The Django Test-Driven Development Cookbook

Tutorial based on: The Django Test Driven Development Cookbook by Martin Brochhaus

Capstone #37Team: Jessica Blasch, Tyson Gieszler, Tyler Gilbert, JP Grattan

Portland State University | Portland, OR, U.S.A.

Authored by: Jessica Blasch
OS: macOS Sierra V 10.12.4

https://www.youtube.com/watch?v=41ek3VNx_6Q

Code Editor:

I have Canopy installed on my machine as well as Python 2.6, 2.7, and 3.6. However, Canopy is geared to interpret Python 2 (legacy). To avoid issues that could arise from this, I believe it important to have an editor that will work with Python 3. I will be using Atom, but here are a few different options.

PyCharm: https://www.jetbrains.com/pycharm/download/#section=mac ← Extremely powerful. Less suitable for a beginner.
Gedit: https://wiki.gnome.org/Apps/Gedit#Download ← Open-source. Available for all operating systems.
Sublime Text 3: https://www.sublimetext.com/3 ← Free evaluation period. For all operating systems.
Atom: https://atom.io/ ← Open-source. Created by GitHub. Available for Windows, OS X, and Linux.

Set up Virtual Environment:

If you are using a Windows machine, look at the links provided below. I had to reference a few different tutorials to find what would work on my machine.

Source(s): http://python-guide-pt-br.readthedocs.io/en/latest/dev/virtualenvs/
https://tutorial.djangogirls.org/en/django_installation/

It is possible to skip this step, but it's *highly recommended*. A virtual environment (also called virtualenv) will isolate your Python/Django setup on a per-project basis. This means that any changes you make to one website won't affect any others you're also developing.

Start by finding a directory in which you want to set up your Django project and create the virtualenv; your home directory, for example. On a Mac, it might look like /Users/Name (where Name is the name of your login)

→ Important: use pip3 for installs, if you want your packages to be applied to Python3

In terminal:

1. Install virtualenv and virtualenvwrapper

```
$ pip3 install virtualenv
$ pip3 install virtualenvwrapper
```

If you already have these installed for Python 3, the system will tell you the requirement is already met.

2. Set up a folder for your Django project and create the virtual environment

```
$ mkdir _tested
$ cd _tested
$ python3 –m venv myvenv
```

← Create a new directory –all lowercase.
← cd into the new directory
← This is the general command to make virtualenv called myvenv. The command will create a directory called "myenv" that contains our virtual environment set up for Python 3.

3. Start your virtual environment

```
$ source myvenv/bin/activate
```

← You will know that you have virtualenv started when you see that the prompt in your console is prefixed with (myenv)

4. Make sure you have the latest version of pip

```
(myvenv) ~$ pip install --upgrade pip
```

5. Install Django into your virtual environment

   (myvenv) ~$ pip install Django

- That's it. You're now ready to create a Django application

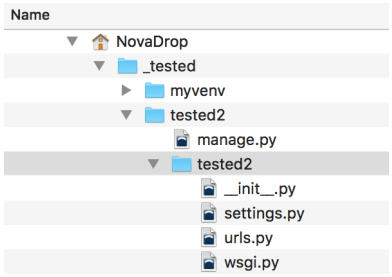
➔ **BEGIN TRANSCRIBING TUTORIAL:**

Project Setup:

1. Start a new Django project

   There are two ways to go about this. Either 'Django-admin' or 'Django-admin.py' will work. What follows will have different results. You may use either. I will be using option (b).

   a. (myvenv) ~$ django-admin.py startproject tested2

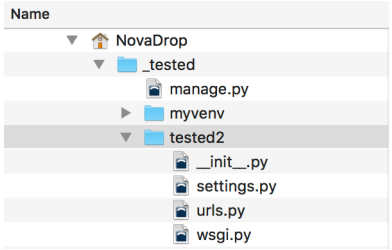


This creates a root folder named tested2 (usually 'mysite' in Django documentation) in the directory you created for your project. Inside tested2 is a manage.py file and settings folder, *also named tested2*, that contains additional python files. The root folder name can be changed to whatever name you want. However, the name of the settings folder contained within (highlighted) must not be changed, because it's in the settings.

To get to settings folder:

$ cd tested2/tested2

   b. (myvenv) ~$ django-admin startproject tested2 .



The period . is crucial because it tells the script to install the Django project in your current directory (for which the period . is a short-hand reference).

The root folder is the directory you created for your project. As before, the name of the settings folder contained within (highlighted) must not be changed, because it's in the settings.

Advantage – To get to settings folder:

$ cd tested2

2. Add a "**test_settings.py**" file in your settings folder _tested/tested2

   a. Create "test_settings.py" file

      This file needs to be placed next to the settings.py file. There are two ways to go about this.

      i. In terminal:

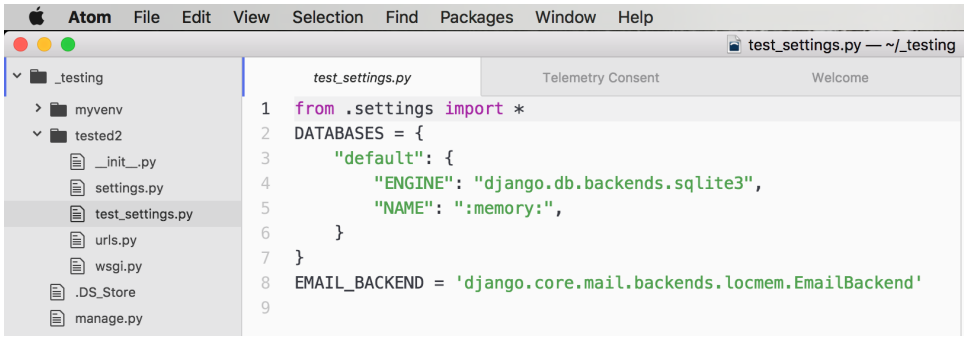
         (myvenv) ~$ touch tested2/test_settings.py ← This is the command if you are in the root folder. You want this file in the settings directory where the "settings.py" file is.

      ii. In IDE:

         Open project in your IDE (Atom shown)
         Right-click on your project folder and add new file called "test_settings.py"

   b. Place the following text in the tested2/**test_settings.py** file

```
from .settings import *

DATABASES = {
   "default": {
      "ENGINE": "django.db.backends.sqlite3",
      "NAME": "memory"
   }
}

EMAIL_BACKEND = 'django.core.mail.backends.locmem.EmailBackend'
```



What this code means:

Instead of using a Postgres or MySQL database, you want to use an "in-memory" SQLite database, which makes the tests *really* fast. –Because, usually, you will have hundreds of tests and every single test will destroy the database and create a new database. If you do that with a database that needs I/O operations on a hard drive, it's going to be *really* slow. So, in-memory is a good way around that.

Email backend is set to local memory, so that you don't actually send real emails while running tests. –Tests are usually run hundreds (or thousands) of times, so you'd send a lot of emails accidentally.

3. Install pytest & plugins and create "**pytest.ini**" in the root folder _tested

   a. Create the "pytest.ini" file in root folder

      This file needs to be placed next to the manage.py file. There are two ways to go about this.

      i. In terminal

         (myvenv) ~$ touch pytest.ini ←This is the command if you're in the root folder.

      ii. In IDE:

         Open project in your IDE (Atom shown)
         Right-click on your project folder and add new file called "test_settings.py"

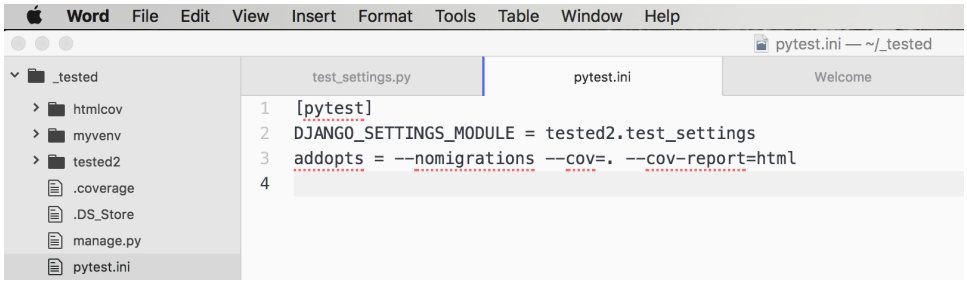
   b. Place the following text in the pytest.ini file

      ```
      [pytest]
      DJANGO_SETTINGS_MODULE = tested2.test_settings
      addopts = --nomigrations --cov=. --cov-report=html
      ```

      ← You need tell it where the test setting file is
      ← This means add more options to the command argument.

      These are flags you would usually use when you execute the pytest command on the command line.
      e.g.
      $ py.test --

      This way, you don't have to type these flags all the time.



   c. Install a few python libraries

      In terminal:

      ```
      (myvenv)…$ pip install pytest
      (myvenv)…$ pip install pytest-django
      (myvenv)…$ pip install pdbpp
      (myvenv)…$ pip install pytest-cov
      (myvenv)…$ deactivate
      (myvenv)…$ source myvenv/bin/activate
      ```

      pytest has lots of plugins. You can use them by just installing them.
      https://pypi.python.org/pypi/pytest/3.0.7

      pytest-django is a plugin in pytest
      https://github.com/pytest-dev/pytest-django

      pdb++ is a drop-in replacement for pdb. It's for setting breakpoints into your tests and then you'll be able to use the pdb++ debugger which is nicer than the original, because it has colors and code completion.
      https://pypi.python.org/pypi/pdbpp/

      pytest coverage helps you generate a coverage report. When we run our tests, it generates a bunch of html files. You can see all your code files and the percentage of lines in the file that have been hit by your tests.
      https://pypi.python.org/pypi/pytest-cov/2.4.0

      100% test coverage doesn't mean you have good tests. However, without 100% test coverage it's easy to get in a situation where sloppy fixes add up over time and your test coverage goes down.

      - After you install all this, you need to deactivate your virtual environment and then reactivate it (as shown, unless you can get virtual environment wrapper to work). Otherwise it will throw some weird errors.

Now try it!

```
_tested — -bash — 92×24
(myvenv) MacBook-Pro:_tested NovaDrop$ ls
manage.py       myvenv          pytest.ini      tested2
(myvenv) MacBook-Pro:_tested NovaDrop$ py.test
============================= test session starts ==============================
platform darwin -- Python 3.6.1, pytest-3.0.7, py-1.4.33, pluggy-0.4.0
Django settings: tested2.test_settings (from ini file)
rootdir: /Users/NovaDrop/_tested, inifile: pytest.ini
plugins: django-3.1.2, cov-2.4.0
collected 0 items

---------- coverage: platform darwin, python 3.6.1-final-0 -----------
Coverage HTML written to dir htmlcov

========================== no tests ran in 10.13 seconds ===========================
(myvenv) MacBook-Pro:_tested NovaDrop$
```

pytest tries to be very smart and recursively go into all your folders and subfolders and search for files that somehow have tests in their name. Then it goes into those files and searches for classes that begin with 'test'. Then it goes into those classes and searches for functions that start with the word 'test_'. Then it tries to execute all this code.

–This is what testing is all about. You try to execute a bunch of functions and in those functions are your tests. You try to execute your actual code and you try to pass in certain arguments. You try to call your function with '0', or with very large numbers, or with negative numbers and see if it crashes somehow.

4. Create a "**.coveragerc**" file

   a. Create the ".coveragerc" file in root folder by the manage.py file
      i. In terminal:

         (myvenv) ~$ touch .coveragerc

         ← This is the command if you're in the root folder. This is a *hidden folder* and won't show up in a file finder window.

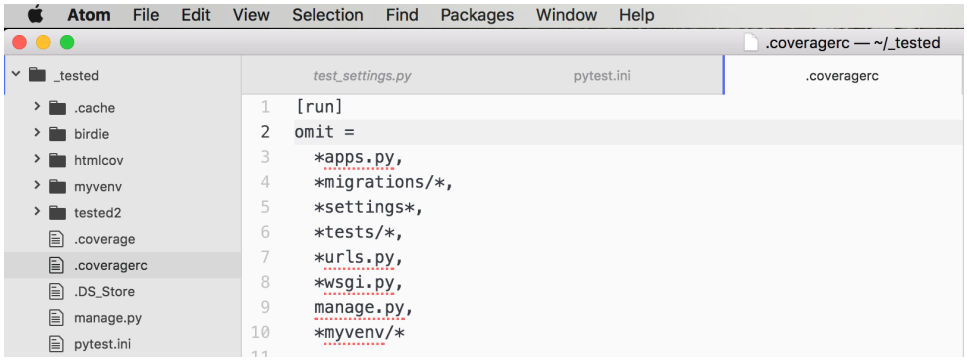
      ii. In IDE:

         Open project in your IDE (Atom shown)
         Right-click on your project folder and add new file called ".coveragerc"

   b. Place the following text in the .covereagerc file

```
[run]
omit =
 *apps.py,
 *migrations/*,
 *settings*,
 *tests/*,
 *urls.py,
 *wsgi.py,
 manage.py,
 *myvenv/*
```

← What this code means is that in certain files you don't care about the test coverage.
e.g.
Don't test migrations, because they're auto-generated. Don't test the test files, because that's over kill. Don't test virtual environment files, because it's not your actual project, etc.



