user.html
9   |    |--templatetags
10  |        |-- circle_item.html
11  |-- payments
12  |    |--_cardform.html
13  |    |--_field.html
14  |    |--edit.html
15  |    |--register.html
16  |    |--sign_in.html
17  |-- contact
18  |    |--contact.html
19  |-- __base.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 the *_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.

The final thing I would like to 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:

```
1   {% load marketing %}
```

The first question I usually have after reading that is, "Well where are the marketing template tags, and what's in there?"

Let's first rename the marketing to `main_marketing` so at a glance we can tell where the template tags are located. 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. I've seen Django projects that 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.*

The way that has worked for me is to use the convention taglibname__tagname. In this case `circle_header_item` becomes `marketing__circle_item`. This way, if I find it in a template, I know it comes from a library called marketing, and if I just jump to the top of the HTML file, I'll see the `{% load main_marketing %}` and thus I'll know to look for the code in `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.

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 huge library, and we have barely scratched the surface. I encourage you to read more about Bootstrap on the official website. There are a lot 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 guys help make your web site more accesible programatically, 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 accesible is generally a good thing.
2. 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 tas 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 I strongly encourage you to go through the exercises on this one and ensure you understand everything that we talked about in this chapter.

Exercises

1. Bootstrap is a front-end framework ,and although we didn't touch much on it in this chapter, Bootstrap uses a number of CSS classes to place things on the page, making it all 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 as in the example above?

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:

```
@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_respnose`. 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 9

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). Returning to the user stories we discussed in Chapter 7, this is what we said we wanted to do with the membership site:

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 happenings 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:

This is what a registered user will see after login. If you recall, the registered user page is in *templates/main/users.html*. If we add three boxes to the *users.html* page - e.g., the '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 %}
```
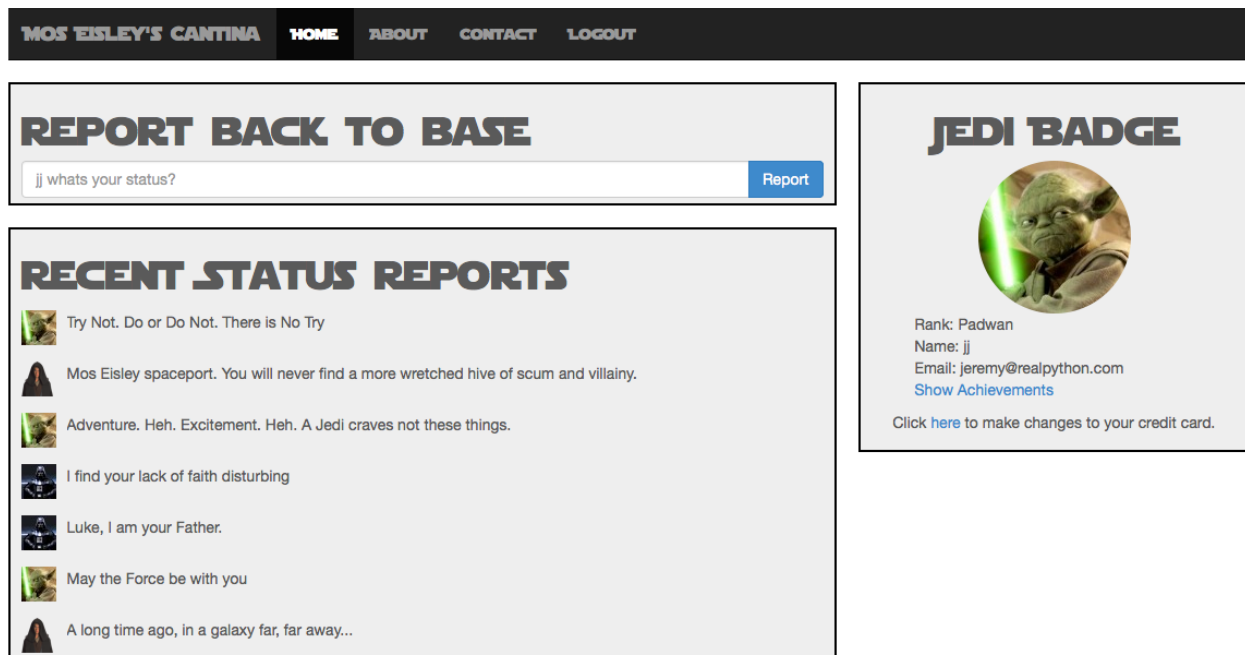
Figure 9.1: Members' Page

Looks pretty simple, right? The first thing that we have done is to put in the basic Bootstrap scaffolding. Remember that Bootstrap uses a grid system, and we can access that grid system by creating classes that use the `row` and `col` classes. 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
.col-xs-	< 768px	Phones
.col-sm-	>= 768px	Tablets
.col-md-	>= 992 px	Desktops
.col-lg-	>= 1200 px	Large Desktops

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, there are three includes that we used.

```
{% include "main/_statusupdate.html" %}
{% include "main/_lateststatus.html" %}    150
{% include "main/_jedibadge.html" %}
```

After the last chapter you should be familiar with includes; they just let us include a separate template

Showing User Info for the Current User

The *main/_jedibadge.html* template displays information about the current logged in user. The template looks like this:

```html
<!-- The jedi badge info box, shows user info -->
{% load staticfiles %}
<section class="info-box" id="user_info">
  <h1>Jedi Badge</h1>
    <img class="img-circle" src="{% static 'img/yoda.jpg' %}" width="140"
        height="140" alt="user.avatar"/>
    <ul>
      <li>Rank: {{user.rank}}</li>
      <li>Name: {{user.name}}</li>
      <li>Email: {{user.email}}</li>
      <li><a href="#">Show Achievements</a>
    </ul>
    <p>Click <a href="{% url 'edit' %}">here</a> to make changes to your credit
    card.</p>
</section>
```

We start with a section tag that wraps the whole section and gets its styling from a class called `info-box`. The css applied to both our `info-box` and the rest of the member page is in our `mec.css`:

```css
.info-box
{
    border: 2px solid #000000;
    margin-bottom: 20px;
    padding-left: 10px;
    padding-right: 10px;
    padding-bottom: 5px;
    background-color: #eee;
}
#user_info
{
    text-align: center;
    margin-left: 20px;
}
#user_info ul
{
    list-style-type: none;
    text-align: left;
}
.member-page
{
    padding-top: 20px;
    padding-bottom: 40px;
```

```
24      background-color: white;
25      margin-left: 0px;
26      margin-right: 0px;
27      margin-top: -20px;
28   }
```

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 having rank a of youngling or padwan.

Adding the rank attribute is just another line to the `payments.models.User` object as:

```
1   rank = models.CharField(max_length=50, default="Padwan")
```

We default the value to "Padwan" because all registered users start with that value, and unregistered 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 will 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 give you some error because it can't add a column to a table that has users in it. You will need to first remove all the users and then the `syncdb` can continue error free.

For right now that is OK; 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 really 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 loosing data. Chapter 11 will cover migrations in more detail.

For now we will just drop the database and then run `./manage.py syncdb` so we can get the user table updated. Since we are developing the membership page (which requires 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 Chapter 2 as a means of loading data for unit testing. You can also use fixtures to load an initial set of data that you always want in the database when you sync your database. While I said in Chapter 2 that I don't generally like fixtures for unit testing because they make things harder to debug, in this case since we are going to be 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. So if you allow me to make an exception to my own rule, let's create a fixture. :)

Since users are in the payments application, we should put our fixture there. To create a fixture that is run each time `./manage.py syncdb` is run (which is also every time our unit test are run), do the following:

1. create a directory called *payments/fixture/*
2. (after registering a few users) run `./manage.py dumpdata payments.User > payments/fixtures/initial`

`manage.py dumpdata` will spit 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`. 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:

```
[{"pk": 1,
  "fields": {
    "last_login": "2014-03-11T08:58:20.136",
    "rank":"Padwan",
    "name": "jj",
    "password":
    "pbkdf2_sha256$12000$c8TnAstAXuo4$agxS589FflHZf+C14EHpzr5+EzFtS1V1tfCYJFVynYU=",
    "email": "jeremy@realpython.com",
    "stripe_id": "cus_3e8fBA8rIUEg5X",
    "last_4_digits": "4242",
    "updated_at": "2014-03-11T08:58:20.239",
    "created_at": "2014-03-11T08:58:20.235"},
  "model": "payments.user"},
{"pk": 2, "fields": {
  "last_login": "2014-03-11T08:59:19.464",
  "rank":"Jedi Knight",
  "name": "kk",
  "password":
    "pbkdf2_sha256$12000$bEnyOYJkIYWS$jqwLJ4iijmVgPHu9na/Jncli5nJnxbl47XK8kIMq2JM=",
  "email": "k@k.com",
  "stripe_id": "cus_3e8gyBJlWAu8u6",
  "last_4_digits": "4242",
  "updated_at": "2014-03-11T08:59:19.579",
  "created_at": "2014-03-11T08:59:19.577"},
"model": "payments.user"},
{"pk": 3, "fields": {
  "last_login": "2014-03-11T09:12:09.802",
  "rank":"Jedi Master",
  "name": "ll",
  "password":
    "pbkdf2_sha256$12000$QE2hn0nj0IWm$Ea+IoZMzv6KYV2ycpe+g7afFWi2wPSSyaVURkiOqoaw=",
  "email": "vader@softworks.com.my",
  "stripe_id": "cus_3e8tB7EaspoOiJ",
  "last_4_digits": "4242",
  "updated_at": "2014-03-11T09:12:10.033",
  "created_at": "2014-03-11T09:12:10.029"},
"model": "payments.user"}]
```

With that, you won't have to worry about re-registering users every time you run a unit test or you resync your database. Buf 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` (`tests.payments.testUserMode.UserModelTest`) should now be failing. Let's fix it really quick:

```
def test_get_by_id(self):
    self.assertEqual(User.get_by_id(self.test_user.id), self.test_user)
```

Before we were hard-coding the id to be 1, which is OK 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.

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:

```
from django import template
from urllib.parse import urlencode
import hashlib

register = template.Library()

@register.simple_tag
def gravatar_img(email, size=140):
    url = get_url(email, size)
    return '''<img class="img-circle" src="%s" height="%s" width="%s"
            alt="user.avatar" />''' % (url, size, size)

def get_url(email, size=140):
    default = 
    'http://starwars.com//img/explore/encyclopedia/characters/anakinskywalker_relationship.png'
    query_params = urlencode([('s', str(size)),
                              ('d', default)])

    return ('http://www.gravatar.com/avatar/' +
            hashlib.md5(email.lower().encode('utf-8')).hexdigest() +
            '?' + query_params)
```

Let's go through this code a bit.

- Lines 1-8: import what we need 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 (because that is what gravatar requires). Finally we add the query_params.

