Previous Next

Unit 3 / Lesson 5 / Assignment 2

Testing web applications

Estimated Time: 2-3 hours

In this lesson you are going to look at different ways to test web applications. For the first assignment you'll look at three different approaches to testing, using the blog from earlier on as a starting point.

Unit testing

So far in the course you have primarily looked at unit testing. In unit testing you make sure that small, self-contained sections of code work in isolation. You should be pretty familiar with how this works, but just as a quick refresher, set up a unit test to make sure that the `dateformat` filter you added is working correctly.

This function is a good candidate for unit testing as it is small and doesn't rely on a other services (for example a running server, or a database). So let's write a couple of simple unit tests to make sure it is working correctly. To get started `cd` into your blog project directory and activate the virtualenv. Then create a new folder called *tests* which will hold the tests.

Next you can create a subclass of `unittest.TestCase` to hold the tests. In *tests/test_filter.py* add the following code:

```
import os
import unittest
```

```
import datetime

# Configure your app to use the testing configuration
if not "CONFIG_PATH" in os.environ:
    os.environ["CONFIG_PATH"] = "blog.config.TestingConfig"

import blog
from blog.filters import *

class FilterTests(unittest.TestCase):
    pass

if __name__ == "__main__":
    unittest.main()
```

The body of the code creates the `FilterTests` class to hold your tests, and then runs the tests in the main block. The one new idea here is that you set the `CONFIG_PATH` environment variable to point to a `TestingConfig` class. You might remember how in the *__init__.py* file you configured the Flask app based upon the value held in this environment variable. So next you need to create the `TestingConfig` class in the *config.py* file:

```
class TestingConfig(object):
    SQLALCHEMY_DATABASE_URI = "postgresql://ubuntu:thinkful@localhost:
    DEBUG = False
    SECRET_KEY = "Not secret"
```

Here you have a separate database URI for testing, and you use a different secret key. You also turn off the debug setting. This means that you will be testing exactly what your clients will see in production without the additional debugging information (which isn't helpful within a testing environment). Go ahead and create the new test database by running `createdb blogful-test`. (Don't forget that connection errors on port 5432 can usually be solved with `sudo service postgresql start`.)

Now, add a couple of tests to make sure that the `dateformat` function is working correctly:

```
class FilterTests(unittest.TestCase):
    def test_date_format(self):
        # Tonight we're gonna party...
        date = datetime.date(1999, 12, 31)
        formatted = dateformat(date, "%y/%m/%d")
        self.assertEqual(formatted, "99/12/31")

    def test_date_format_none(self):
        formatted = dateformat(None, "%y/%m/%d")
        self.assertEqual(formatted, None)
```

The first test creates a `datetime.date` object, runs it through the `dateformat` function and makes sure that the resulting string is correct. The second test passes `None` into the function, and makes sure that you get a `None` object back in return.

Try running your tests using `PYTHONPATH=. python tests/test_filters.py`. You should see that the tests pass fine. Notice how when you run your tests you have to set the `PYTHONPATH` environment variable. This is so the tests can import the `blog` module correctly, even though it is in a different location to the test files.

Integration testing

The next type of testing is integration testing. For this you are going to look at how to test the `add_entry_post` view. Now your view is definitely pretty small – it's around 10 lines of code with a generous allocation of whitespace. So why are you writing an integration test rather than a unit test?

The code is relying on a few relatively large subsystems and services. It touches the ORM, which works with the database, and it also works hand-in-hand with the login system. If you were going to write a unit test in the strict definition of the term you would need to eliminate your dependency on these subsystems.

Generally you'll do this by mocking, creating simple code which mimics the functionality of the subsystems. But in this case mocking the database and login system seems like a lot of work for little tangible reward; your tests may run slightly faster but you will have another 100 lines of code to maintain.

The solution is to create integration tests. These are tests that make sure the various subsystems are working together correctly. In practical terms you will find little difference between writing integration tests and unit tests; you just have to do a little more work to set up your testing environment.

So let's get started writing an integration test for the `add_entry_post` view. In *tests/test_views_integration.py* add the following class:

```
import os
import unittest
from urllib.parse import urlparse

from werkzeug.security import generate_password_hash

# Configure your app to use the testing database
os.environ["CONFIG_PATH"] = "blog.config.TestingConfig"

from blog import app
from blog.database import Base, engine, session, User, Entry

class TestViews(unittest.TestCase):
    def setUp(self):
        """ Test setup """
        self.client = app.test_client()

        # Set up the tables in the database
        Base.metadata.create_all(engine)

        # Create an example user
        self.user = User(name="Alice", email="alice@example.com",
                         password=generate_password_hash("test"))
        session.add(self.user)
        session.commit()

    def tearDown(self):
        """ Test teardown """
```

```
        session.close()
        # Remove the tables and their data from the database
        Base.metadata.drop_all(engine)

if __name__ == "__main__":
    unittest.main()
```

Here you use the environment variable trick to use the testing configuration. You then create a test client using the `app.test_client` function. This will allow you to make requests to views and inspect the responses you get from the app. Next you call the `Base.metadata.create_all` function to create tables in the test database. Finally you create an example user and add it to the database. You'll use the user to log in and be the author of the test entry.

Now, in the `TestViews` class, add a method to simulate logging in, and a test which attempts to add an entry.

```
    def simulate_login(self):
        with self.client.session_transaction() as http_session:
            http_session["user_id"] = str(self.user.id)
            http_session["_fresh"] = True

    def test_add_entry(self):
        self.simulate_login()

        response = self.client.post("/entry/add", data={
            "title": "Test Entry",
            "content": "Test content"
        })

        self.assertEqual(response.status_code, 302)
        self.assertEqual(urlparse(response.location).path, "/")
        entries = session.query(Entry).all()
        self.assertEqual(len(entries), 1)

        entry = entries[0]
        self.assertEqual(entry.title, "Test Entry")
        self.assertEqual(entry.content, "Test content")
        self.assertEqual(entry.author, self.user)
```

The `simulate_login` method essentially mimics what Flask-Login looks for when determining whether a user is logged in. You use the `self.client.session_transaction` method to get access to a variable representing the HTTP session. You then add two variables to this session: the id of the user and a variable which tells Flask-Login that the session is still active (`_fresh`).

In the `test_add_entry` method you call your `simulate_login` method so you can act as a logged in user. Then you send a POST request to `/entry/add` using the `self.client.post` method. You use the `data` parameter to provide the form data for an example entry.

Next you start to check that the response from the app looks correct. Make sure that your user is being redirected to the `/` route by checking the status code and location header of the response.

Then check to make sure that the entry has been added to the database correctly. Look to see that only one entry has been added, and make sure that the title, content and author are set to the correct values.

Try running the test using `PYTHONPATH=. python tests/test_views_integration.py` . You should see that the entry is correctly added to the database.

Acceptance testing

The final type of testing is acceptance testing. Unit and integration tests tend to be very developer-centric in their view of the world. They are designed to make sure that the code you write does what it says it does. Acceptance testing is a different slant on how to test software. Acceptance tests are designed to make sure that your code does what your users expect it to do.