This is a hidden file, so it's easy to forget it exists. It will not show up in normal file searches. However, without it, the coverage report will look messed up. The html file will be a huge list where half of the files don't even interest you. Because you don't test files added to this list it means the coverage will be 0%, which will lower your total coverage and you want your total coverage to be 100%. So, whenever you have a file you don't care about, you need to add it to this list so that it gets removed from the coverage report.

- You are now ready to test!
  py.test will:
  - find all files called "**test_.*py**"
  - execute all functions called "**test_*()**" on all classes that start with "**Test***"

Testing Models:

- Install "**mixer**" and create your first app
- Remove "**tests.py**" and create "**tests**" folder instead
- Each Django app will have a "**tests**" folder
- For each code file, e.g. "**forms.py**" we will have a tests file. e.g. "**test_forms.py**"

mixer is a tool for helping you create test fixtures. Usually, when you run your tests, say you have a certain view that shows your user profile and you want to test that view. That view can only be called by giving the primary key of that user, something like /profile/5. There needs to be something in the database, so this view can even be called, because the first thing this view is going to try is to fetch that model from the database with the model key 5 and if the database is empty, then the view will probably throw a 404 error, because there is no such user. So, we always have the problem of having to put some data into our database that has a certain state so that the test can do its thing.

Now, you could fill your database for tests and it would look something like this:

```
Post = Post()
Post.message = "Hello"
Post.save()
```

Or you can use mixer and have it be one line:
(app name is birdie, model name is post, message is 'Hello')

```
mixer.blend('birdie.Post', message = "Hello")
```

If the birdie model has 20 other fields, mixer will put random values into each of these fields. Sometimes, **mixer** will put in negative numbers, very big numbers, or Unicode names. Maybe you have Unicode errors in your code and you didn't know about it. Your tests might run repeatedly without errors and then one day they fail because **mixer**, by chance, put something into your tests that you never thought about. So, it can help you find test cases you hadn't thought about.

Each Django application has a model. Most Django applications have a database table –a model that they want to use.

→ → →→ → Assume we are building some kind of Twitter clone and we will call it "**birdie**" ← ← ← ← ←

1. Install mixer

   In terminal:

   (myvenv) ~$ pip install mixer

2. Create the folder setup for writing your first test on the birdie Post model

   Django has a chapter on testing in their documentation and they even create a test file for you to remind you to do testing. However, this makes it so all your tests have to live in one file. We're going to do something different.

   In terminal:

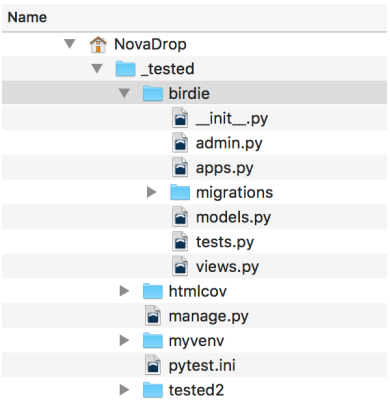
   (myvenv) ~$ django-admin.py startapp birdie
   (myvenv) ~$ rm birdie/tests.py
   (myvenv) ~$ mkdir birdie/tests
   (myvenv) ~$ touch birdie/tests/__init__.py
   (myvenv) ~$ touch birdie/tests/test_models.py

   - Again, "django-admin" will work also.

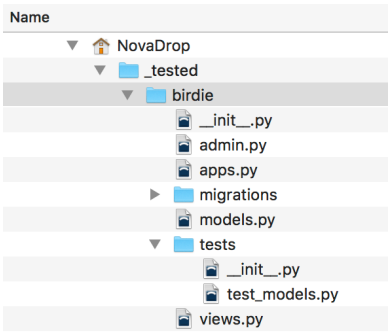
   ← Create "**birdie**" app in the root folder
   ← Delete the "**tests.py**" file in the "**birdie**" folder
   ← Create "**tests**" folder in the "**birdie**" folder
   ← Create an "**__init__.py**" file "**birdie/tests**" folder
   ← Create an "**__test_models.py**" file "**birdie/tests**" folder

   Instead of using the test.py Django created for us, we're going to create a folder for tests. The **birdie/tests** folder has to be a Python module, so it needs an __init__.py file and then we will create one test file for each real code file. We see the **models.py** folder in the **birdie** folder, so we'll create a test_models.py under **birdie/tests**. We're just repeating the name of the original file that we want to test.

   Before alterations to birdie folder:



   After alterations to birdie folder:



- The main building block of most apps is a model
- We should start writing a test for our model
- Some models can have many mandatory fields and it can be quite tedious to create values for all those fields. "mixer" will help here.

Regarding test-driven development:

This means we write the tests before we wright the code. This is the thing that makes it so hard for beginners, because if you're just learning Django, how are you supposed to write tests. Writing tests means you need to know how the real thing works and looks like. Otherwise, it's going to be very hard to write the tests.

We're going to try it anyway…

3. Set up and Test Model: birdie "**Post**"

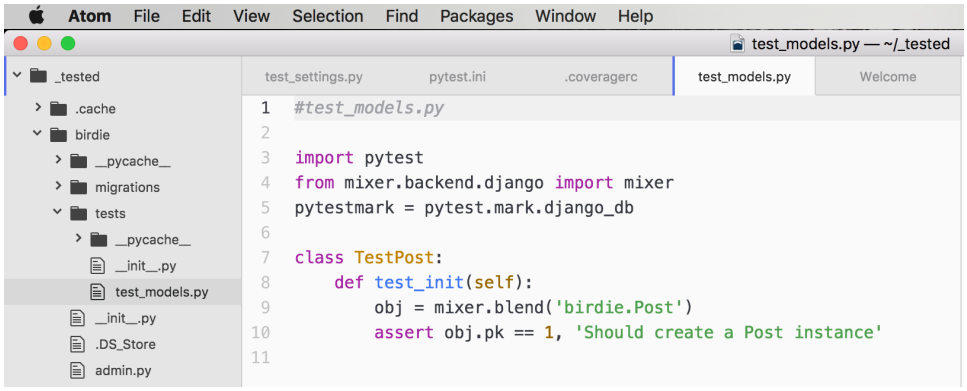
   a. Create a test for Post in the birdie/tests/test_models.py file.

   We know our Twitter clone app needs a certain model, so by writing a test we need to think about the real code. So, we need to think about a name. Let's call the model "Post". This is the first conscious decision about the actual implementation; we came up with a name. "Test" needs to appear before the name, so **pytest** can find this class and realize it's a test class. You can then come up with any function name, but it needs to start with "test_" and then you describe what you're trying to test.

   Here, we try to create an instance of birdie.post and make our first assertion.

   Testing is all about creating a certain situation, then call the assert statement to see if something is true.

```
#test_models.py

import pytest
from mixer.backend.django import mixer
pytestmark = pytest.mark.django_db

class TestPost:
    def test_init(self):
        obj = mixer.blend('birdie.Post')
        assert obj.pk == 1, 'Should create a Post instance'
```



The above code should work without importing pytest or the pytestmark line.

Now, some people will say unit tests need to be completely isolated; not talking to an outside system. The database is an outside system. So, if you want to have extremely fast unit tests you shouldn't even us an in-memory database or any database at all. However, most computing systems are fast enough to handle it and you will always get to a point where you have to test a save function of your object anyway and mock out a Django database.

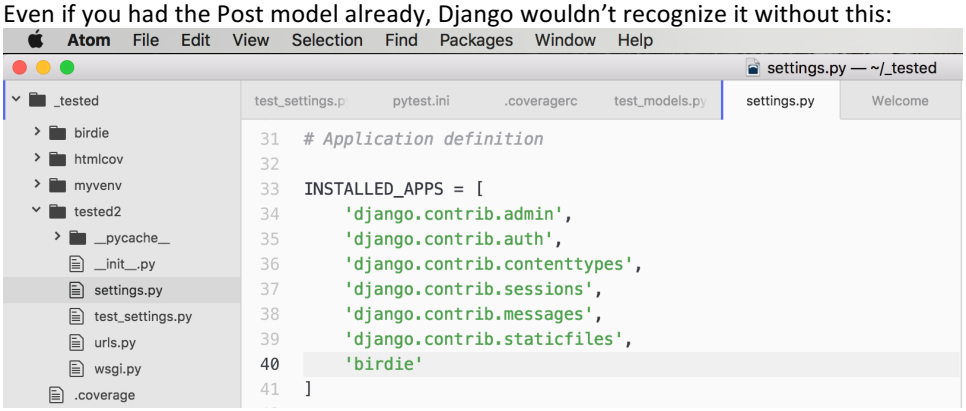
Pytest, by default, has a feature that protects you from actually writing into the database. The above code would crash after the mixer line because mixer tries to call the save function after it has created the object. So, it will trigger, unless we use the "**pytestmark = pytest.mark.django_db**" line of code.

→ You need to tell Django the app exists & create a model for it, before running the test or it will fail.

We still need to create a model named "**Post**" and add birdie to your "**INSTALLED_APPS**"

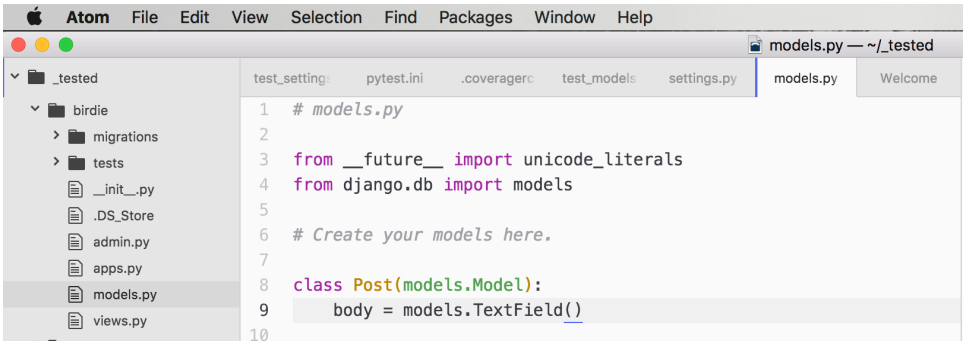
b. Add "**birdie**" app to the tested2/settings.py file

Even if you had the Post model already, Django wouldn't recognize it without this:

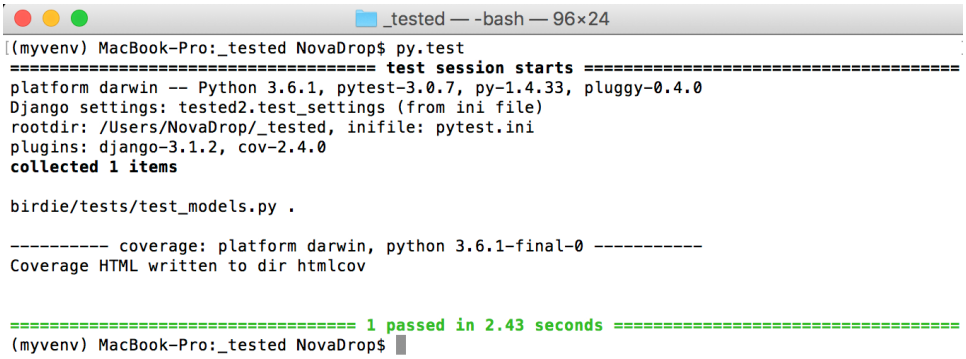


c. Create "**Post**" model in birdie/tests/**models.py** file

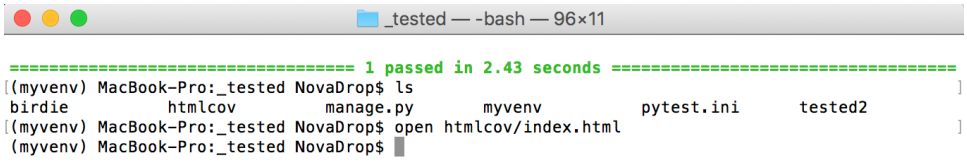
```
# models.py

from __future__ import unicode_literals
from django.db import models

# Create your models here.

class Post(models.Model):
   body = models.TextField()
```