We are going to 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://starwars.com//img/explore/encyclopedia/characters/anakinskywalker_relationship.png'
2  query_params = urlencode([('s', str(size)),
3                            ('d', default)])
```

urlencode is important as it will do 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 I've just picked a random Star Wars image. There are several other querystring 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 href="#">Show Achievements</a>
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 2 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: FYI - there are actually a number of gravatar plugins available on GitHub, most of which are basically the same thing we just implemented. While I'm not necessarily a fan of re-inventing the wheel, I don't see the point of downloading 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 rerun our tests and make sure nothing fails, as well as add some new tests for the gravatar tag. As this is review of Chapter 2 and 3, I won't go into the details of how to do it here, but I will add it as an exercise at the end of the chapter - because it's always good to get some extra practice.

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.

Figure 9.2: 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:

```
<!-- represents the status update info box -->
<section class="info-box" id="updates">
  <h1>Report back to base</h1>
  <form accept-charset="UTF-8" action="{% url 'report' %}"
    role="form" method="post">{% csrf_token %}
    <div class="input-group">
      <input id="status" type="text" class="form-control" name="status"
```

```
8              placeholder="{{user.name}} whats your status?">
9          <span class="input-group-btn">
10              <button class="btn btn-primary" type="submit">Report</button>
11          </span>
12      </div>
13  </form>
14  </section>
```

It's just a simple form that POSTS the `report` URL, which I have defined in the `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.model.StatusReport` is below:

```
1  class StatusReport(models.Model):
2      user = models.ForeignKey(User)
3      when = models.DateTimeField(auto_now=True)
4      status = models.CharField(max_length=200)
```

Just three columns here:

1. `user` (a foreign key into the `payments_user table`)
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 I chose a straight-forward view function which is implemented in `main.view.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)
```

A brief description of the important lines:

• Line 2: only respond to POSTs

- Line 3: pull the status out of the request, which corresponds to the control with name of `status` on our form:

```html
<input id="status" type="text" class="form-control" name="status"
       placeholder="{{user.name}} what's your status?">
```

- Line 7-8: Get 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, because we have also updated the `main.view.index` function. It now looks like this:

```python
def index(request):
    uid = request.session.get('user')
    if uid is None:
        #main landing page
        market_items = MarketingItem.objects.all()
        return render_to_response('main/index.html',
                                  {'marketing_items':market_items})
    else:
        #membership page
        status = StatusReport.objects.all().order_by('-when')[:20]
        return render_to_response('main/user.html',
                                  {'user': User.get_by_id(uid),
                                   'reports':status},
                                  context_instance=RequestContext(request),
                                 )
```

The main difference from our previous version of this function is:

```python
status = StatusReport.objects.all().order_by('-when')[:20]
```

This line gets a list of status reports ordered by when they were posted in reverse order and takes the top 20. 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 querysets, which means it won't actually query the database until you access the data from the queryset, which in this case is the point when you do the slicing, `[:20]`. Thus we will only pull at most 20 rows from the database.

> **NOTE**: Ensure Django ORM is doing what you expect! We can prove that we are only pulling up to twenty results by turning on logging and running the query. The simplest 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  >>> import logging
3  >>> logger = logging.getLogger('django.db.backends')
4  >>> logger.setLevel(logging.DEBUG)
5  >>> logger.addHandler(logging.StreamHandler())
6  >>> StatusReport.objects.all().order_by('-when')[:20]
7  (0.002) SELECT "main_status_reports"."id", "main_status_reports"."user_id",
       "main_status_reports"."when",
8                  "main_status_reports"."status" FROM "main_status_reports"
9                   ORDER BY "main_status_reports"."when"
10                  DESC LIMIT 20; args=()
11 [<StatusReport: StatusReport object>, <StatusReport: StatusReport object>,
12 <StatusReport: StatusReport object>, <StatusReport: StatusReport object>,
13 <StatusReport: StatusReport object>, <StatusReport: StatusReport object>,
14 <StatusReport: StatusReport object>]
```

> **NOTE**: As you can see from line 7, the exact SQL that Django executes is outputed for
> you to read and it does include a LIMIT 20 at the end. This by the way is a good generic
> way to quickly verify that the ORM is executing what you think it is executing. So keep
> this snippet of code handy for those times when you need to quickly verify what SQL
> Django is actually executing.

Coming back to our view function, lines 10-14:

```
1  return render_to_response('main/user.html',
2                            {'user': User.get_by_id(uid),
3                             '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 script-
ing 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. If you glance back up at the code listing for `main.views.index` you will
notice that I have already included the code to return the list of Status Reports. We just need to
display it with a template, namely `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          <img src="{% gravatar_url report.user.email 32 %}" width="32" height="32"
10    alt="{{ report.user.name }}"/>
11        </div>
12        <div class="media-body">
13        <p>{{ report.status }}</p>
14        </div>
15      </div>
16    {% endfor %}
17  </section>
```

You will notice the following.

- Line 6: Looping through the list of status reports returned from the database (20 most recent).
- Line 9: we have a new template tag `gravatar_url` that we will explain in a minute.
- Line 13: we actually display 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, 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://starwars.com//img/explore/encyclopedia/characters/anakinskywalker_relationship.png'
 4      query_params = urlencode([('s', str(size)),
 5                                ('d', default)])
 6
 7      return ('http://www.gravatar.com/avatar/' +
 8              hashlib.md5(email.lower().encode('utf-8')).hexdigest() +
 9              '?' + 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.

Exercises

1. Our User Story **US3 Main Page** says that the main page should be a place for announcements and a listing of current happenings. We have implemented user announcements, i.e. status reports, but we should also have a section for system announcements / current events. Using the architecture described in this chapter, can you create an Announcements `info_box` so we can post 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 10

REST

In the last chapter, one feature we implemented for the membership page was status updating. It works, but we can do better.

One issue with the status updating is that when you submit a status, the entire page reloads, and then 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. AJAX is a client-side technology for making asynchronous requests that don't cause an entire page refresh. The client-side of AJAX is often coupled with a server-side API that makes it easy to get at 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. :) If I could hazard a definition, I would say:

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. However, original REST APIs were generally implemented with XML, but today all the "cool/hipster kids" use JSON. JSON is extremely simple to work with in Python, since 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. So I'm going to cut through all the debate and tell you the one true way to implement it. :) No seriously, there is no such thing, but I am going to tell you how I generally do it and ignore other ways, as otherwise we might never finish this chapter!

Structuring a REST API

There are three key points to implementing a nice REST-based API.

1. 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 throug the **Member** URI (also commonly refered 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: `http://<site-name>/status_reports/`

And for the member (i.e. an individual status_report): `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., `http://<site-name>/api/status_report`

This helps to differentiate between the REST API and those URLIs that just return pages.

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:

`http://<site-name>/api/v1/status_reports/`

Similarly we could create similar URLs for all of the resources we want to make available in our API. For example users would be at `http://<site-name>/api/v1/users/` and `http://<site-name>/api/v1/users/1` (or whatever id).

2. REST is built on top of HTTP; thus, it should use the appropriate HTTP verbs. The following table 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	Return the entire collection as a list, potentially with related information
PUT	Replace the entire collection with the passed-in collection
POST	Create a new entry in the collection, generally with an auto-assigned ID
DELETE	Blow 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. In this case, the Status_report with id == 2
PUT	Replace the addressed member with the one passed in, or create a new one if it doesn't exist
POST	*USUALLY NOT USED* POST to the Collections URI instead
DELETE	Delete the member with corresponding address

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 update. Either update existing or insert new; 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. I just ignore it as I am either creating a new item with POST or updating with PUT; Honestly, I don't see the need for PATCH.

3. 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 the main concerns with 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 in some meaninful 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 implement something like:

```
GET api/v1/users/1/login
```

Consider login as a separate resource. Or better yet, call it session; then you have:

```
POST api/v1/session - login and return a session ID
DELETE api/v1/session/sessionID - log out
```

This sticks more strictly to the REST definition. Of course with REST there are only suggestion / conventions, and nothing will stop you from implementing whatever URL structure you wish.

For example, I have seen APIs created completely with POST requests to get around cross-site scripting limitations (but of course, you can also get around those restrictions with a properly configured NGINX proxy). The point is, do your best to stick with the conventions, but if you have truly compelling reasons not to, by all means 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, and when, we need them.

Following the design considerations discussed above, we will use the following url structure:

Queries

- GET - api/v1/status_reports/ - returns a particular report status by id
- GET - api/v1/status_reports - return ALL status_reports

Creates

- POST = api/v1/status_reports - (pass in the appropriate POST data) will create a status update and return the id

That's all we need for now. You can see that we could simply add a number of 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, so we won't add them.

Django REST framework

Before jumping into another framework, 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 do it in a few lines of code:

```python
from django.http import HttpResponse
from django.core import serializers

def report(request):
    if request.method = "GET":
        status = StatusReport.objects.all().order_by('-when')[:20]

        return HttpResponse(
            serializers.serialize("json", status),
            content_type='application/json',
            context_instance=RequestContext(request)
        )
```

The only difference from before is that we are returning the context as JSON as opposed to our usual HTML template. That is what the `django.core.serializers` do. 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 stuff 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. But I changed it a bit to use serializers:

```python
from django.core import serializers
from django.http import HttpResponse

class JSONResponseMixin(object):
    """
    A mixin that can be used to render a JSON response.
    """
    def render_to_json_response(self, context, **response_kwargs):
        """
        Returns a JSON response, transforming 'context' to make the payload.
        """
        return HttpResponse(
            serializers.serialize("json",context),
            content_type='application/json',
            **response_kwargs
        )
```

While this is a bit better, it still isn't going to help a whole lot with authentication, API discovery, handling POST parameters, etc. etc... To that end, we are going to use a framework to help us out with REST. The two most popular ones as of this writing are Django REST Framework and Tastypie.

169

Which is better? Django REST Framework has a cooler logo, and that's enough for me... Really, they are about the same in terms of functionality, and it's up to personal preference.

If your still unsure you can look at the popularity of the package, how active it is and when was it's last release date to help you decide. There are two great placed to find this information.

- Django Packages - allows you to search for a package and shows information about the last PyPi release date, number of repo forks and number of watchers. All good information when deciding if you should choose a particular project.
- Github - a huge number of projects are hosted on Github these days. If you look at the github page for a projetct you see the number of watchers, the number of people who stared the project and the number of people that forked the project (all are a good judge of popularity). You can also see informaiton like number of releases, number of contibutors and there is also an issues page that is worth looking through to see if there are any reported show stopper issues with the project.

Using this informiaton, especially when you are unfamiliar with the project can greatly aid in the decision making process of which project to use.

Let's jump right into the meat and potatoes.

Django REST Framework (DRF) Installation

The first thing to do is install DRF. Pip will do:

```
pip install djangorestframework
```

Also, let's update our *requirements.txt* file to include our new dependency:

```
Django
mock
requests
stripe
sycopg2
djangorestframework
```

DRF provides a number of tools you can use. We will 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:

```
from django.forms import widgets
from rest_framework import serializers
from main.models import StatusReport
```