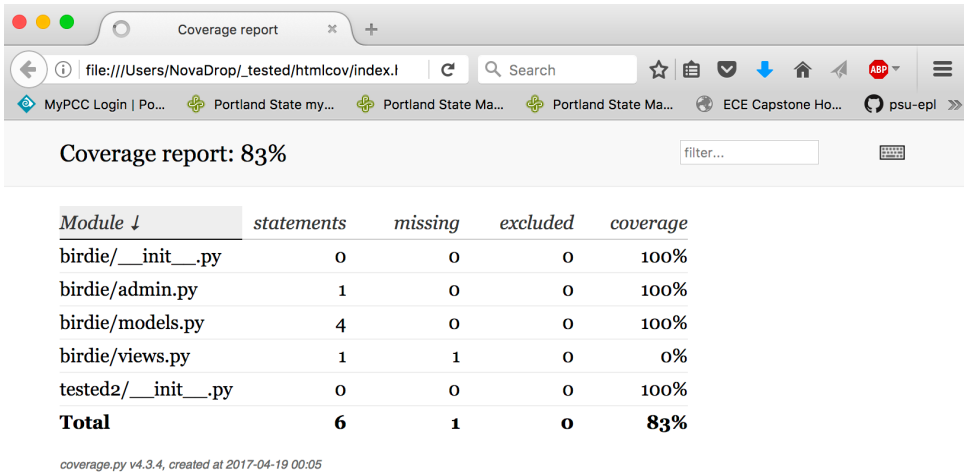


d. Run your test



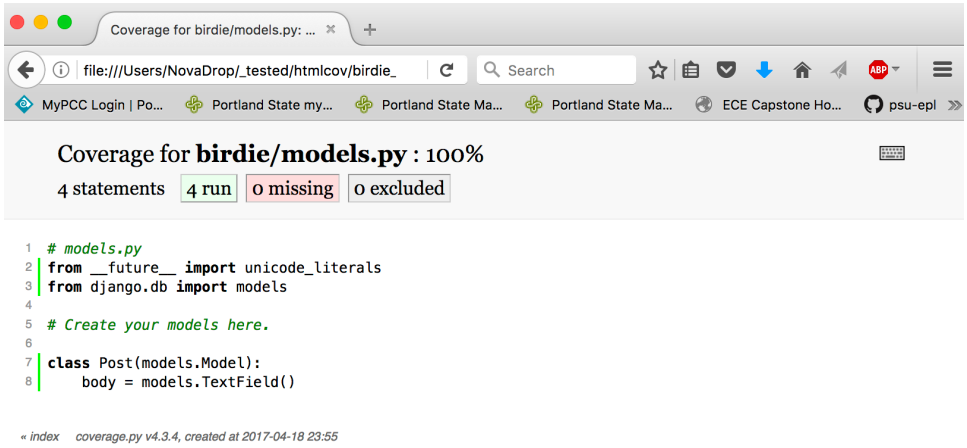
e. Look at your coverage report





Click on "**birdie/models.py**" in the coverage report:

You can see that every line of code has been executed, because our test instantiated the model and there are no functions that could be called.

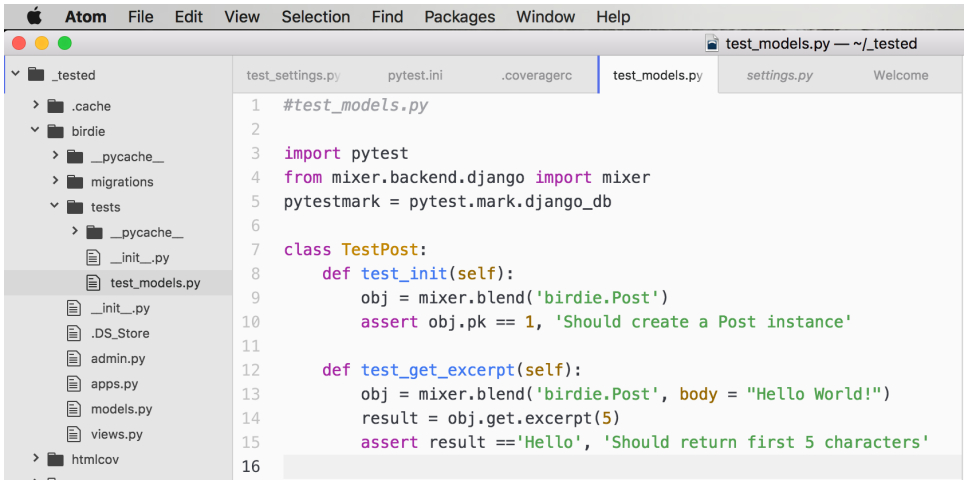


4. Create and Test a Function: function gives a shorter version of the birdie **Post** body.

- Imagine a model function that returns truncated body text
- Before you implement the function, you have to write the test
- That means you have to "use" your function before it even exists
- It helps to think deeply about it, come up with a name with allowed arguments, with type of return value, with different kinds of invocations

a. Create a "**test_get_excerpt**" function for the class TestPost –in the tests/**test_models.py** file

```
#test_models.py

def test_get_excerpt(self):
   obj = mixer.blend('birdie.Post', body = "Hello World!")
   result = obj.get.excerpt(5)
   assert result =='Hello', 'Should return first 5 characters'
```

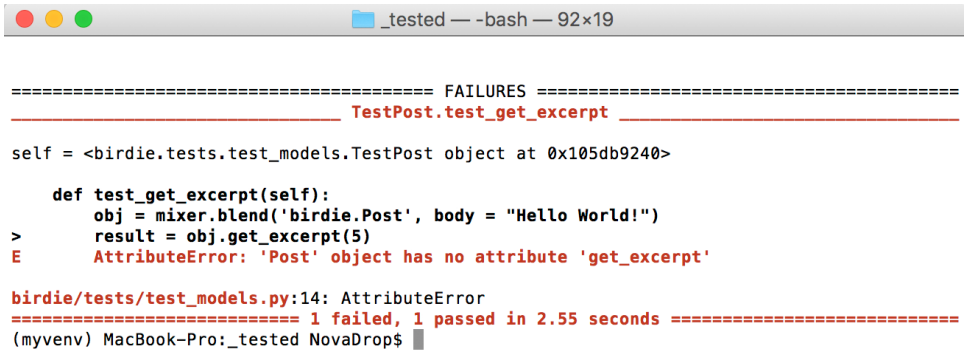


→ Run your tests often and fix each error until they pass.

If you run your tests now you will get failures, but this helps guide what you need to do.

AttributeError: Post object has no attribute "get_excerpt"

Here we see we need to create a "get_excerpt" attribute for the Post model:

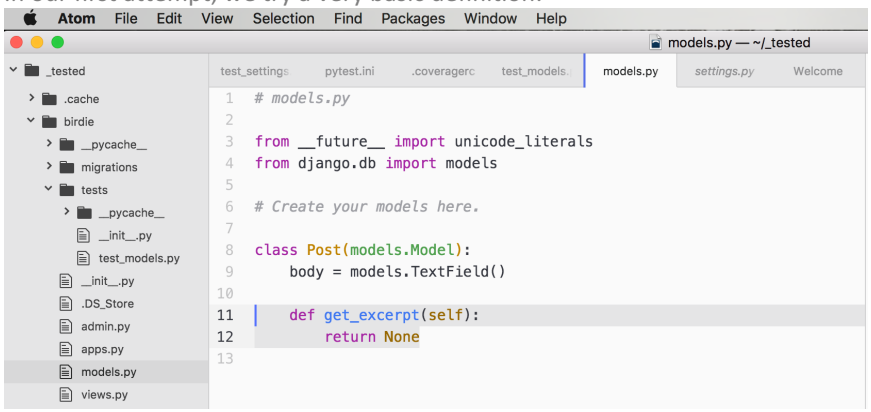


b. Create "**get_excerpt**" attribute definition: for Post In the tests/models.py file

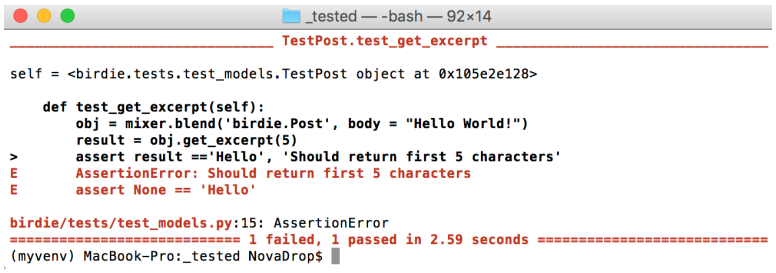
i. Incorrect coding

For the sake of providing examples of possible errors one might get, we show some.

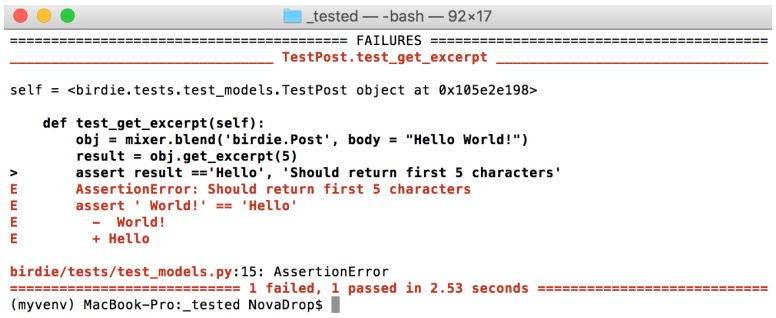
In our first attempt, we try a very basic definition:



→Above says get_excerpt only got one argument and it needs two. We know that we need to allow for filling in the number of characters we want to get back, so we alter the code:



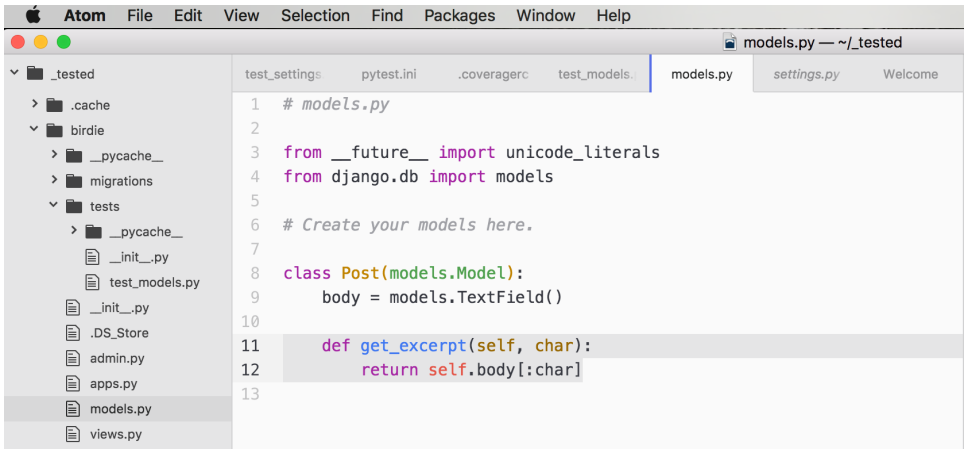
→ So, it's calling the function now, but the result isn't correct, because we're only returning "None". (We need to return what we actually want):



→ This is letting us know we are returning the last 5 characters, not the first 5.

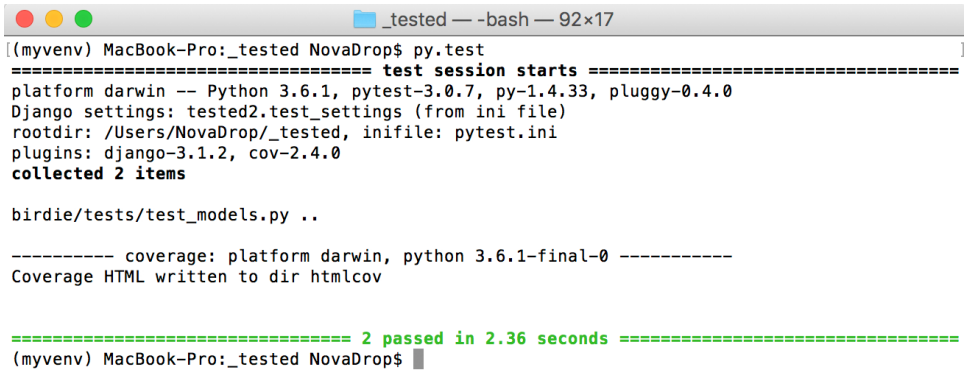
ii. Correct code

```
#tests/models.py

def get_excerpt(self, char):
      return self.body[:char]
```

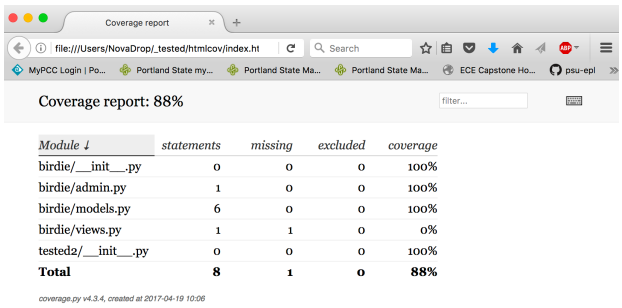


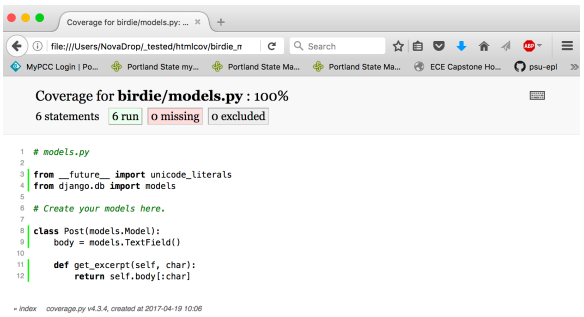
In this case, for example, it may be hard to remember if it's [char:] or [:char] to splice the string. If you don't run test driven development, you would fill in the thing you think it is, then restart your webserver, go to the url, and look. --This takes a bit.

But, here we see our final code revision works.



Don't forget to run/look at your coverage reports:





Testing Admins:

- We want to show the excerpt in our admin list view
- We need to write a function for this because "**excerpt**" is not a database field on the model
- Whenever we need to write a function, we know: We must also write a test for that function
- In order to instantiate an admin class, you must pass in a model class and an AdminSite() instance

We implemented an "excerpt" function on our model. Django admin is a database management tool that can show all the posts that are already in your database. Our "**Post**" object only has a body field which can be a lot of text, so our list would look pretty ugly if its thousands of characters per post. So, we want to only show the excerpt in our list admin. Right now, we can't do that because Django Admin can only show the real fields, *unless you provide an extra function*, then you can show anything. So we need to write this function for our Django Admin as well.

Question: How do you test Django Admin classes? You can't just instantiate them and call their functions.

You will need:

1.	an import of AdminStite	from django.contrib.admin.sites import AdminSite
2.	an import of admin	import admin
3.	an import of models	from import models
4.	an AdminSite instance. → Then you can instantiate your own admin class	site = AdminSite()
5.	admin class instantiation	post_admin = admin.PostAdmin(models.Post,site)

1. Instantiate your admin class and call the new "excerpt" function

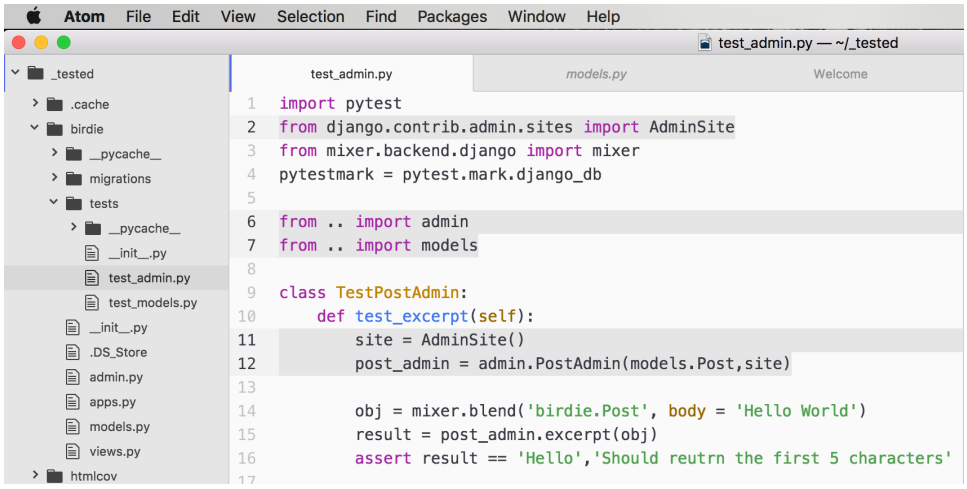
   a. Create birdie/tests/**test_admin.py**

      In terminal:

      (myvenv) ~$ touch birdie/tests/test_admin.py

   b. Place code in birdie/tests/**test_admin.py**

```
import pytest
from django.contrib.admin.sites import AdminSite
from mixer.backend.django import mixer
pytestmark = pytest.mark.django_db

from .. import admin
from .. import models

class TestPostAdmin:
    def test_excerpt(self):
        site = AdminSite()
        post_admin = admin.PostAdmin(models.Post,site)

        obj = mixer.blend('birdie.Post', body = 'Hello World')
        result = post_admin.excerpt(obj)
        assert result == 'Hello','Should reutrn the first 5 characters'
```



How do you get to this without knowing what to do?

- You know you want to test the admin class.
  `result = post_admin.excerpt(obj)`

- You know you want to test the excerpt function.
  `result = post_admin.excerpt(obj)`

- By definition, from Django documentation on writing admin functions, they always have self and then object. So, we know we have to pass in an object.
  `result = post_admin.excerpt(obj)`

- That means we have to create an object here.
  `obj = mixer.blend('birdie.Post', body = 'Hello World')`

- This is why you go to meet ups. This is the stuff you have to Google yourself.
  ```
  site = AdminSite()
  post_admin = admin.PostAdmin(models.Post,site)
  ```

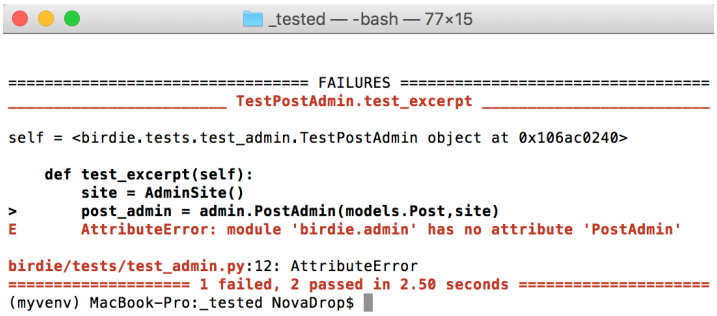
  Because you cannot do this:
  `admin.PostAdmin()`

  It's not a model, like Post, where you can do this:
  `Post()`

  It's not a form, where you can do this:
  `CreateForm()`

  Admins need the model that they are about and the admin site
  `post_admin = admin.PostAdmin(models.Post,site)`

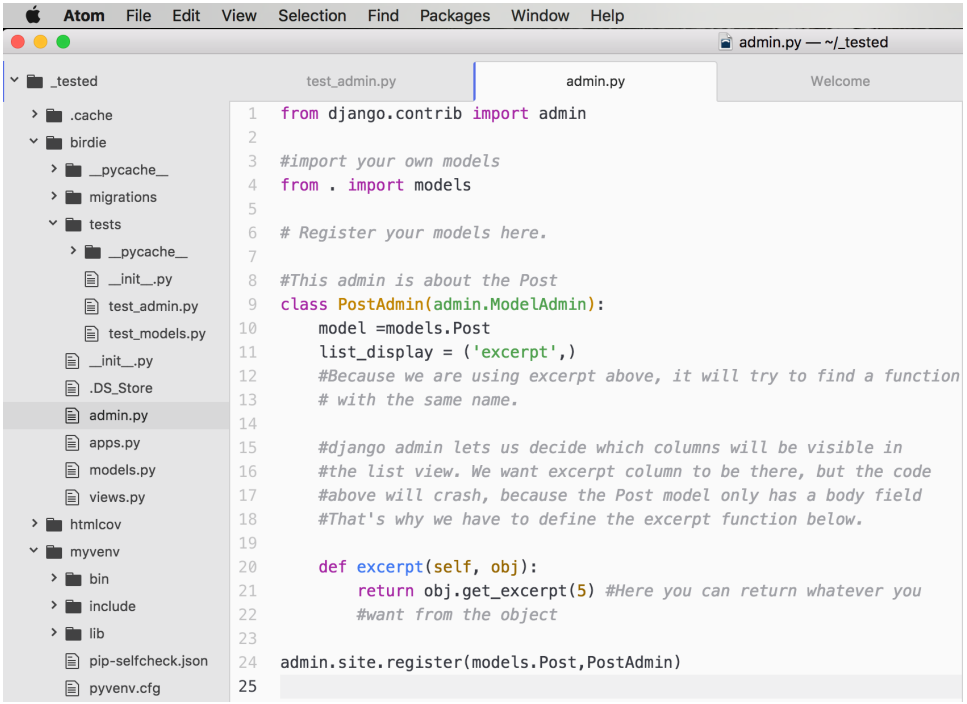
If you run your tests now, you will get an error letting you know you are trying to instantiate a class (admin) that doesn't exist. We'll fix this in the next step.



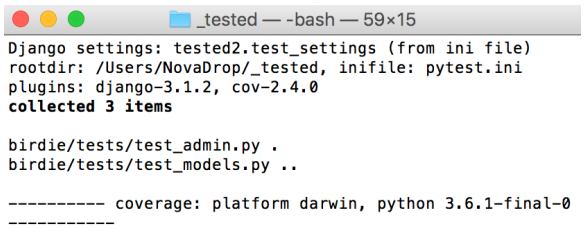
2. Implement the admin class and run the tests again

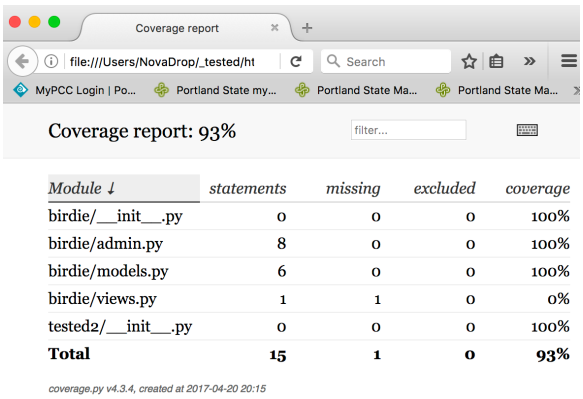
Place the following code in the birdie/tests/**admin.py** file:

```
#admin.py

from django.contrib import admin

from . import models

# Register your models here.

class PostAdmin(admin.ModelAdmin):
   model =models.Post
   list_display = ('excerpt',)

   def excerpt(self, obj):
      return obj.get_excerpt(5)

admin.site.register(models.Post,PostAdmin)
```



Your tests should now pass and you should have 100% coverage.





Testing Views:

- We want to create a view that can be seen by anyone
- Django's "**self.client.get()**" is slow
  "**self.client.get()**" allows you to call any of your views, but your tests will become slow using it. This is because it behaves like it's simulating a real request and going through the whole middleware, template processors, your urls.py to find the view it needs to execute, so it's really slow. If you're missing a template, it will crash, so that's good, but we care more about speed. We should be testing everything in a browser, so we should catch that there when we get a 500 error.
- We will use Django's "**RequestFactory**" instead
  Allows you to instantiate view classes. View classes only work when you pass in a request, so you need to create a fake request object that you can pass into your view. This is what "**RequestFactory**" is good for. It will generate a Django request object for you.
- We can instantiate our class-based views just like we do it in our "**urls.py**", via "**ViewName.as_view()**"
  This turns the view class into a function and then you can use this function and put your request inside.
- To test our views, we create a **Request**, pass it into our view, then make assertions on the returned **Response**
- Treat class-based views as black-boxes

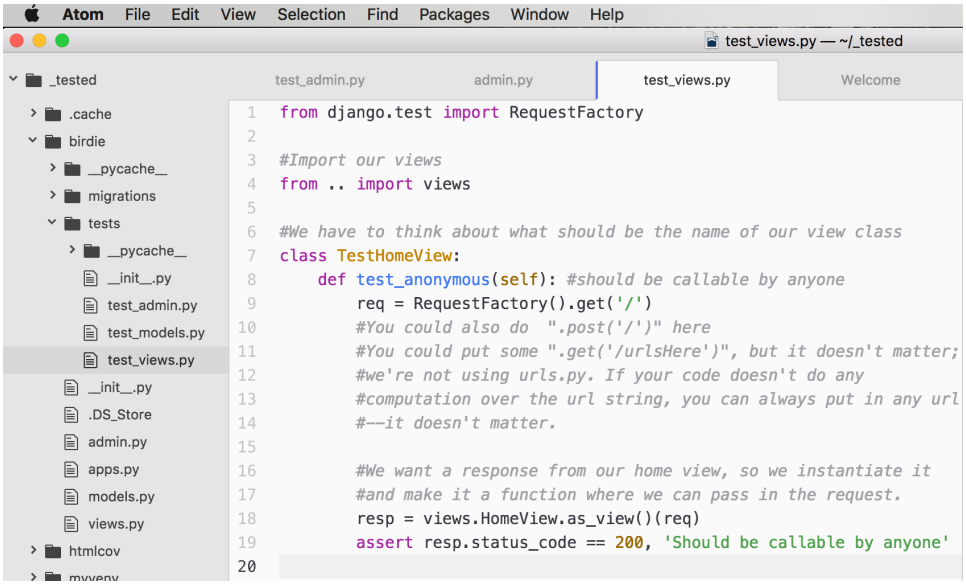
1. Create birdie/tests/**test_views.py** file

   a. In terminal:

   (myvenv) ~$ touch birdie/tests/test_views.py

   b. Place the following code in the birdie/tests/**test_views.py** file

```
from django.test import RequestFactory

from .. import views

class TestHomeView:
    def test_anonymous(self): #should be callable by anyone
        req = RequestFactory().get('/')
        resp = views.HomeView.as_view()(req)
        assert resp.status_code == 200, 'Should be callable by anyone'
```



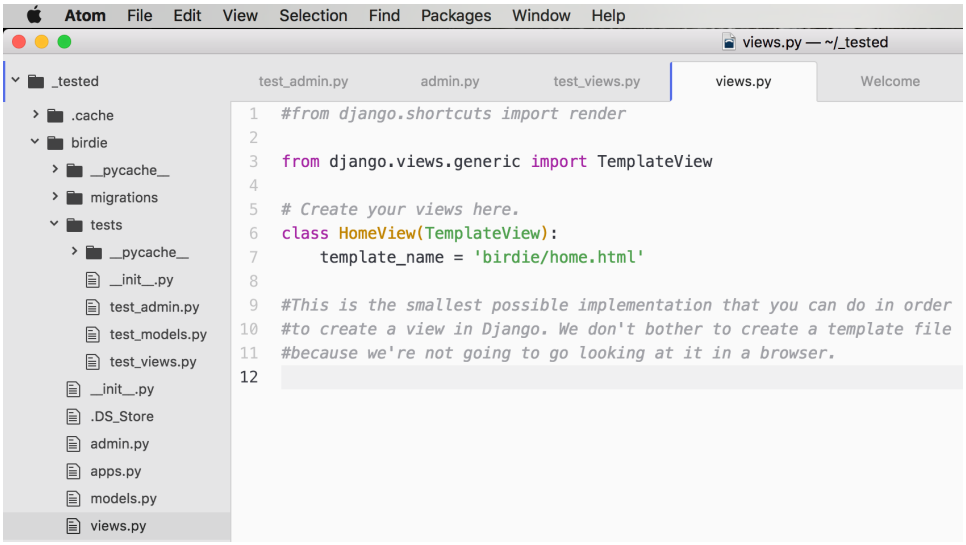
If we run our tests now, we get the following error, which lets us know there is no **HomeView**:

```
_tested — -bash — 79×21
================================== FAILURES ==================================
_____________________ TestHomeView.test_anonymous _____________________

self = <birdie.tests.test_views.TestHomeView object at 0x106ce4630>

    def test_anonymous(self): #should be callable by anyone
        req = RequestFactory().get('/')
        #You could also do  ".post('/')" here
        #You could put some ".get('/urlsHere')", but it doesn't matter;
        #we're not using urls.py. If your code doesn't do any
        #computation over the url string, you can always put in any url
        #--it doesn't matter.

        #We want a response from our home view, so we instantiate it
        #and make it a function where we can pass in the request.
>       resp = views.HomeView.as_view()(req)
E       AttributeError: module 'birdie.views' has no attribute 'HomeView'

birdie/tests/test_views.py:18: AttributeError
==================== 1 failed, 3 passed in 2.56 seconds ====================
(myvenv) MacBook-Pro:_tested NovaDrop$
```

2. Implement the "**Homeview**" and run tests again

   a. Place the following code in the birdie/tests/**views.py** file

```
#views.py

from django.views.generic import TemplateView

# Create your views here.
class HomeView(TemplateView):
   template_name = 'birdie/home.html'
```



   b. We run the tests again and see it passes, which means this view can be called by anyone

```
_tested — -bash — 79×21
(myvenv) MacBook-Pro:_tested NovaDrop$ py.test
============================= test session starts ==============================
platform darwin -- Python 3.6.1, pytest-3.0.7, py-1.4.33, pluggy-0.4.0
Django settings: tested2.test_settings (from ini file)
rootdir: /Users/NovaDrop/_tested, inifile: pytest.ini
plugins: django-3.1.2, cov-2.4.0
collected 4 items

birdie/tests/test_admin.py .
birdie/tests/test_models.py ..
birdie/tests/test_views.py .

---------- coverage: platform darwin, python 3.6.1-final-0 -----------
Coverage HTML written to dir htmlcov


========================== 4 passed in 2.38 seconds ==========================
(myvenv) MacBook-Pro:_tested NovaDrop$
```

Final note:

- This does NOT render the view and test the template.
  If you forgot to create the template, you won't know about it.
  If your template is using lots of template tags, they will never be called, but you should be testing your template tags as unit tests individually already.

- This does NOT call "**urls.py**"
  If you forgot to hook up your urls, this won't catch that.

BUT – these are things you should catch when testing in the browser at the end of production.

Testing Authentication:

- We want to create a view that can only be accessed by superusers
- We will use the **"@method_decorator(login_required)**" trick to protect our view
- That means there must be a "**.user**" attribute on the Request
- Even if we want to test as an anonymous user, in that case Django automatically attaches a "**AnonymousUser**" instance to the request, so we have to fake this as well.

1. Add the following code to the existing code to the birdie/tests/**test_views.py** file

```
#test_views.py

import pytest
from django.contrib.auth.models import AnonymousUser
from mixer .backend.django import mixer
pytestmark = pytest.mark.django_db

class TestAdminView:
   def test_anonymous(self):
      req = RequestFactory().get('/')
      req.user = AnonymousUser()
      resp = views.AdminView.as_view()(req)
      assert 'login' in resp.url

   def test_superuser(self):
      user = mixer.blend('auth.User', is_superuser = True)
      req = RequestFactory().get('/')
      req.user = user #attach user variable to request.
      resp = veiws.AdminView.as_view()(req)
      assert resp.status_code == 200,'Authenticated user can access'
```

← The naming convention used here is a little misleading. It sounds like we're testing for a super user. We're just testing for a normal authenticated user.

A few words on why the coding above is done like it is:

def test_anonymous:

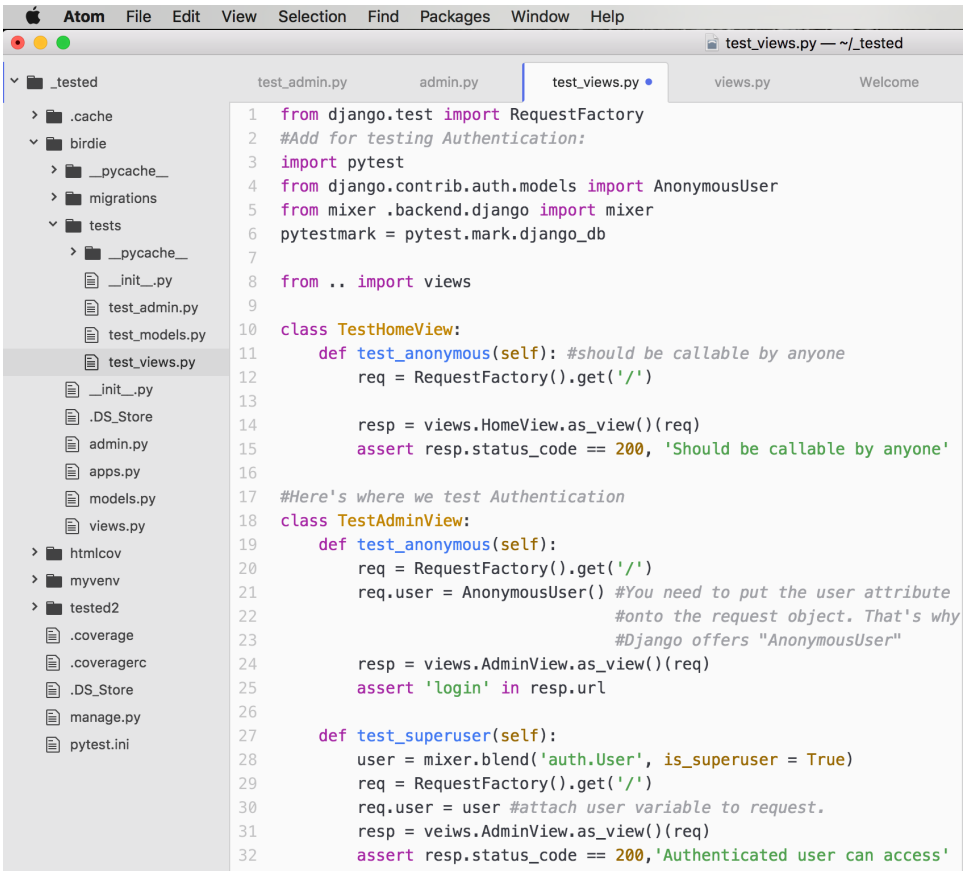
We could do this assertion: assert resp.status_code == 302

If you access a view that's protected with the login decorator and you are not logged in, it will redirect you to the login view. But what if there's code in your view that under certain circumstances does a redirect, so that means maybe a redirect is happening, but not because the user is anonymous.

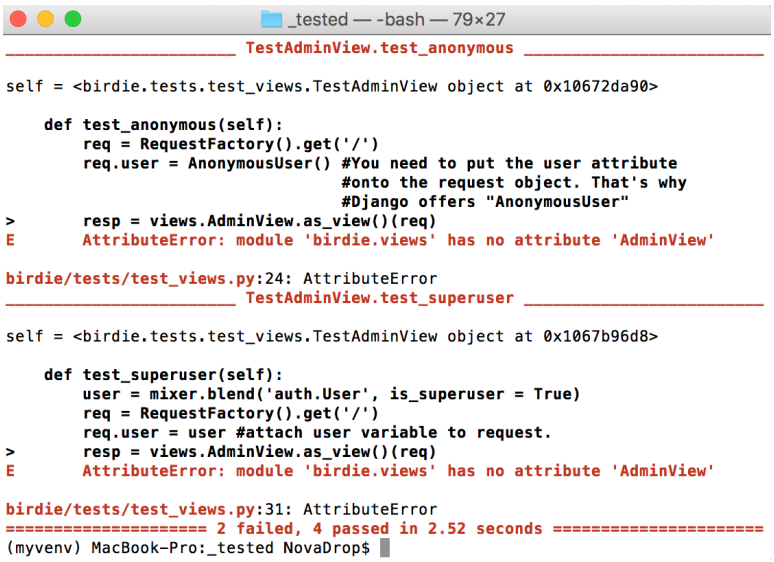
By default the assert using login will look something like assert '/account/login/?next=/WhereYouCameFrom' in resp.url. We're too lazy to write all this. Usually there are not any other views in the project also called login.

def test_superuser:

"auth.User" is baked in user model that comes with Django. It lives in the "auth" app. Model name is user. It lives inside the Django framework, so we can use it even though we haven't written it ourselves.



And when you run your tests, you should see that you still need to create the "**AdminView**" view.

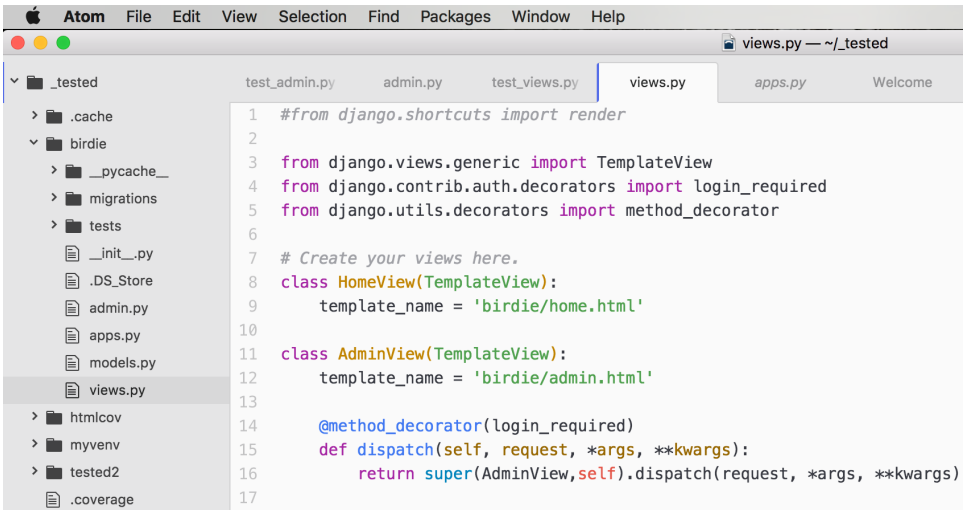


2. Implement the **AdminView** view and run the tests again

Add the following code to the birdie/**views.py**

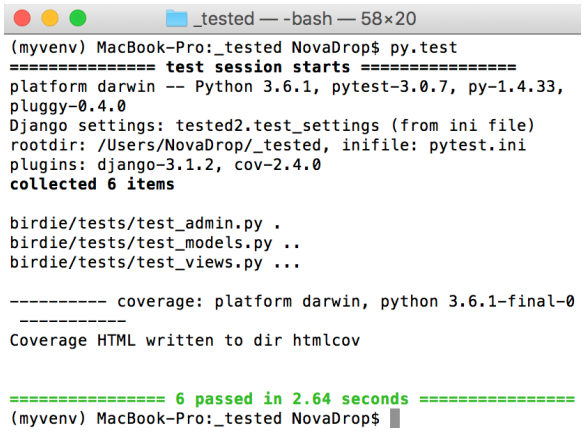
```
#views.py

from django.contrib.auth.decorators import login_required
from django.utils.decorators import method_decorator

class AdminView(TemplateView):
   template_name = 'birdie/admin.html'

   @method_decorator(login_required)
   def dispatch(self, request, *args, **kwargs):
      return super(AdminView,self).dispatch(request, *args, **kwargs)
```