```
4
5
6    class StatusReportSerializer(serializers.Serializer):
7        pk = serializers.Field()
8        user = serializers.RelatedField(many=False)
9        when = serializers.DateTimeField()
10       status = serializers.CharField(max_length=200)
11
12       def restore_object(self, attrs, instance=None):
13           """create or update a new StatusReport instance"""
14
15           if instance:
16               instance.user = attrs.get('user', instance.user)
17               instance.when = attrs.get('when', instance.when)
18               instance.status = attrs.get('status', instance.status)
19               return instance
20
21           return StatusReport(**attrs)
```

If you have worked with Django Forms before (which we already touched), 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 a pk (primary key) field as type `serializers.Field()`. This is a read-only field that cannot be changed, but it needs to be there; it maps to our id field.
2. Our user field is of type `serializers.RelatedField(many=False)`. This represents a many-to-one relationship and says we should serialize the user object by using its `__str__` function. In our case this is the user's email address.

There is one function, `restore_object`. This function is responsible for converting back into a `StatusReport` object. So any custom creation logic you may need can be put in here. By "creation logic" I'm not referring to the code that would normally be put into your `__init__` function, because that is already going to be called, but any logic you may need to include when creating the object from a deserialized JSON string.

For a single object serializer like the one shown here, 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

OK, 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).

The DRF serializers actually work in two steps:

- **Step 1**: Convert the object into a dictionary.
- **Step 2**: Convert the dictionary into JSON.

Let's test that those two steps work as we expect. Create a file called *../tests/main/testSerializers.py* and include the following:

```python
from django.test import TestCase
import unittest
from main.models import StatusReport
from payments.models import User
from main.serializers import StatusReportSerializer
from rest_framework.renderers import JSONRenderer
from collections import OrderedDict


class StatusReportSerializer_Tests(TestCase):

    @classmethod
    def setUpClass(cls):
        cls.u = User(name="test", email="test@test.com")
        cls.u.save()

        cls.new_status = StatusReport(user=cls.u, status="hello world")
        cls.new_status.save()

    @classmethod
    def tearDownClass(cls):
        cls.u.delete()
        cls.new_status.delete()

    def test_model_to_dictionary(self):
        serializer = StatusReportSerializer(self.new_status)

        expected_dict = {'pk': self.new_status.id,
                         'user':'test@test.com',
                         'when':self.new_status.when,
                         'status': 'hello world',
                         }
        self.assertEquals(expected_dict, serializer.data)
```

```
34
35      def test_dictionary_to_json(self):
36          serializer = StatusReportSerializer(self.new_status)
37          content = JSONRenderer().render(serializer.data)
38
39          expected_dict = OrderedDict([('pk', self.new_status.id),
40                              ('user','test@test.com'),
41                              ('when', self.new_status.when),
42                              ('status', 'hello world'),
43                                          ])
44
45          expected_json = JSONRenderer().render(expected_dict)
46
47          self.assertEquals(expected_json, content)
```

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   import unittest
3   from main.models import StatusReport
4   from payments.models import User
5   from main.serializers import StatusReportSerializer
6   from rest_framework.renderers import JSONRenderer
7   from collections import OrderedDict
8
9
10  class StatusReportSerializer_Tests(TestCase):
11
12      @classmethod
13      def setUpClass(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 world")
18          cls.new_status.save()
19
20      @classmethod
21      def tearDownClass(cls):
22          cls.u.delete()
23          cls.new_status.delete()
```

-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)
```

173

```
3
4       expected_dict = {'pk': self.new_status.id,
5                        'user':'test@test.com',
6                        'when':self.new_status.when,
7                        'status': 'hello world',
8                       }
9       self.assertEquals(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. Gravy!

The next step in the object to JSON conversion process is converting the dictionary to JSON. So we test that here:

```
1   def test_dictionary_to_json(self):
2       serializer = StatusReportSerializer(self.new_status)
3       content = JSONRenderer().render(serializer.data)
4
5       expected_dict = OrderedDict([('pk', self.new_status.id),
6                        ('user','test@test.com'),
7                        ('when', self.new_status.when),
8                        ('status', 'hello world'),
9                                                ])
10
11      expected_json = JSONRenderer().render(expected_dict)
12
13      self.assertEquals(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 draw back there 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, and the option I chose here, was 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 the `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.

So serialization seems to work correctly. How about deserializaiton? We need to run the opposite test; let's see what we get:

```
1  def test_json_to_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      self.assertEqual(self.new_status.status, serializer.object.status)
9      self.assertEqual(self.new_status.when, serializer.object.when)
```

Run that. Boom. Failure. :(If you look closely at the output, you'll get something like this:

```
1  self.assertEqual(self.new_status.when, serializer.object.when)
2  AssertionError: datetime.datetime(2014, 3, 17, 23, 11, 55, 510634) !=
       datetime.datetime(2014, 3, 17, 23, 11, 55, 510000)
```

Of course your dates will be different. But why are the two not equal? Welcome to the wonderful world of date format. :) JavaScript or actually ECMA script (as it is properly called) defines a format for a datetime string to be down to millisecond precision. However, Python stores dates with microsecond precision. Hence the two don't match and our test fails.

What do we do? We have two options. The first is to just change the test to compare the second representation of the date time fields (because for our purposes, that's close enough). Change the last line of the test to:

```
1  self.assertEqual(self.new_status.when.strftime("%Y-%m-%d %H:%M:%S"),
2                   serializer.object.when.strftime("%Y-%m-%d %H:%M:%S"))
```

Alternatively if we do want them to really be the same, we can change our model so that it actually stores with second precision instead of microsecond precision. A naive implementation might look like:

```
1  def _getNowNoMicroseconds():
2      """We want to get time without microseconds so it
3      converts to JavaScript time correctly"""
4      t = datetime.now()
5      return datetime(t.year, t.month, t.day, t.hour,
6                      t.minute, t.second, 0, t.tzinfo)
7
8  class StatusReport(models.Model):
9
10     user = models.ForeignKey(User)
11     when = models.DateTimeField(default=_getNowNoMicroseconds())
12     status = models.CharField(max_length=200)
```

All we are doing in the above code is getting `datetime.now()` and setting the microseconds to 0.

The problem with that approach is that our REST implementation built on top of it will want us to ALWAYS pass in a date for `when` - because the Django validation will read that as `null=False`. But from a usability standpoint, it's probably better if we don't have to pass in the date for `when` and just have the server default it for us. Thus a slightly better way to do this would look like:

```python
class StatusReport(models.Model):

    user = models.ForeignKey(User)
    when = models.DateTimeField(blank=True)
    status = models.CharField(max_length=200)

    def save(self, *args, **kwargs):
        if self.when is None:
            self.when = self._getNowNoMicroseconds()
        super(StatusReport, self).save(*args, **kwargs)

    def _getNowNoMicroseconds(self):
        """We want to get time without microseconds so it
        converts to JavaScript time correctly"""
        t = datetime.now()
        return datetime(t.year, t.month,t.day,t.hour,
                        t.minute,t.second,0,t.tzinfo)
```

On line 4 we set `when` to have `blank=True` which just tells Django validation to allow nulls; that way, we don't get any validation errors. Then we overwrite the save function so if `when` isn't passed in, we "default" it to the current time with no microseconds. It's a subtle difference but gives us a nice usability gain in our REST API.

Since we are on the topic of dates and date formats, you can control the date format that the serializer outputs by doing something like below:

```python
class StatusReportSerializer(serializers.Serializer):
    pk = serializers.Field()
    user = serializers.RelatedField(many=False)
    when = serializers.DateTimeField(format='%Y-%m-%d -- %H:%M')
    status = serializers.CharField(max_length=200)
```

This will cause a string of the specified datetime format to be inserted into `serializer.data` instead of the default `datetime` object. This can be useful if you want to generate custom datetime strings instead of sticking to the standard.

See all the fun things we learn from testing?

Ultimately it's up to you to choose which one solution for the datetime issue you think is best for your particular use-case. But at least now you know if you need microsecond precision, and you want to export using JSON, that you're going to have your work cut out for you!

OK, let's finish the test. There is one more assert I want to add. We need to make sure we got our related user object back correctly:

```
1   def test_json_to_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       self.assertEqual(self.new_status.status, serializer.object.status)
9       self.assertEqual(self.new_status.when, serializer.object.when)
10      self.assertEqual(self.new_status.user, serializer.object.user)
```

You probably already guessed it, but this test is going to fail as well…

```
1   Traceback (most recent call last):
2     File
        "/Users/j1z0/Code/RealPython/mvp_for_Adv_Python_Web_Book/tests/main/testSerializers.py",
        line 59, in test_json_to_StatusReport
3        self.assertEqual(self.new_status.user, serializer.object.user)
4     File
        "/Users/j1z0/.virtualenvs/py3/lib/python3.3/site-packages/django/db/models/fields/related.p
        line 324, in __get__
5        "%s has no %s." % (self.field.model.__name__, self.field.name))
6   payments.models.DoesNotExist: StatusReport has no user.
```

The important part is the `payments.models.DoesNotExist`; 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.RelatedField(many=False)
```

This says, serialize the user field by calling its `__str__` function, which just returns an email. Then when we deserialize the object, our `restore_object` function is called, which looks like this:

```
1   def restore_object(self, attrs, instance=None):
2       """Create or update a new StatusReport instance"""
3
4       if instance:
5           instance.user = attrs.get('user', instance.user)
6           instance.when = attrs.get('when', instance.when)
7           instance.status = attrs.get('status', instance.status)
8           return instance
9
10      return StatusReport(**attrs)
```

This means our user is never getting passed to the restore_object function, because DRF knows that it can't create an object from a string. We'll come back to the solution to this in just a second. First,

let's talk about `ModelSerializers`.

ModelSerializers

Our initial `StatusReportSerializer` contains a ton of boilerplate coce. We just keep copying the field from our model and that kinda sucks. But fear not; there is a better way. Enter ModelSerializers. If we rewrite our `StatusReportSerializer` using DRF's `ModelSerializer`, it looks like this:

```
class StatusReportSerializer(serializers.ModelSerializer):

    class Meta:
        model = StatusReport
        fields = ('id', 'user', 'when', 'status')
```

Wow! That's seriously 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:

```
cls.expected_dict = OrderedDict([('id', cls.new_status.id),
                ('user',cls.u.id),
                ('when', cls.new_status.when),
                ('status', 'hello world'),
                                    ])
```

With that, all of our tests pass. It's not quite what we want, though. 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 while still allowing the deserialization to work? We create a custom relationship field:

```
from payments.models import User

class RelatedUserField(serializers.RelatedField):

    read_only = False

    def from_native(self, data):
        return User.objects.get(email=data)
```

- **Line 3**: we inherit from the `serializers.RelatedField` which serializes our User field by

178

calling the `__str__` function; this gives us the email address that we want, so we just implement the deserialization part.

- **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 `from_native(self, data)` will pass in the JSON value from the field, and it's our job to return the object we want. We do this by looking it up in the users table.

Now if we change our `cls.expected_dict` in `../test/main/testSerializers.py` to expect the email as it did before then all of our tests will pass. Perfect. There are several other ways to manage / display relationships in DRF; for more information on these, check the docs.

Now Serving JSON

Since we spent all that time getting our serializer to work just the way we wanted it, let's make a view for it and get our REST API up and running. Let's 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 returns JSON and one returns HTML, but that the API views (which return JSON) are defining the REST framework for our application - i.e., how we want other clients / 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 backend. *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", that is 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 API from your web app.