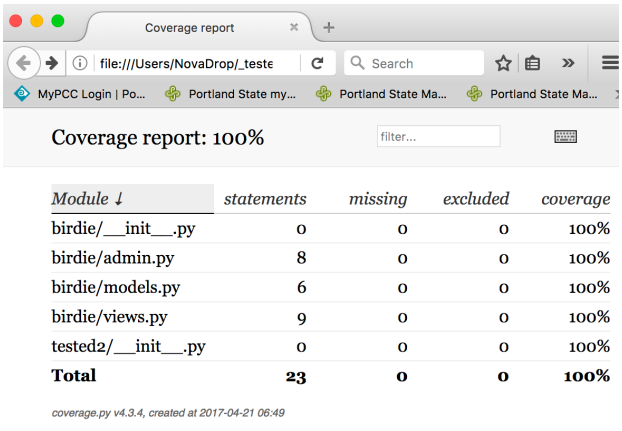
← If you forget where these things
← live, Google “Django login required”



Note:

Your tests should now pass. – Don’t forget to check your test coverage. (Should be 100%)





Testing Forms:

- When you implement the form, step by step, you will see various test errors
  - ImportError: cannot import name 'forms'
  - AttributeError: 'module' object has no attribute 'PostForm'
  - ImproperlyConfigured: Creating a ModelForm without either the 'fields' attribute or the 'exclude' attribute is prohibited; form Postform needs updating
  - AssertionError: Should be invalid if body text is less than 10 characters
- The guide you toward your final goal.

Let's say we have a site where there's a text field and people can enter a message, press a submit button, and a new post is added into our database, BUT your message has to be longer than 10 characters. We need a form for this.

1. Create a birdie/tests/**test_forms.py** file

   In terminal:

   (myvenv) ~$ touch birdie/tests/test_forms.py

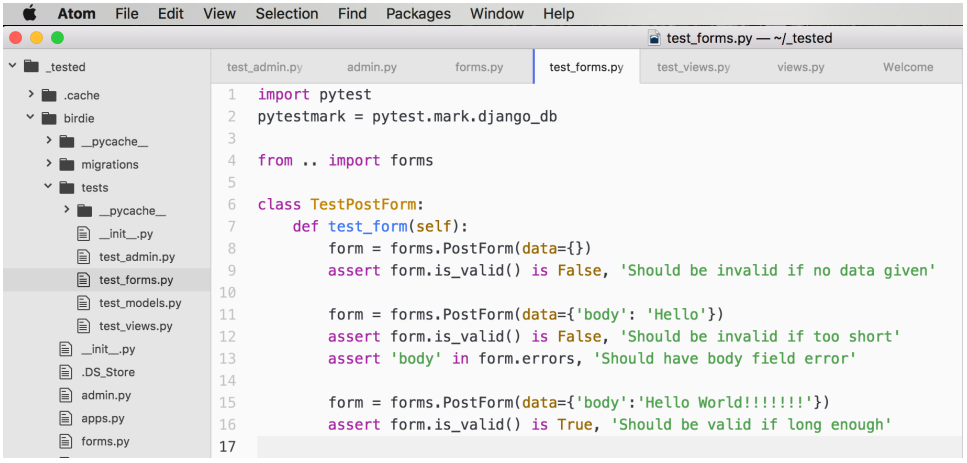
2. Place code in birdie/tests/**test_forms.py** file

```
#test_forms.py

import pytest
pytestmark = pytest.mark.django_db

from .. import forms

class TestPostForm:
    def test_form(self):
        form = forms.PostForm(data={})
        assert form.is_valid() is False, 'Should be invalid if no data given'

        form = forms.PostForm(data={'body': 'Hello'})
        assert form.is_valid() is False, 'Should be invalid if too short'
        assert 'body' in form.errors, 'Should have body field error'

        form = forms.PostForm(data={'body':'Hello World!!!!!!!'})
        assert form.is_valid() is True, 'Should be valid if long enough'
```

Code Line Explanation:

assert 'body' in form.errors, 'Should have body field error'

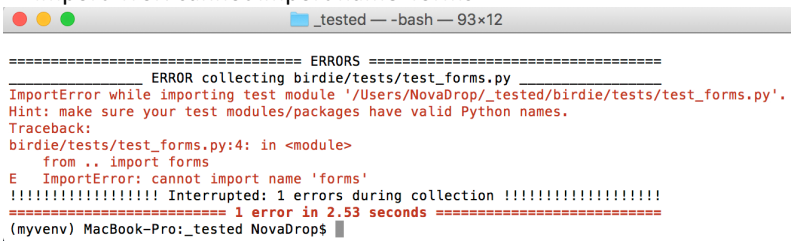
We know there will be an error, but we want to make sure it's because of the 'body' field. -What if it's invalid because of something else? There is an errors dictionary, a list, with every field name that has an error and then a list of errors for that field name. So we will expect that body which is a field name is in this dictionary. There's a key in this dictionary with 'body' because that field has an error.



Note: Passing badly formed dictionary here caused problems later. Originally the presenter left out the dictionary key "' body': " when writing values 'Hello' and 'Hello!!!!!!!' – See 3(d) later in this section.

If you run your tests:

E ImportError: cannot import name 'forms'



Which is correct, because there's no such file.

3. Create birdie/**forms.py** file

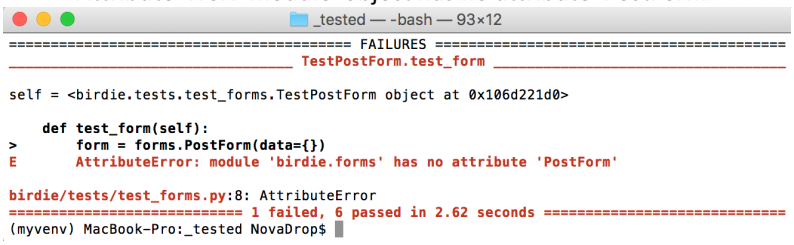
   a. In terminal

      (myvenv) ~$ touch birdie/forms.py

   b. Place following code in birdie/**forms.py** file

```
#forms.py

from django import forms
```

E AttributeError: 'module' object has no attribute 'PostForm'



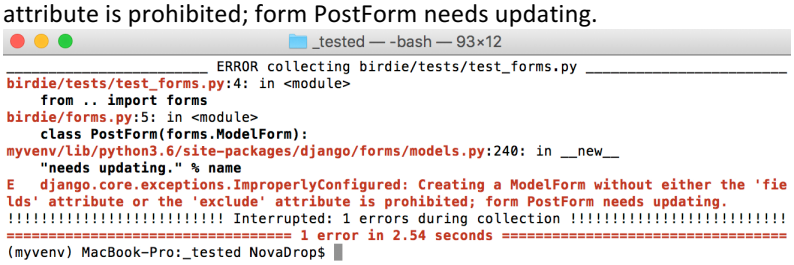
In the next few subsections, we show the errors that can be generated during this procedure and what they mean. There is also an unexpected error cited here from poorly coded dictionary keys/values during creating the test_forms.py file. If you would like to jump straight to the correct code, go to the end of 3(e) of this section.

   c. Add the following code to create the attribute in the birdie/**forms.py** file

```
from . import models

class PostForm(forms.ModelForm):
    class Meta:
        model = models.Post
```

This should be enough, but it's not.

E ImproperlyConfigured: Creating a ModelForm without either the 'fields' attribute or the 'exclude' attribute is prohibited; form PostForm needs updating.

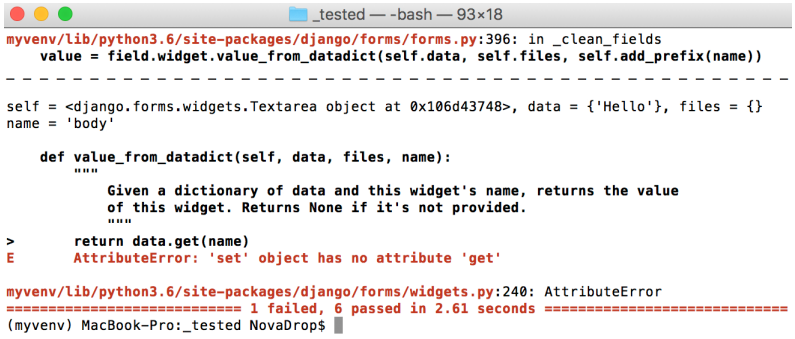


d. Add the following code to the “class PostForm” section in the birdie/**forms.py** file

```
fields = ('body',)
```

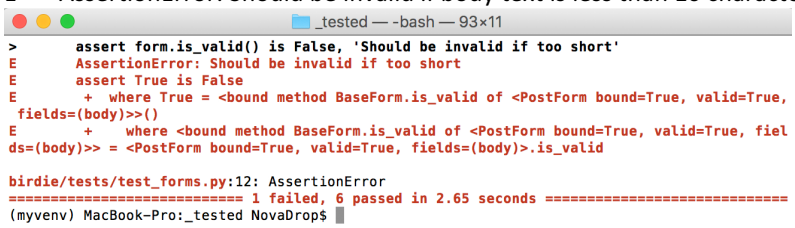
This is the error generated from passing poorly coded dictionary keys/values from section 2. It took this long to show up. It threw the presenter. He said this was not a helpful error message. I am including it as it is a valuable example.

Error generated from a mistake in part 2 of this section:



After adding the above code, we get the following error:

E AssertionError: Should be invalid if body text is less than 10 characters



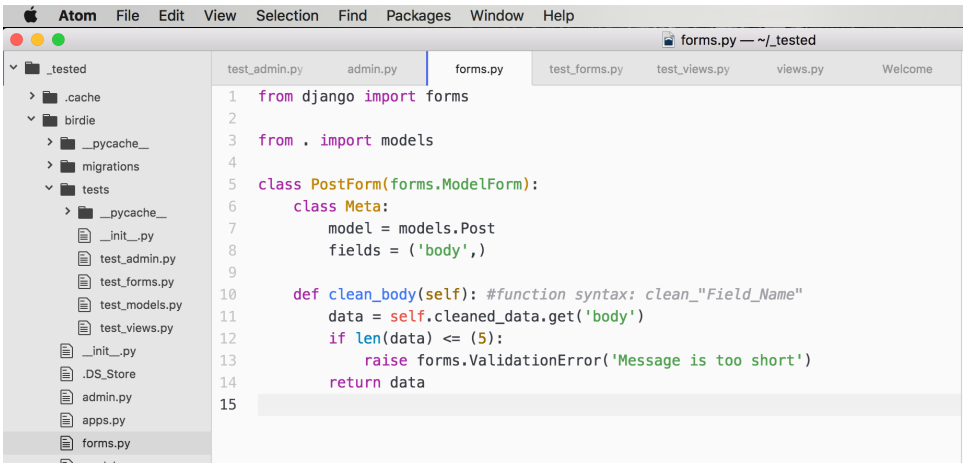
At the moment, the form is always valid. We haven’t implemented the restriction of 10 characters minimum, yet.

e. Add the following code at the end in the birdie/**forms.py** file

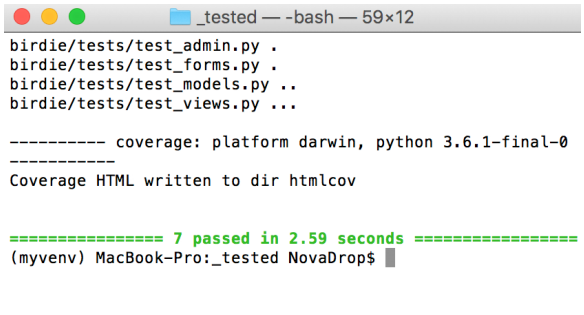
```
def clean_body(self): #function syntax: clean_"Field_Name"
    data = self.cleaned_data.get('body')
    if len(data) <= (5):
        raise forms.ValidationError('Message is too short')
    return data
```

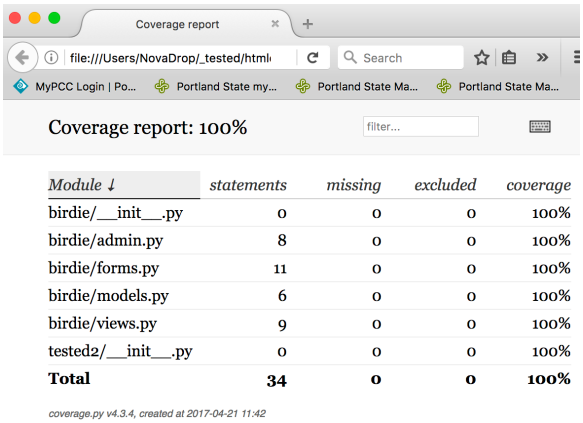
Final, correct code: birdie/**forms.py** file

```
#form.py

from django import forms

from . import models

class PostForm(forms.ModelForm):
    class Meta:
        model = models.Post
        fields = ('body',)

    def clean_body(self):
        data = self.cleaned_data.get('body')
        if len(data) <= (5):
            raise forms.ValidationError('Message is too short')
        return data
```



Our tests pass again and we have 100% coverage.





Testing Post Requests:

- We want to create a view that uses PostForm to update a Post
- Testing POST requests works in the same way like GET requests
- The next example also shows how to pass POST data into the view and how to pass URL kwargs into the view.

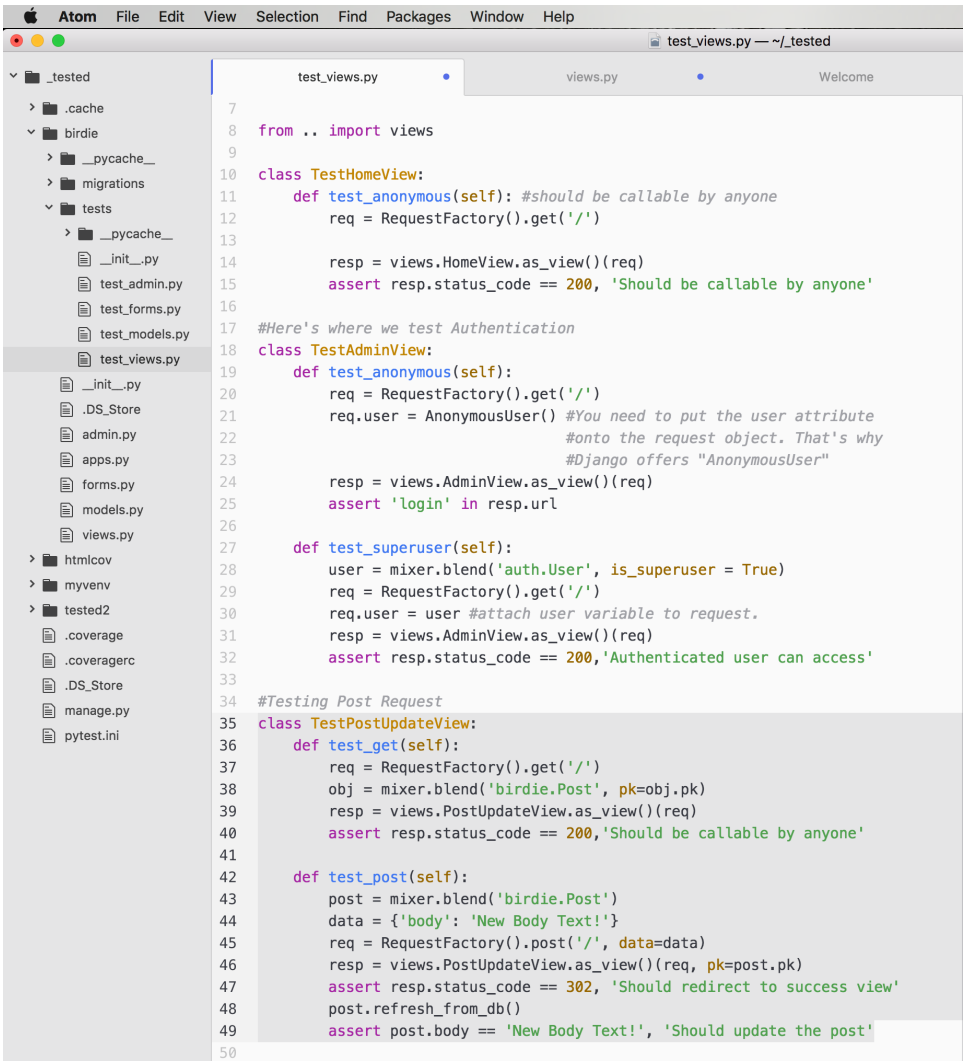
We did GET requests. Now we want to POST requests. So, we have a form. We already know that we are able to save stuff to the database. We need to create a view that is using that form, so we can go to a url, see that form, press a submit button. There has to be a view for this.

First you have to make a GET request, so you can see the form. Then you click the button and you finally make a POST request. We have to test both.

1. Create the test for PostUpdateView class

Add the following code to the birdie/tests/**test_views.py**

```
#test_views.py

class TestPostUpdateView:
   def test_get(self):
      req = RequestFactory().get('/')
      obj = mixer.blend('birdie.Post')
      resp = views.PostUpdateView.as_view()(req, pk=obj.pk)
      assert resp.status_code == 200,'Should be callable by anyone'

   def test_post(self):
      post = mixer.blend('birdie.Post')
      data = {'body': 'New Body Text!'}
      req = RequestFactory().post('/', data=data)
      resp = views.PostUpdateView.as_view()(req, pk=post.pk)
      assert resp.status_code == 302, 'Should redirect to success view'
      post.refresh_from_db()
      assert post.body == 'New Body Text!', 'Should update the post'
```



What this code means

```
#test_views.py

class TestPostUpdateView:
    def test_get(self):
        req = RequestFactory().get('/')
        obj = mixer.blend('birdie.Post')
        resp = views.PostUpdateView.as_view()(req, pk=obj.pk)
        assert resp.status_code == 200,'Should be callable by anyone'

    def test_post(self):
        post = mixer.blend('birdie.Post')
        data = {'body': 'New Body Text!'}
        req = RequestFactory().post('/', data=data)
        resp = views.PostUpdateView.as_view()(req, pk=post.pk)
        assert resp.status_code == 302, 'Should redirect to success view'
        post.refresh_from_db()
        assert post.body == 'New Body Text!', 'Should update the post'
```

This view is supposed to edit an already existing Post, so the url will be something like post/5 (5 being the primary key of the post). How do you call a view like that? That means we need mixer, we need to create a Post, and under the response, when we call the view, we will pass in the primary key as url kwarg and it's going to be the primary key of that object that already exists in the database.

Since this is an update view we need an object.

The user will type something into the field, so this view needs some data.

Here is the trick.

The primary key is passed in here when we call the dispatch function of the view.
The data is attached to the post request
- This makes sense because if you send GET or POST requests, they might have GET or POST data.

Then we try to call that view.
Django update views are built in such a way that if the post is successful it will redirect you to a "success" url. So we're testing for 302 redirect status code

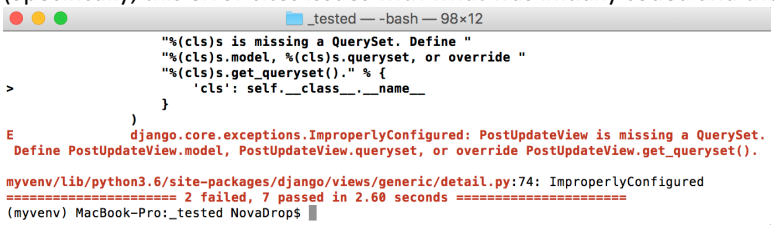
Then you can call refresh from db
because when we use mixer, the body will be some random text, because we didn't provide a special message like post = mixer.blend('birdie.Post', 'body' = 'Hello')

But the post data is something we know
data = {'body': 'New Body Text!'}

After the post request we want the database object to be updated, so the new text should be inside the database. So we refresh our object and then we compare the body with text that we expect.

assert post.body == 'New Body Text!', 'Should update the post'

And again, our tests will fail, because we need to create the view for **PostUpdateView**
(Specifically, this error cites issues with what was initially coded and then abandoned to work on the test code.)



2. Create a class for the PostUpdateView view

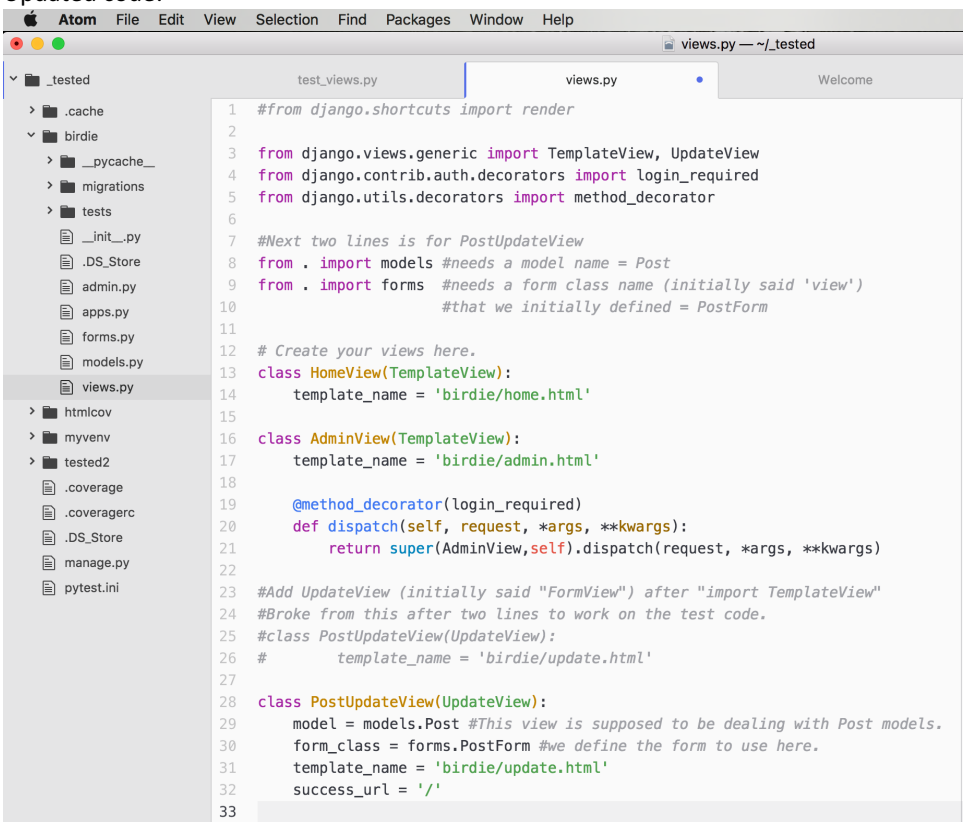
Add the following code to the birdie/**views.py** file

```
#views.py

from . import models
from . import forms


class PostUpdateView(UpdateView):
    model = models.Post
    form_class = forms.PostForm
    template_name = 'birdie/update.html'
    success_url = '/'
```

← With just this few lines of code Django will be
← smart enough to see a url that has a primary key
← inside and use it and try to get a Post object With that pk. If it doesn't exist, it will show a 404 page. If it does exist it will render the PostForm. If you make a Post Request , press the submit button, your data will be passed into that PostForm. If the form is valid it will call "save" on the form and redirect you to the success url.

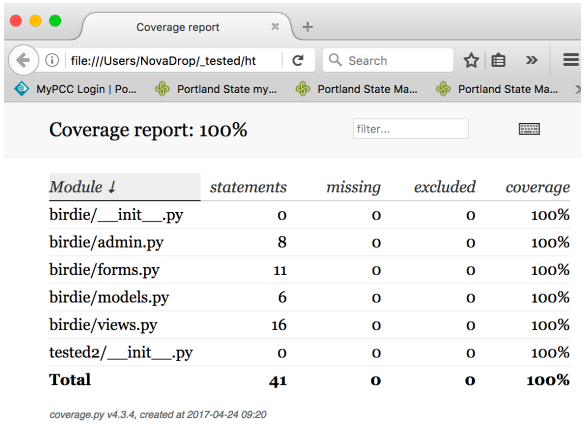
Let's just say it goes back to the home view.

Updated code:



Our tests should pass now and we should have 100% coverage

```
_tested — -bash — 70×12

birdie/tests/test_admin.py .
birdie/tests/test_forms.py .
birdie/tests/test_models.py ..
birdie/tests/test_views.py .....

---------- coverage: platform darwin, python 3.6.1-final-0 -----------
Coverage HTML written to dir htmlcov


====================== 9 passed in 2.58 seconds ======================
(myvenv) MacBook-Pro:_tested NovaDrop$
```



Coverage report

file:///Users/NovaDrop/_tested/ht

Search

MyPCC Login | Po... Portland State my... Portland State Ma... Portland State Ma...