Our new view function in *main/json_views.py* should look something like:

```python
from rest_framework import status
from rest_framework.decorators import api_view
from rest_framework.response import Response
from main.serializers import StatusReportSerializer
from main.models import StatusReport


@api_view(['GET','POST'])
def status_collection(request):
    """Get the collection of all status_reports
    or create a new one"""

    if request.method == 'GET':
        status_report = StatusReport.objects.all()
        serializer = StatusReportSerializer(status_report, many=True)
        return Response(serializer.data)
    elif request.method == 'POST':
        serializer = StatusReportSerializer(data=request.DATA)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.erros, status=status.HTTP_400_BAD_REQUEST)
```

- **Line 1-5**: import what we need

- **Line 7**: `@api_view` is a decorator provided by DRF; it provides us with several things, some of which are:

  - checking the appropriate `Request` is passed into the view function
  - adding context to the `Response` so we can deal with stuff like CSRF tokens
  - providing authentication functionality (which we will discuss later)
  - handling `ParseErrors`

- **Line 7**: the arguments to the `@api_view` are a list of the HTTP verbs to support. In this case, `GET` and `POST`.

- **Line 12-15**: a `GET` request on the collection view should return the `who` list, so we grab the list, serialize to JSON and return it

- **Line 15**: DRF also provides the Response object. This object inherits from `django.template.response.Simp` It takes in unrendered content (for example, JSON) and renders it based upon a `Content-Type` specified in the `request.header`

- **Line 16-19**: For the `POST` request, just create a new object based upon passed-in data.

- **Line 16**: Notice the user of `request.DATA` DRF provides the `Request` class that extends Django's `HTTPRequest` and provides a few. enhancements. The main one is `request.DATA`, which works similar to `HTTPRequest.POST` but handles `POST`, `PUT` and `PATCH` methods.

- **Line 20**: On successfully saving, return a response with HTTP return code of 201, which means created. Notice the use of `status.HHTP_201_CREATED`. You could simply put in 201, but using the DRF status identifiers, it 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 21**: 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 will flesh out the member URL below:

```python
@api_view(['GET', 'PUT', 'DELETE'])
def status_member(request, id):
    """Get, update or delete a status_report instance"""

    try:
        status_report = StatusReport.objects.get(id=id)
    except StatusReport.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)
```

```
 9
10
11     if request.method == 'GET':
12         serializer = StatusReportSerializer(status_report)
13         return Response(serializer.data)
14     elif request.method == 'PUT':
15         serializer = StatusReportSerializer(status_report, data=request.DATA)
16         if serializer.is_valid():
17             serializer.save()
18             return Response(serializer.data)
19         return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
20     elif request.method == 'DELETE':
21         status_report.delete()
22         return Response(status=status.HTTP_204_NO_CONTENT)
```

This is nearly the same as the collection class, but we support different HTTP verbs and we are dealing with one object instead of an entire collection of objects. With that, we now have the entire API for our StatusReport Now were getting somewhere.

Let's not forget to test it. First the GET functionality:

```
 1  from django.test import TestCase
 2  from main.models import StatusReport
 3  from main.serializers import StatusReportSerializer
 4  from main.json_views import status_collection
 5  from payments.models import User
 6  from django.core import serializers
 7
 8
 9  class dummyRequest(object):
10
11      def __init__(self, method):
12          self.method = method
13          self.encoding = 'utf8'
14          self.user = "root"
15          self.QUERY_PARAMS = {}
16          self.META = {}
17
18
19  class JsonViewTests(TestCase):
20
21      def test_get_collection(self):
22          status = StatusReport.objects.all()
23          expected_json = StatusReportSerializer(status, many=True).data
24          response = status_collection(dummyRequest('GET'))
25
26          self.assertEqual(expected_json,response.data)
```

Above we create a dummyRequest that has the information that DRF expects. (We can't use the `RequestFactory` yet because we haven't setup the URLs.). Then in our `JsonViewTest` 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. Normally in our views we are calling `render_to_response`, which does the rendering, but not in this view.

Otherwise the test is the same as any other view test. To be complete, we should check the case 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, faithful reader.

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 our 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:

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

urlpatterns = patterns('main.json_views',
    url(r'^status_reports/$', 'status_collection'),
    url(r'^status_reports/(?P<id>[0-9]+)$', 'status_member'),
    )
```

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:

```
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:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'main',
    'django.contrib.admin',
    'django.contrib.flatpages',
    'contact',
    'payments',
    'rest_framework',
```

```
14  )
```

Now if your start the development server and navigate to `http://127.0.0.1:8000/api/v1/StatusReport` you should see something like:
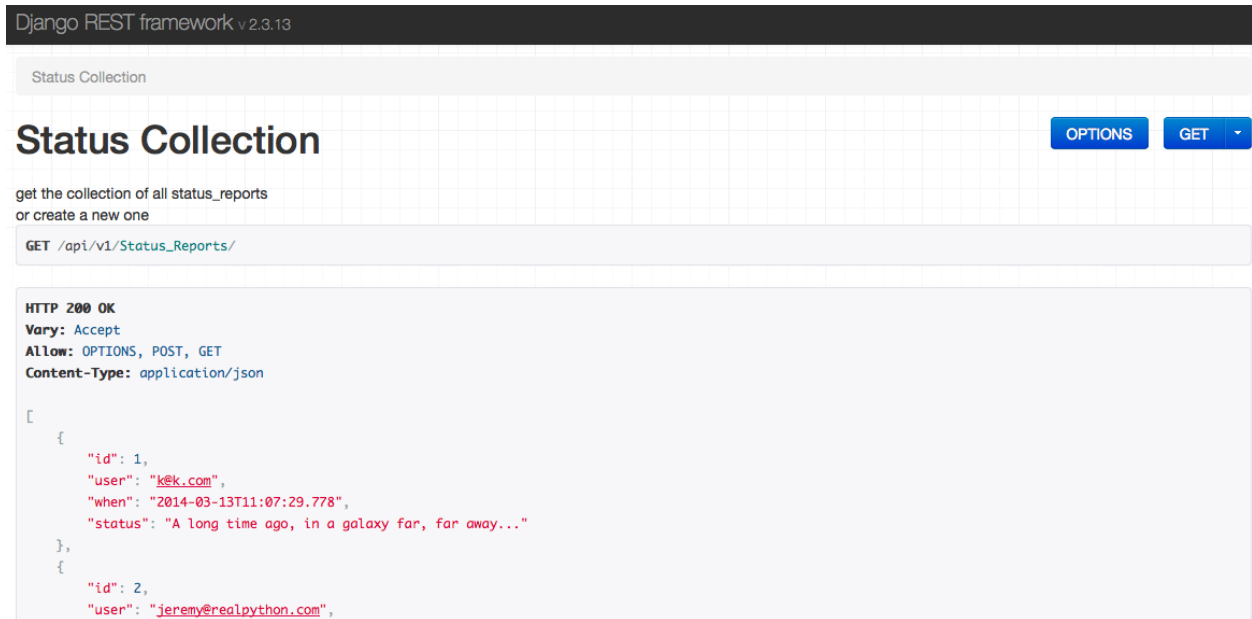

Figure 10.1: DRF browsable API for Status Collection

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 give 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 is the guy that is giving 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:

> **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 approaprite download for you system.

```
1  $ curl http://127.0.0.1:8000/api/v1/StatusReports/
```

Or, to be more explicit:

```
1  $ curl http://127.0.0.1:8000/api/v1/StatusReports/ -H `Accept: application/json`
```

And you will get back raw JSON. Thus returning JSON is the default action the `Response` template will take. However, by default your browser will set the `Accept` header to `text/html`, which you can also do from curl like this:

```
1  $ curl http://127.0.0.1:800/api/v1/StatusReports/ -h `Accept: text/html`
```

And then you'll get back a whole mess of HTML. Hats off to the DRF folks; that is 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.

Start with the `status_collection`; we could just create a class that inherits from `APIView` which is the same as using the `@api_view` on our function, and then create a function for GET and a function for POST. That would make it a class-based view but it wouldn't really save us much in terms of typing or provide much reuse. So let's take it one step further.

By using some of the mixins that the DRF provides, we can get away with a lot less code:

```python
from main.serializers import StatusReportSerializer
from main.models import StatusReport
from rest_framework import mixins
from rest_framework import generics


class StatusCollection(mixins.ListModelMixin,
                       mixins.CreateModelMixin,
                       generics.GenericAPIView):

    queryset = StatusReport.objects.all()
    serializer_class = StatusReportSerializer

    def get(self, request):
        return self.list(request)

    def post(self, request):
        return self.create(request)
```

Where did all the code go, you may ask? That's what the mixins are for:

- **Line 6**: `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 - i.e., creating a new object of the collection type.
- **Line 8**: `generics.GenericAPIView` - this mixin provides the "core" functionality plus the Browsable API we talked about in the previous section.
- **Line 9**: defining a class-level queryset member is required so the `ListModelMixin` can works its magic.
- **Line 10**: defining a class-level `serializer_class` member is 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 request vs a POST request without having a number of if statements, so there is better separation of concerns.

Want to write even less code? Notice that we are calling list and create with no special parameters, so we can take it one step further and write the class like this:

```
class StatusCollection(generics.ListCreateAPIView):
    queryset = StatusReport.objects.all()
    serializer_class = StatusReportSerializer
```

Yup, that's it; three lines of code. Pretty concise and to the point. I understand why the DRF folks implemented this, as it makes it dead-simple to create the "correct" REST API with a very minimal amount of work.

> **NOTE:** Personally though, I wish 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 / understand 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, I prefer code that is more explicit like the previous example even though it's more code. However, there is nothing preventing you from putting in a nice docstring to explain what the function does, and that's 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:

```
class StatusMember(mixins.RetrieveModelMixin,
                   mixins.UpdateModelMixin,
                   mixins.DestroyModelMixin,
                   generics.GenericAPIView):

    queryset = StatusReport.objects.all()
    serializer_class = StatusReportSerializer

    def get(self, request, *args, **kwargs):
        return self.retrieve(request, *args, **kwargs)

    def put(self, request, *args, **kwargs):
        return self.update(request, *args, **kwargs)

    def delete(self, request, *args, **kwargs):
        return self.destroy(request, *args, **kwargs)
```

Or in its more simplified form:

```
class StatusMember((generics.RetrieveUpdateDestroyAPIView):
    queryset = StatusReport.objects.all()
```

```
3        serializer_class = StatusReportSerializer
```

We also need to change our *main/urls.py* function slightly to account for the class-based views:

```
1  urlpatterns = patterns('main.json_views',
2      url(r'^status_reports/$', json_views.StatusCollection.as_view()),
3      url(r'^status_reports/(?P<pk>[0-9]+)/$', json_views.StatusMember.as_view())
4      )
```

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 would need to modify our test a little bit to account for the class-based views:

```
1  def test_get_collection(self):
2      status = StatusReport.objects.all()
3      expected_json = StatusReportSerializer(status, many=True).data
4      response = StatusCollection.as_view()(dummyRequest("GET"))
5
6      self.assertEqual(expected_json,response.data)
```

Notice in line 4 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.

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:

- Only authenticated users can post status reports
- Only the creator of a status report can update / delete that status report
- Unauthenticated requests should be denied access

Starting from the last listed permission, we can use DRF's built-in permissions to take care of that: `rest_framework.permissions.IsAuthenticated`. All we need to do is add that class as a property of our class views (i.e., `StatusCollection` and `StatusMember`) by adding the property like so:

```
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:

```
from rest_framework import permissions
```

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. Let's give them an authenticated user:

```python
class dummyRequest(object):

    class dummyUser(object):

        is_authed = True

        def is_authenticated(self):
            return self.is_authed

    def __init__(self, method):
        self.method = method
        self.encoding = 'utf8'
        self.user = self.dummyUser()
        self.QUERY_PARAMS = {}
        self.META = {}
```

You'll notice that we added a `dummyUser` class (lines 3-8) as an internal class of our dummyRequest and then set the requests user to our dummyUser. This dummyUser has only one function, `is_authenticated`, which is the function that Django and DRF use to check if a user is indeed logged

in. We set `is_authenticated` by default to return True, but also provide an `is_authenticated` attribute so we can overwrite the default value in a test if we need to.

Now our `test_get_collection` will pass. We can also put in a `test_get_collection_requires_logged_in_user` test to verify our authentication is working correctly.

```python
#we need to add this import up top
from rest_framework import status

def test_get_collection_requires_logged_in_user(self):

    anon_request = dummyRequest("GET", authed=False)
    response = StatusCollection.as_view()(anon_request)

    self.assertEqual(response.status_code, status.HTTP_403_FORBIDDEN)
```

- **Line 2**: we will 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 6**: we pass in `authed=False` to our dummyRequest, which sets the user to be unauthorized (updated code for `dummyRequst` below)
- **Line 9**: verify that we return `status.HTTP_403_FORBIDDEN`, which is the correct HTTP status code to return in the case of unauthorized access

For this to all work, we need to update our mock to be able to take in a parameter specifying if the user should be authenticated or not. The updated `__init__` is here:

```python
def __init__(self, method, authed=True):
    self.method = method
    self.encoding = 'utf8'
    self.user = self.dummyUser()
    self.user.is_authed = authed
    self.successful_authenticator = True
    self.QUERY_PARAMS = {}
    self.META = {}
```

Notice line 6 - `self.successful_authenticator = True`. We use this because DRF checks this attribute to decide which type of error to throw when the permission check fails.

Now we have our permission setup to ensure unauthorized users get an access denied. This also takes care of our first authorization requirement, which was, *Only authenticated users can post status reports*. Now for our final requirement, which was: *Only the creator of a status report can update / delete that status report*.

To implement this type of permission, we will have to create our own custom permission class. Let's first create a *main/permissions.py* file and then add the following code:

```python
from rest_framework import permissions
```

```
 2
 3  class IsOwnerOrReadOnly(permissions.BasePermission):
 4
 5      def has_object_permissions(self, request, view, obj):
 6
 7          #Allow all read type requests
 8          if request.method in ('GET', 'HEAD', 'OPTIONS'):
 9              return True
10
11          #this leaves us with write requests (i.e. POST / PUT / DELETE)
12          return obj.user == request.user
```

Once the class is created, we need to update our `StatusMember` class like so:

```
 1  #import me at the top of the module
 2  from main.permissions import IsOwnerOrReadyOnly
 3
 4  #add me as a attribute of StatusMember class
 5  permission_classes = (permissions.IsAuthenticated, IsOwnerOrReadOnly)
```

And that's it. Now only owners can update / delete their status reports.

It is important to note that all of this authentication we have been doing is using the default Authentication classes which are `SessionAuthentication` and `BasicAuthentication`. Our unit test above took advantage of the `SessionAuthentication`, which is usually handled by your browser. As such 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 docs do 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 quite get to the solution yet. But don't worry; we are almost there (and we will get to it next chapter). But we did go over one of the key ingredients to making that no-refresh happen, and that is a good REST API. REST is increasingly popular because of its simplicity to consume resource 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 / deserialize our data, how to structure our views appropriately, 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, I generally prefer not to use them because of the same explicit / implicit tradeoff we talked about with the 'ListCreateAPIView' class. 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. I've tried to cover 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, I will say this: **REST is everywhere on the web today, and it's here to stay. If you're going to do much web development, you will surely have to work with REST APIs, so make sure you understand that concepts presented in this chapter well. **

Exercises

1. Flesh out the unit tests. In the `JsonViewTests`, check the case where there is no data to return at all, and test the POST with and without valid data.
2. Just for practice 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, and it wouldn't do to have our browsable API not look like the rest of our site, update the browsable API Template to fit with our overall site template.
4. Speaking of API, we don't have permissions on the browsable API; put them in.

Chapter 11

Appendix A - Solutions to Exercises

Chapter 2

Exercise 1

Our URL routing testing example only tested one route. Write test to test the other routes. Where would you put the test to verify the `pages/` route? Do you need to / want to do anything special to test the admin routes?

The solution for exercise 1 can be found in the repo.

```
1   git checkout Ch2_ex1
```

There are of course, many solutions but I decided to create a `ViewTesterMixin` class to help with the view tests. It looks like this:

```
1   class ViewTesterMixin(object):
2
3       @classmethod
4       def setupViewTester(cls, url, view_func, template,
5                                    status_code = 200,
6                                    session={}):
7           from django.test import RequestFactory
8           request_factory = RequestFactory()
9           cls.request = request_factory.get(url)
10          cls.request.session = session
11          cls.status_code = status_code
12          cls.url = url
13          cls.view_func = staticmethod(view_func)
14          cls.template = template
15
16      def test_resolves_to_correct_view(self):
17          test_view = resolve(self.url)
```

```
18          self.assertEquals(test_view.func, self.view_func)
19
20      def test_returns_appropriate_respose_code(self):
21          resp = self.view_func(self.request)
22          self.assertEquals(resp.status_code, self.status_code)
23
24      def test_uses_appropriate_template(self):
25          resp = self.view_func(self.request)
26          self.assertTemplateUsed(resp, self.template)
```

And it's meant to be used like this:

```
1   class SignInPageTests(TestCase, ViewTesterMixin):
2
3       @classmethod
4       def setUpClass(cls):
5           ViewTesterMixin.setupViewTester('/sign_in',
6                                           sign_in,
7                                           "sign_in.html")
```

The `ViewTesterMixin` creates some default tests that can be run against most standard tests. In our case I use it for all the view functions in the payments app. These are the same functions that we implemented in Chapter 2 to test our view function, but I've now refactor the functions and placed them in a base class. All you have to do is call the `setupViewTest` class method, which is meant to be called from the derived classes `setUpClass` class method. Once you have got the setup taken care of, the `ViewTesterMixin` will do the basic testing for routing, return codes, and making sure you're using the appropriate template.

Then you can implement any other tests you would like. The thing I like about using Mixins for testing is that they can reduce a lot of the boiler plate / common test routines that you often run against Django's system. And because I'm an especially lazy developer, anything that saves me a ton of type is good by me!

As for the second part of the question. I wouldn't bother testing the admin views as they are generated by Django, so we can assume they are correct / 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 I'm going to leave this out until we get to Chapter 4 – Upgrading to Django 1.6, and then you can see how the improved test discovery in Django 1.6 will allow you to better structure your tests to model the application under test.

Exercise 2

Write a simple test to verify the functionality of the `sign_out` view. Do you recall how to handle the session?

I've actually included the solution to exercise 2 in the answer for exercise 1. Can you spot it?

Have a look at the SignOutPageTests class:

```
1    class SignOutPageTests(TestCase, ViewTesterMixin):
2
3        @classmethod
4        def setUpClass(cls):
5            ViewTesterMixin.setupViewTester('/sign_out',
6                                            sign_out,
7                                            "index.html",
8                                            status_code=302,
9                                            session={"user":"dummy"},
10                                           )
11
12       def setUp(self):
13           # sign_out clears the session, so let's reset it every time
14           self.request.session = {"user": "dummy"}
```

There are two things to notice here. One, we are passing in session={"user": "dummy"} to our setupViewTest function. And if we look at the third and fourth lines of that function:

```
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 **Mocs, Fakes, Test Doubles, Stubs...** section of Chapter 2.

Exercise 3

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 Model testing section of Chapter 2, 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 timestamp.

You can get the code here:

```
1    git checkout Ch2_ex3
```

Or you can see the code below. 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. Here is the code:

```
1    class UserModelTest(TestCase):
2
```

```
3      @classmethod
4      def setUpClass(cls):
5          ContactForm(email="test@dummy.com", name="test").save()
6          ContactForm(email="j@j.com", name="jj").save()
7          cls.firstUser=ContactForm(email="first@first.com", name="first",
8                  timestamp=datetime.today() + timedelta(days=2))
9          cls.firstUser.save()
10         #cls.test_user = User(email="j@j.com", name="test user")
11         #cls.test_user.save()
12
13     def test_contactform_str_returns_email(self):
14        self.assertEquals("first@first.com", str(self.firstUser))
15
16     def test_ordering(self):
17         contacts = ContactForm.objects.all()
18         self.assertTrue(contacts.ordered)
19         self.assertEquals(self.firstUser, contacts[0])
```

Exercise 4

The QA teams I have worked with have been particularly keen on 'boundary checking'. Write some unit tests for the CardForm that ensure boundary checking is working correctly.

To accomplish this 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          {'data': {'last_4_digits' : '123'},
4           'error' : ('last_4_digits', [u'Ensure this value has at least 4
   characters (it has 3).'])},
5          {'data': {'last_4_digits' : '12345'},
6           'error' : ('last_4_digits', [u'Ensure this value has at most 4
   characters (it has 5).'])}
7      ]
8
9      for invalid_data in invalid_data_list:
10         self.assertFormError(CardForm,
11                 invalid_data['error'][0],
12                 invalid_data['error'][1],
13                 invalid_data["data"])
```

Chapter 3

Exercise 1

If you really want to get your head around mocks, have a go at mocking out the `test_registering_user_twice_cause_error_msg` **test. Start by mocking out the** `User.create` **function and have it throw the appropriate errors. HINT [the docs](#) are a great resource. In particular, have a search for** `side_effect`**. For extra credit, mock out the UserForm as well.**

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.