Coverage report: 100%

filter...

Module ↓	statements	missing	excluded	coverage
birdie/__init__.py	0	0	0	100%
birdie/admin.py	8	0	0	100%
birdie/forms.py	11	0	0	100%
birdie/models.py	6	0	0	100%
birdie/views.py	16	0	0	100%
tested2/__init__.py	0	0	0	100%
Total	**41**	**0**	**0**	**100%**

coverage.py v4.3.4, created at 2017-04-24 09:20

Note: We're not testing templates. We are most interested in security (e.g. Can users access other users objects by changing the primary key in the url?) and certain form validations (e.g. Can users submit half empty forms etc.?)

Testing 404 Errors:

- Your views will often raise 404 errors
- Unfortunately, they are exceptions and they bubble up all the way into your tests, so you cannot simply check "**assert resp.status_code ==404**"
- Instead you have to execute the view inside a "with-statement"

404 errors are a little trickier to test because Django raises a 404 exception. This is why they bubble up into your tests. Your tests will actually crash, show you the exception, and not move on. We want this exception to happen, because it means our tests were successful. It shouldn't crash. So, there is a way, in your tests, to catch exceptions.

1. Add the following code to the "**class TestPostUpdateView**" section of the birdie/test/**test_views.py** file

   A silly example: We want to make sure, if there's a logged in user trying to use this update form and the user's first name is "Martin", then it's not allowed. It will throw a 404 error. This will be the functionality of our view.

```
#test_views.py

from django.http import Http404

    def test_security(self):
        user = mixer.blend('auth.User', first_name='Martin')
        post = mixer.blend('birdie.Post')
        req = RequestFactory().post('/', data={})
        req.user = user
        with pytest.raises(Http404):
            views.PostUpdateView.as_view()(req, pk=post.pk)
```

← Add at top of file

← Create a user with first name field "Martin"
← Create a post to update a post
← Create a RequestFactory (data irrelevant here)
← Attach user object to request
← with statement is part of Python language. Pytest has raises function and here we can add from django.http import Http 404. So this is the exception name.

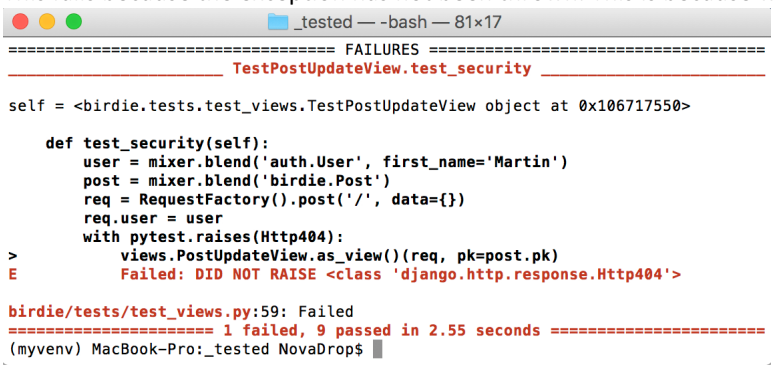
You can add any kind of exception in Python. We could do something like this:

```
with pytest.raises(Exception):
```

Where we expect any generic exception to happen. "Http404" is one exception class that's part of the Django framework. Inside the with statement is code we execute that we expect should throw the exception.

Code added is highlighted below:

This fails because the exception has not been thrown. This is because we haven't modified our view code.

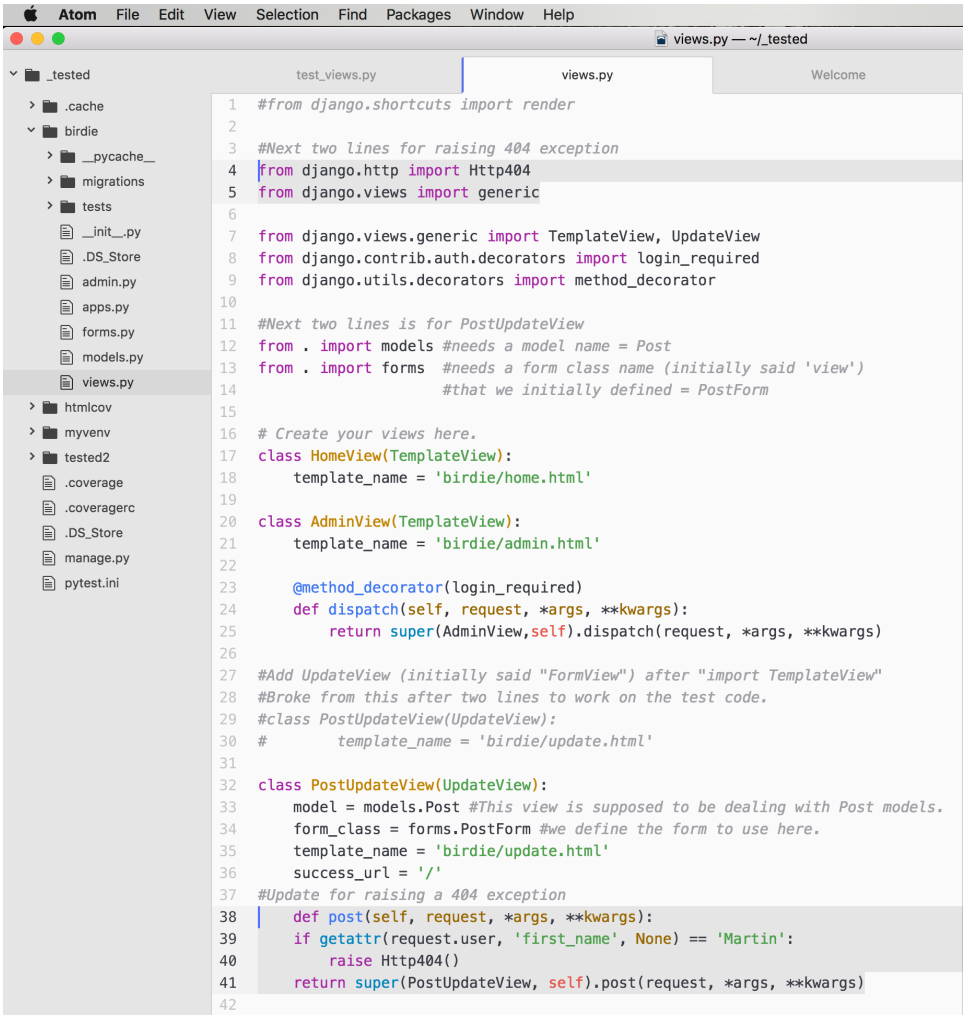


2. Update your implementation

Add the following code to the birdie/**views.py**

#views.py
 	
from django.http import Http404
from django.views import generic
 	
#Update for raising a 404 exception
def post(self, request, *args, **kwargs):
if getattr(request.user, 'first_name', None) == 'Martin':
raise Http404()
return super(PostUpdateView, self).post(request, *args, **kwargs)
	At some point, the dispatch function will figure out this is a post request
	← and then it will call the post function
	← Here we are trying to get the user from the request, access the first name and if it is "Martin" we're going to raise a 404 exception.

Code added is highlighted below



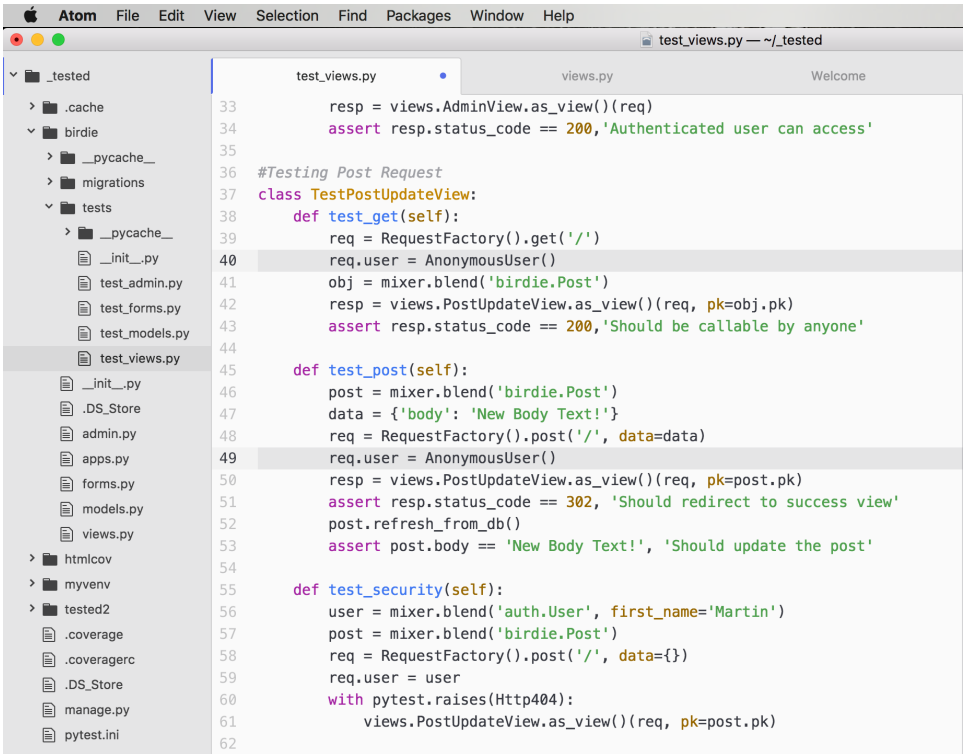
It should work, but now we are getting an error because we added some functionality that makes our older tests incompatible.

```
_tested — -bash — 81×29
=================================== FAILURES ===================================
_____________________ TestPostUpdateView.test_post _____________________

self = <birdie.tests.test_views.TestPostUpdateView object at 0x106dd7b00>

    def test_post(self):
        post = mixer.blend('birdie.Post')
        data = {'body': 'New Body Text!'}
        req = RequestFactory().post('/', data=data)
>       resp = views.PostUpdateView.as_view()(req, pk=post.pk)

birdie/tests/test_views.py:48:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
myvenv/lib/python3.6/site-packages/django/views/generic/base.py:68: in view
    return self.dispatch(request, *args, **kwargs)
myvenv/lib/python3.6/site-packages/django/views/generic/base.py:88: in dispatch
    return handler(request, *args, **kwargs)
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

self = <birdie.views.PostUpdateView object at 0x106aad908>
request = <WSGIRequest: POST '/'>, args = (), kwargs = {'pk': 1}

    def post(self, request, *args, **kwargs):
>       if getattr(request.user, 'first_name', None) == 'Martin':
E       AttributeError: 'WSGIRequest' object has no attribute 'user'

birdie/views.py:39: AttributeError
====================== 1 failed, 9 passed in 2.57 seconds ======================
(myvenv) MacBook-Pro:_tested NovaDrop$
```

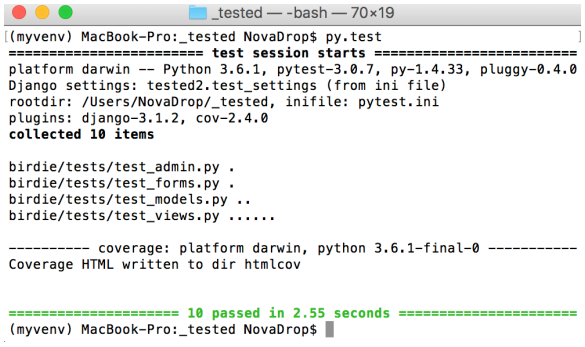
This is because now we are trying to access the user attribute on the request, which we have not done before. But we have written tests before here where we did not add the user object to the request. So, in this case, we'll just use the "AnonymousUser()"

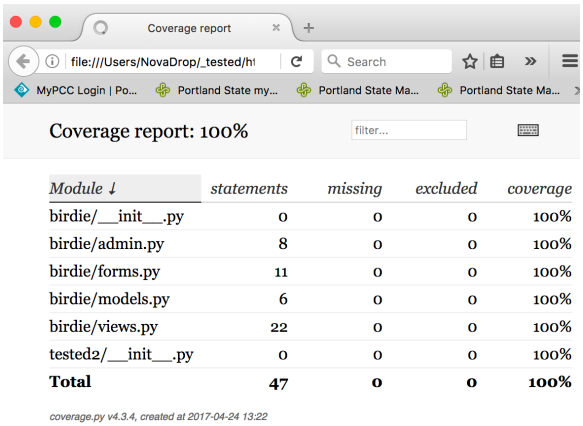
3. Update birdie/tests/**test_views.py** file "**class TestUpdateView**" functions with "**req.user = AnonymousUser()**"

Code shown highlighted



Tests should now pass and coverage should be 100%:





Mocking Requests:

Not doing this last section. It is for simulating credit card purchases; outside the scope of this project.