The extra credit part, **For extra credit mock out the UserForm as well**, is a bit more involved. There are many ways this can be done with the mock library, but I think 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 {'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          def addError(self, error):
```

```
20                    pass
21
22            return MockUserForm()
```

A few quick points:

1. I wrapped the class in a function so it's easy to reach.
2. I only created the methods / properties that would be called by this test.
3. I only need the methods to do the minimal amount of work necessary to get the test to behave as I would like it to.

Once I have the Mock class created, I can override the "real" `UserForm` class with my 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', side_effect=IntegrityError)
3    def test_registering_user_twice_cause_error_msg(self, save_mock, view_mock):
4
5        # create the request used to test the view
6        self.request.session = {}
7        self.request.method = 'POST'
8        self.request.POST = {}
9
10       # create the expected HTML
11       html = render_to_response('register.html',
12           {
13               'form': self.get_MockUserForm(),
14               'months': range(1, 12),
15               'publishable': settings.STRIPE_PUBLISHABLE,
16               'soon': soon(),
17               'user': None,
18               'years': range(2011, 2036),
19           })
20
21       # mock out Stripe so we don't hit their server
22       with mock.patch('stripe.Customer') as stripe_mock:
23           config = {'create.return_value': mock.Mock()}
24           stripe_mock.configure_mock(**config)
25
26           # run the test
27           resp = register(self.request)
28
29           # verify that we did things correctly
30           self.assertEquals(resp.content, html.content)
31           self.assertEquals(resp.status_code, 200)
32           self.assertEquals(self.request.session, {})
33
```

```
34          # assert there are no records in the database
35          users = User.objects.filter(email="python@rocks.com")
36          self.assertEquals(len(users), 0)
```

Notice what object I patched: `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 I'm testing view.registration, so I want the `UserForm` imported in the `view` module. If you're still unclear about this, re-read this section of the docs. I personally read it at least ten times when I got started with mocks. :)

Exercise 2

* As alluded to in the conclusion, pulling out the customer creation logic from register into a separate CustomerManager class may be a good idea. Re-read the first paragraph of the conclusion and do that. Don't forget to update the tests accordingly.*

The solution for this is pretty straightforward; grab all the Stripe stuff and wrap it in a class. I admit 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 really this is about TDD and how it helps you with refactoring. To make this change, the first thing I would do is create a simple test to help me 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': 'test@test.com',
6                           'card': '4242', 'plan': 'gold'}
7              cust = Customer.create(**cust_data)
8
9              create_mock.assert_called_with(**cust_data)
```

This test says that I want my `Customer.create` function to call Stripe with the arguments I pass in. I could implement a simple solution to that like this:

```
1  class Customer(object):
2
3      @classmethod
4      def create(cls, **kwargs):
5          return stripe.Customer.create(**kwargs)
```

That's probably the simplest way to do it. (Note: I just used **kwargs, but you could enumerate the names.) This will pass the test, but the next requirement is to support both subscription and one_time payments. Let's update the tests a little bit:

```
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': 'test@test.com',
6                         'card': '4242','plan': 'gold'}
7            cust = Customer.create("subscription", **cust_data)
8
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           cust = Customer.create("one_time", **cust_data)
18
19           charge_mock.assert_called_with(**cust_data)
```

I could have created two separate functions, but I decided to just design it with one function and pass in the type of billing as the first argument. I can make both of these test pass with a slight modification to my 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 thing would be to actually substitue our new `Customer` class in place of `stripe.Customer` in our `register` function, and then we could update our tests cases to not reference Stripe at all. Here is 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_succesfully(self, create_mock,
     stripe_mock):
```

Not much different. We changed the first patch to `payments.views.Customer.create` instead of `stripe.Customer.create` In fact, if we would have left it as is, the test would have stilled 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 or change how we interact with Stripe.

And of course you could continue and factor out the Stripe stuff from the edit function as well.

You can get the full code here:

```
git checkout Ch2_ex2
```

Chapter 8

Exercise 1

Bootstrap is a front-end framework and although we didn't touch much on it in this chapter, Bootstrap uses a number of CSS classes to place things on the page, make it all 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 Bootstraps CSS page. Have a look through it, and then let's put some of those classes to use.

    a. In the main carousel, Darth Vader image, the "Join the Dark Side" text kind of blocks the image of Darth Vader. Using the Bootstrap / carousel CSS can you move the text and sign up button to be to the left of the image so as to not cover Darth?

    b. If we do part a (above) 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 screen the text is in the "normal" position (centered / lower portion of the image) and for larger images it's on the left as in (a) above?

For part a, if you just wanted to do it in the HTML:

```
<figure class="item active row">
  <img src="{% static 'img/darth.jpg' %}" alt="Join the Dark Side">
   <figcaption class="carousel-caption"
   style="top:0%;left:10%;right:60%;text-align:left">    (1)
      <h1 class>Join the Dark Side</h1>
      <p>Or the light side.  Doesn't matter.  If you're into Star Wars then this
   is the place for you.</p>
      <p><a class="btn btn-lg btn-primary" href="{% url 'register' %}"
   role="button">Sign up today</a></p>
   </figcaption>
</figure>
```

    1. Above we simply change the style of the figcaption to push the caption to the top and left `top:0%;left:10%` and have the text wrap at 60% of the right edge, `right:60%`. We could also leave the text centered, but I think it looks better with `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 the CSS and out of our HTML / templates - but we don't want this style to apply to all `.corousel-caption` items. So let's give this particular caption an id (say id=darth_caption) and then update our `mec.css` (where we put all of our custom CSS rules) as follows:

```
1   #darth_caption
2   {
3     top: 0%;
4     left: 10%;
5     right: 60%;
6     text-align: left;
7   }
```

This has the same effect as our previous example but keeps the styling from cluttering up our HTML.

For part b:

Media queries give you the power to create different styling depending upon the size of the screen that is being used. Bootstrap makes it easy to use the power of media queries by defining several screen sizes for you. The following are defined for you (this is taken directly from getbootstrap.com/css:

```
1   /* Extra small devices (phones, less than 768px) */
2   /* No media query since this is the default in Bootstrap */
3
4   /* Small devices (tablets, 768px and up) */
5   @media (min-width: @screen-sm-min) { ... }
6
7   /* Medium devices (desktops, 992px and up) */
8   @media (min-width: @screen-md-min) { ... }
9
10  /* Large devices (large desktops, 1200px and up) */
11  @media (min-width: @screen-lg-min) { ... }
```

you can also set max-width in addition to min-width. If you just wanted to target medium screen sizes, the following media query would do the trick:

```
1   @media (min-width: @screen-md-min) and (max-width: @screen-md-max) { ... }
```

This all makes it look pretty easy, but actually the documentation is a bit confusing here. What this is showing you is LESS and not CSS. So the @screen-lg-min isn't actually defined in the bootstrap.css. You create it in LESS and then compile it to CSS. Using LESS is beyond the scope of this book, so we will just take the values and plug them directly into our CSS. In this case, the CSS will look something like:

```
1   @media screen and (min-width: 992px) and (max-width: 1200px) { ... }
```

The media query is exactly what we need to solve our dilemma. The default settings on the .carousel-caption (centered lower third of the image) is probably what we want for the extra small view. And our custom settings that we did for part a are what we want for everything else.

Remember that Bootstrap is "mobile first", so it's no coincidence that we are using the default setting for mobile. To change our styling for large displays, wrap our original `#darth_caption` CSS rules in a media query like so:

```css
@media screen and (min-width: 768px) {
  #darth_caption
  {
    top: 0%;
    left: 10%;
    right: 60%;
    text-align: left;
  }
}
```

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

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. Go ahead and 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 I'll list some examples of what I have chosen to do. Let's start with the login page.

Short and sweet. One way to do it is make it look like this:

All you have to do is use a little bit of Bootstrap; our 'templates/payments/sign_in.html' file now looks like:

```django
{% extends "__base.html" %}
{% load staticfiles %}
{% block extra_css %}
    <link href="{% static 'css/signin.css' %}" rel="stylesheet">
{% endblock %}
{% block content %}
  <div class="container">
    <form accept-charset="UTF-8" action="{% url 'sign_in' %}"
          class="form-signin" role="form" method="post">{% csrf_token %}
    <h2 class="form-signin-heading">Sign in</h2>
        {% if form.is_bound and not form.is_valid %}
        <div class="alert-message block-message error">
```
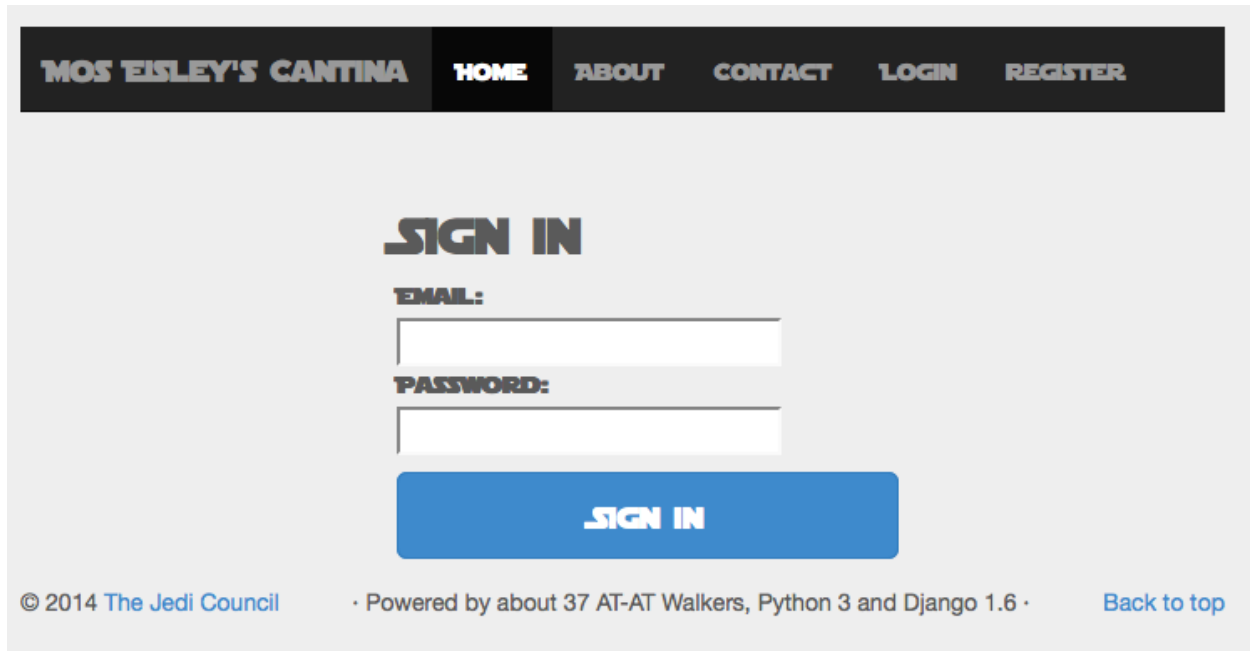
Figure 11.1: Sign-in page

```
13      <div class="errors">
14        {% for field in form.visible_fields %}
15        {% for error in field.errors %}
16        <p>{{ field.label }}: {{ error }}</p>
17        {% endfor %}
18        {% endfor %}
19        {% for error in form.non_field_errors %}
20        <p>{{ error }}</p>
21        {% endfor %}
22      </div>
23    </div>
24    {% endif %}
25    {% for field in form %}{% include "payments/_field.html" %}{% endfor %}
26      <input class="btn btn-lg btn-primary btn-block" name="commit"
   type="submit" value="Sign in">
27    </form>
28  </div>
29 {% 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 %}
```

What I did is to create another block in the `_base.html` template so that I can easily add new stylesheets from inherited pages. The modified `__base.html` now looks like:

```
1   <!DOCTYPE html>
2   {% load staticfiles %}
3
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>Mos Eisley's Cantina</title>
10      <link href="{% static "css/bootstrap.min.css" %}" rel="stylesheet">
11
12
13      <!-- HTML5 Shim and Respond.js IE8 support of HTML5 elements and media
        queries -->
14      <!-- WARNING: Respond.js doesn't work if you view the page via file:// -->
15      <!--[if lt IE 9]>
16        <script
        src="https://oss.maxcdn.com/libs/html5shiv/3.7.0/html5shiv.js"></script>
17        <script
        src="https://oss.maxcdn.com/libs/respond.js/1.4.2/respond.min.js"></script>
18      <![endif]-->
19
20      <link href="{% static "css/mec.css" %}" rel="stylesheet">
21      <!-- custom style of carousel template from bootstrap.com -->
22      <link href="{% static "css/carousel.css" %}" rel="stylesheet">
23      {% block extra_css %}
24      {% endblock %}
25
26    </head>
27    ...
```

As you can see, just after the stylesheets on line 23-24 we have 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-top: 40px;
3     padding-bottom: 40px;
4     background-color: #eee;
5   }
```

```
6
7   .form-signin {
8     max-width: 330px;
9     padding: 15px;
10    margin: 0 auto;
11    font-family: 'starjedi', sans-serif;
12  }
13
14  .form-signin .form-signin-heading,
15  .form-signin .checkbox {
16    margin-bottom: 10px;
17  }
18
19  .form-signin .checkbox {
20    font-weight: normal;
21  }
22
23  .form-signin .form-control {
24    position: relative;
25    height: auto;
26    -webkit-box-sizing: border-box;
27    -moz-box-sizing: border-box;
28    box-sizing: border-box;
29    padding: 10px;
30    font-size: 16px;
31  }
32
33  .form-signin .form-control:focus {
34      z-index: 2;
35  }
36
37  .form-signin input[type="email"] {
38    margin-bottom: -1px;
39    border-bottom-right-radius: 0;
40    border-bottom-left-radius: 0;
41   }
42
43  .form-signin input[type="password"] {
44      margin-bottom: 10px;
45      border-top-left-radius: 0;
46      border-top-right-radius: 0;
47  }
```

Note this particular stylesheet is relatively short as far as stylesheets go, so I could have included it into the `mec.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, I'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. I won't go into details explaining it here, because it's basically the same idea as the sign-in page, but you can get the code here:

```
$ git checkout tags/ch8_ex2_complete
```

One thing I didn't talk about much in this chapter is keeping the unit tests up-to-date. Moving around templates and whatnot may cause your unit tests to fail, so don't forget to update your unit tests as you go. I've updated them as part of the above tagged commit.

Exercise 3

Previously in the chapter we introduced the marketing_*circle*item template tag. The one issue we had with it was that it requires a whole lot of data to be passed into it. Let's see if we can fix that. Inclusion tags don't have to get passed in data. They can inhert context from the parent template. This is done by passing in `takes_context=True` to the inclusion tag decorator like so:

```
@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_respnose`. 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`, 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:

```
from django.shortcuts import render_to_response
from django.template import RequestContext
from payments.models import User
#from main.templatetags.main_marketing import marketing__circle_item

class market_item(object):

    def __init__(self, img, heading, caption, button_link="register",
                 button_title="View details"):
        self.img = img
        self.heading = heading
        self.caption = caption
```

```python
13          self.button_link = button_link
14          self.button_title = button_title

15
16  market_items = [market_item(img="yoda.jpg", heading="Hone your Jedi Skills",
17                              caption="All members have access to our unique"
18                              " training and achievements latters.  Show off "
19                              "your Star Wars skills, progress through the "
20                              "levels and show everyone who the top Jedi Master
    is!"
21                              ),
22                  market_item(img="clone_army.jpg", heading="Build your Clan",
23                              caption="Engage in meaningful conversation, or "
24                              "bloodthirsty battle! If it's related to "
25                              "Star Wars you better believe we do it :)"
26                              ),
27                  market_item(img="leia.jpg", 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.", button_title="Sign Up Now"
32                              ),
33                  ]

34
35  def index(request):
36      uid = request.session.get('user')
37      # for now just hard-code all the marketing info stuff
38      # to see how this works
39      if uid is None:
40          return render_to_response('main/index.html',
41                                    {'marketing_items':market_items})
42      else:
43          return render_to_response('main/user.html',
44                                    {'marketing_items':market_items,
45                                     'user': User.get_by_id(uid)}
46                                    )
```

I first create a dummy class called `market_item`. (Naming conventions be damned; I'm just going to delete it anyways. :) The class is 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.

Next is to create a bunch of 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 I put the list at the module level namespace; this will make it easier to access from my unit tests (which are going to break as soon as I implement this).

The final thing I need to do is pass the newly created list of marketing items to the template as the context. This is done with one of the following lines:

```
1  if uid is None:
2      return render_to_response('main/index.html',
3                               {'marketing_items':market_items})
4  else:
5      return render_to_response('main/user.html',
6                               {'marketing_items':market_items,
7                                'user': User.get_by_id(uid)}
8                               )
```

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 themarketing_*circle*item' template. It now does more or less nothing:

```
1  @register.inclusion_tag('main/templatetags/circle_item.html',
2                          takes_context=True)
3  def marketing__circle_item(context):
4      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 `market_items` and display the nicely formatted marketing blurb. The code will look like this:

```
1  {% load staticfiles %}
2
3  {% for m in marketing_items %}
4    <div class="col-lg-4">
5      <img class="img-circle" src="{% static 'img/'|add:m.img %}" width=140
6      height=140" alt="{{ m.img }}">
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 creating 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! 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.MainPageT` This is because you index page now requires the `marketing_info` context variable, and we are not passing it into our test. Remember earlier when I said we were putting the `marketing_items` list at the module level to aid with our testing... well, looking at `tests.main.testMainPageView.MainPageTests`

we can do something like this to fix it:

First import the `marketing_items` into the test, so we can reuse the same data:

```
1 from main.views import index, market_items
```

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(resp.content,
4                      render_to_response("main/index.html",
5                                         {"marketing_items":market_items}
6                      ).content)
```

Now rerun and all your tests should pass! Great work.

If you want to go for the bonus and data drive this thing from the database (as opposed to hardcoding a bunch of stuff in your view function), then here is how.

You are 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 # Create your models here.
4 class MarketingItem(models.Model):
5     img = models.CharField(max_length=255)
6     heading = models.CharField(max_length=300)
7     caption = models.TextField()
8     button_link = models.URLField(null=True, default="register")
9     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 django.shortcuts import render_to_response
2 from payments.models import User
3 from main.models import MarketingItem
4
5 def index(request):
6     uid = request.session.get('user')
7     market_items = MarketingItem.objects.all()
8     if uid is None:
9         return render_to_response('main/index.html',
10                                   {'marketing_items':market_items})
```

```
11        else:
12            return render_to_response('main/user.html',
13                                        {'marketing_items':market_items,
14                                         'user': User.get_by_id(uid)}
15                                        )
```

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... and you'll probably get some errors because you didn't run syncdb to create the new table. But even if you did, you won't see any of the marketing items on your index page since you've got nothing in your database. To have Django automatically load some starting data for you, do the following:

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`. For example:

```
1  [
2    {
3      "model": "main.MarketingItem",
4      "pk": 1,
5      "fields":{
6          "img":"yoda.jpg",
7          "heading":"Hone your Jedi Skills",
8          "caption":"All members have access to our unique training and
     achievements ladders.  Show off your Star Wars skills, progress through the
     levels and show everyone who the top Jedi Master is!"
9              }
10    },
11    {
12     "model": "main.MarketingItem",
13     "pk": 2,
14     "fields":{
15         "img":"clone_army.jpg",
16         "heading":"Build your Clan",
17         "caption":"Engage in meaningful conversation, or bloodthirsty battle! If
     it's related to Star Wars you better believe we do it :)"
18             }
19    },
20    {
21     "model": "main.MarketingItem",
22     "pk": 3,
23     "fields":{
24         "img":"leia.jpg",
```

```
25        "heading":"Find Love",
26        "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.",
27        "button_title":"Sign Up Now"
28       }
29    }
30 ]
```

Once this is all in place, every time you run `syncdb`, the data from the `initial_data.json` file will automatically be put into the database. Rerun syncdb, fire up the Django development server, and you should now see your marketing info. Awesome! Oh and don't forget, you now have a failing unit test... You might want to fix that. :)

After you've done that, you can get the final version of the code here:

```
1 git checkout tags/Ch8_ex3
```

Chapter 9

Exercise 1

US3 Main Page says that the main page should be a place for announcements and listings of current happenings. We have implemented user announcements, i.e. status reports, but we should also have a section for system announcements / current events. Using the architecture described in this chapter, can you create an Announcements `info_box` so we can post announcements?

The simplest way to implement this requirement is with a simple template. Let's call it `main/_announcements.html`:

```
<! -- system announcements -->
{% load staticfiles %}
<section class="info-box" id="announcements">
  <h1>orders from the Council</h1>
  <div class="full-image">
     <img src="{% static 'img/beat_jarjar.jpg' %}"/>
  </div>
  <div >
     <h3>April 1: Join us for our annual Smash Jar Jar bash</h2>
     <p>Bring the whole family to MEC for a fun-filled day of smashing Jar Jar
   Binks!</p>
  </div>
</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 / position of the image. They are put in the mec.css and look like:

```
.full-image
{
    overflow: hidden;
}

.full-image img {
  position: relative;
  display: block;
  margin: auto;
  min-width: 100%;
  min-height: 100px;
}
```

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 small, 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 is just another include. We have 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` (Yeah I know, big suprise there, :)

```
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 are going to allow 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 are timestamping 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 years old on the site.

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 horribly customizable, but it's pretty easy to use and does all the heavy lifting for us.

We can install it with pip:

```
$ pip install django-embed-video
```

Don't forget to add it to the requirements.txt file as well as the `INSTALLED_APPS` tuple in `django_ecommerce\settings.py`.

Once that is done, we can update our `main/_announcements.html` template.

```html
<! -- system announcements -->
{% load staticfiles %}
{% load embed_video_tags %}
<section class="info-box" id="announcements">
  <h1>orders from the Council</h1>
  {% for a in announce %}
    <div class="full-image">
      {% if a.vid %}
        {% video a.vid "medium" %}
      {% else %}
        <img src="{% static 'img/'|add:a.img %}"/>
      {% endif %}
    </div>
    <div > {{ a.info | safe }}</div>
    <br>
  {% endfor %}
</section>
```

- **Line 3**: You'll notice we are loading a new set of tags. These are the tags from `django-embed-video`
- **Line 6**: Loop through all the announcements
- **Line 8**: Give videos priority; if we have a video entry in the database, show that
- **Line 9**: load the video in an iframe using `django-embed-vide`
- **Line 11**: If it's an image, show that instead
- **Line 14**: Insert 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` we add another context variable. The relevant part is below:

```python
announce_date = date.today() - timedelta(days=30)
```

```
2    announce = (Announcement.objects.filter(
3                    when__gt=announce_date).order_by('-when'))
4
5    return render_to_response('main/user.html',
6                            {'user': User.get_by_id(uid),
7                             'reports': status,
8                             'announce': announce},
9                            context_instance=RequestContext(request),
10                           )
```

Basically we are grabbing all the announcements in the last 30 days and ordering them, so the most recent will appear at the top of the page.

Exercise 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 actions that we might come up with in the future? This may be a nice way to increase participation, because everybody likes badges, right? Go ahead and implement that. 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 user has.

There are several ways to do this. Here is what I think is probably 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, 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:

```
class StatusReport(models.Model):
    user = models.ForeignKey('payments.User')
    ...
```

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 `users.html` page. Updating our `main.views.index()` function we now have:

```
#first half of the function remains unchanged
else:
    #membership page
    status = StatusReport.objects.all().order_by('-when')[:20]

    announce_date = date.today() - timedelta(days=30)
    announce = (Announcement.objects.
                filter(when__gt=announce_date).order_by('-when'))

    usr = User.get_by_id(uid)
    badges = usr.badges.all()

    return render_to_response('main/user.html',
                              {'user': usr,
                               'badges': badges,
                               'reports': status,
                               'announce': announce},
                              context_instance=RequestContext(request),
                              )
```

On **Line 12** 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 will just set that list to the context variable `badges` and pass it into the template.

Now the `users.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 looks like this:

```
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         <img class="img-circle" src="{% static 'img/'|add:bdg.img %}"
8         width="100" height="100" alt=" {{ bdg.name }}"
9         title="{{ bdg.name }} - {{ bdg.desc }}">
10        <p>{{ bdg.desc }}</p>
11      </div>
12    {% endfor %}
13 </section>
```

We loop through the badges and show then 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 Achieve-

ments 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 book yet, but we will in an upcoming chapter. For now, just have a look at the code, which will be placed in `application.js`:

```
//show status
$("#show-achieve").click(function() {
    a = $("#achievements");
    l = $("#show-achieve");
    if (a.hasClass("hide")) {
        a.hide().removeClass('hide').slideDown('slow');
        l.html("Hide Achievements");
    } else {
        a.addClass("hide");
        l.html("Show Achievements");
    }
    return false;
});
```

This is the click event handler that get's called when you click on the "Show Achievements" link. Let's walk through it.

- **Line 2**: Set up the click even handler.
- **Line 3**: Grab a reference to the achievements info box.
- **Line 4**: Grab 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).
- **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**: Change 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.

That's it. I've also set up some dummy data for you to use in the git repo:

```
$ git checkout tags/ch9_ex2
```

Chapter 10

Exercise 1

Flesh out the unit tests. In the `JsonViewTests`, check the case where there is no data to return at all, and test the POST with and without valid data.

As a little bonus to the readers that actually do the exercises (because I know not everybody does), I'm going to talk about a "better" way to do unit testing for DRF views: use DRF itself. In Chapter 10 we showed a number of examples of how to unit test DRF views using Django's standard Unit-test library, mainly so we could better understand how DRF works behind the scenes.

There is, however, a better way. DRF provides some very useful helper classes that extend Django's standard unit-testing classes and make testing API requests pretty straightforward. We will go through a couple of examples below. Also be sure to check out the docs.

We will start off by re-writing the two tests discussed in the chapter:

```python
from django.test import TestCase
from main.models import StatusReport
from main.serializers import StatusReportSerializer
from main.json_views import StatusCollection, StatusMember
from rest_framework import status
from rest_framework.test import APIRequestFactory, force_authenticate
from payments.models import User

class JsonViewTests(TestCase):

    @classmethod
    def setUpClass(cls):
        cls.factory = APIRequestFactory()
        cls.test_user = User(email="test@test.com")

    def get_request(self,authed=True):
        request = self.factory.get("")
        if authed:
            force_authenticate(request, self.test_user)

        return request

    def test_get_collection(self):
        status = StatusReport.objects.all()
        expected_json = StatusReportSerializer(status, many=True).data

        response = StatusCollection.as_view()(self.get_request())

        self.assertEqual(expected_json, response.data)
```

```
31    def test_get_collection_requires_logged_in_user(self):
32        response = StatusCollection.as_view()(self.get_request(authed=False))
33
34        self.assertEqual(response.status_code, status.HTTP_403_FORBIDDEN)
```

We are no longer creating our `dummyRequest` object but rather we are using `rest_framework.test.APIRequestFact`
to create the request that we need. This way we don't have to spend time implementing a fake request
object. To make it even easier to use the `APIRequestFactory`, I've put in a couple of helper functions
because we are going to be creating a lot of requests. The two functions are:

```
1  @classmethod
2  def setUpClass(cls):
3      cls.factory = APIRequestFactory()
4      cls.test_user = User(id=2222, email="test@test.com")
5
6  def get_request(self,authed=True):
7      request = self.factory.get("")
8      if authed:
9          force_authenticate(request, self.test_user)
10
11     return request
```

The `setUpClass` function creates a test_user (that we will use for authentication) and DRF's
`APIRequestFactory`. This just saves us some typing. Then I've created a function called
`get_request` that will return a (you guessed it) `GET` request. There is an optional argument
`authed` which allows us to specify if this request should be from an authenticated user. If `authed`
`== True` then we use DRF's `force_authenticate` function, which makes the request act like it has
a logged in user. (This is in effect the same thing we were doing in Chapter 10, but here it's done for
us by DRF.)

With the helper functions out of the way, we have two tests: to verify getting the collection with a
logged in and a logged out user.

Expanding the tests, we can write a `test_get_member`:

```
1  def test_get_member(self):
2      status = StatusReport.objects.get(pk=1)
3      expected_json = StatusReportSerializer(status).data
4
5      response = StatusMember.as_view()(self.get_request(), pk=1)
6
7      self.assertEqual(expected_json, response.data)
```

The only difference here is we are passing in the `pk=1` 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  #modify get_request to work on all the HTTP methods
2  def get_request(self, method='GET', authed=True):
```

```
3       request_method = getattr(self.factory, method.lower())
4       request = request_method("")
5       if authed:
6           force_authenticate(request, self.test_user)
7
8       return request
9
10  def test_delete_member(self):
11      status = StatusReport(user=self.test_user, status="testing")
12      status.save()
13
14      response = StatusMember.as_view()(
15          self.get_request(method='DELETE'), pk=status.pk)
16
17      self.assertEqual(response.status_code, status.HTTP_204_NO_CONTENT)
```

First we change our `get_request` helper function to take in the name of the HTTP method to use for the request. Then, we use getattr to call the method by the name that we passed in. This is basically the same thing as saying:

```
1  def get_request(self, method='GET'):
2      if method == 'GET':
3          request = self.factory.get()
4      elif method == 'DELETE':
5          request = self.factory.delete()
6      elif method == 'POST':
7          request = self.factory.post()
8      else:
9          #raise some nasty error
```

Think of it as shorthand. :) So now we can call:

```
1  response =StatusMember.as_view()(
2      self.get_request(method='DELETE'), pk=status.pk)
```

That will get us to call delete on the status_report that we just created.

Now it's just a question of testing all the various combinations of 'GET', 'POST', 'DELETE' etc on the 'StatusCollection' and 'StatusMember' views.

Exercise 2

Just for practice, extend the REST API to cover the `user.models.Badge`.

To create the JSON API you need to update three files, namely:

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

First to create the serializer:

```python
class BadgeSerializer(serializers.ModelSerializer):

    class Meta:
        model = Badge
        fields = ('id', 'img', 'name', 'desc')
```

Nothing to it. :) Just use the handy `serializer.ModelSerializer`.

Now let's create the Collection and Member views:

```python
class BadgeCollection(mixins.ListModelMixin,
                      mixins.CreateModelMixin,
                      generics.GenericAPIView):

    queryset = Badge.objects.all()
    serializer_class = BadgeSerializer
    permission_classes = (permissions.IsAuthenticated,)

    def get(self, request):
        return self.list(request)

    def post(self, request):
        return self.create(request)

class BadgeMember(mixins.RetrieveModelMixin,
                 mixins.UpdateModelMixin,
                 mixins.DestroyModelMixin,
                 generics.GenericAPIView):

    queryset = Badge.objects.all()
    serializer_class = BadgeSerializer
    permission_classes = (permissions.IsAuthenticated,)

    def get(self, request, *args, **kwargs):
        return self.retrieve(request, *args, **kwargs)

    def put(self, request, *args, **kwargs):
        return self.update(request, *args, **kwargs)

    def delete(self, request, *args, **kwargs):
        return self.destroy(request, *args, **kwargs)
```

It is a bit of copy and paste from the `StatusMember` and `StatusCollection`, but it's pretty clear exactly what is going on there. Finally we have to wire up our URI's in `main/urls.py`:

```
urlpatterns = patterns('main.json_views',
    url(r'^status_reports/$', json_views.StatusCollection.as_view()),
    url(r'^status_reports/(?P<pk>[0-9]+)/$',json_views.StatusMember.as_view()),
    url(r'^badges/$', json_views.BadgeCollection.as_view()),
    url(r'^badges/(?P<pk>[0-9]+)/$', json_views.BadgeMember.as_view())
```

Don't forget to add the unit tests as well. I'll leave this bit out because 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 simliar 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

Did you know that the browsable API uses Bootstrap for the look and feel? Since we just learned Bootstrap, and it wouldn't do to have our browsable API not look like the rest of our site, 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 the `rest_framework/api.html` template look like this:

```
{% extends "rest_framework/base.html" %}
{% load staticfiles %}
{% block bootstrap_theme %}
    <link href="{% static "css/bootstrap.min.css" %}" rel="stylesheet">
    <!-- HTML5 Shim and Respond.js IE8 support of HTML5 elements and media
    queries -->
    <!-- WARNING: Respond.js doesn't work if you view the page via file:// -->
    <!--[if lt IE 9]>
      <script
    src="https://oss.maxcdn.com/libs/html5shiv/3.7.0/html5shiv.js"></script>
      <script
    src="https://oss.maxcdn.com/libs/respond.js/1.4.2/respond.min.js"></script>
    <![endif]-->
    <link href="{% static "css/mec.css" %}" rel="stylesheet">
{% endblock %}
```

All we did was to overwrite the `bootstrap_theme` block and set it to use the Bootstrap CSS that we are using (i.e. v3 instead of the v2.2 that comes with DRF) and then also use our custom `mec.css`, mainly to get our cool star_jedi font. Again the docs hava lots more information about how to customize the look and feel of the browsable API, so have a look if you're interested.

Exercise 4

Speaking of APIs, we don't have permissions on the browsable API; put 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:

```
url(r'^api-auth/', include('rest_framework.urls', namespace='rest_framework')),
```

With this, you will now be able to browse to `/api/v1/api-auth` and you will get a login screen 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:

```
from rest_framework.decorators import api_view
from rest_framework.response import Response
from rest_framework.reverse import reverse


@api_view(('GET',))
def api_root(request):
    return Response({
        'status_reports': reverse('status_reports_collection',
                                  request=request),
        'badges': reverse('badges_collection', request=request),
    })
```

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`:

```
url(r'^$', 'api_root'),
```

Now as an API consumer we don't have to guess what the avaliable resources are in our REST API; just browse to `api/v1` and we are good to go.