Test Coverage

Writing unit tests for your application lets you check that the code you wrote works the way you expect. Flask provides a test client that simulates requests to the application and returns the response data.

You should test as much of your code as possible. Code in functions only runs when the function is called, and code in branches, such as `if` blocks, only runs when the condition is met. You want to make sure that each function is tested with data that covers each branch.

The closer you get to 100% coverage, the more comfortable you can be that making a change won't unexpectedly change other behavior. However, 100% coverage doesn't guarantee that your application doesn't have bugs. In particular, it doesn't test how the user interacts with the application in the browser. Despite this, test coverage is an important tool to use during development.

Note:

This is being introduced late in the tutorial, but in your future projects you should test as you develop.

You'll use pytest and coverage to test and measure your code. Install them both:

```
$ pip install pytest coverage
```

Setup and Fixtures

The test code is located in the `tests` directory. This directory is *next to* the `flaskr` package, not inside it. The `tests/conftest.py` file contains setup functions called *fixtures* that each test will use. Tests are in Python modules that start with `test_`, and each test function in those modules also starts with `test_`.

Each test will create a new temporary database file and populate some data that will be used in the tests. Write a SQL file to insert that data.

`tests/data.sql`

```sql
INSERT INTO user (username, password)
VALUES
  ('test', 'pbkdf2:sha256:50000$TCI4GzcX$0de171a4f4dac32e3364c7ddc7c14f
  ('other', 'pbkdf2:sha256:50000$kJPKsz6N$d2d4784f1b030a9761f5ccaeeaca4
INSERT INTO post (title, body, author_id, created)
VALUES
  ('test title', 'test' || x'0a' || 'body', 1, '2018-01-01 00:00:00');
```

The `app` fixture will call the factory and pass `test_config` to configure the application and database for testing instead of using your local development configuration.

tests/conftest.py

```
import os
import tempfile

import pytest
from flaskr import create_app
from flaskr.db import get_db, init_db

with open(os.path.join(os.path.dirname(__file__), 'data.sql'), 'rb') as
    _data_sql = f.read().decode('utf8')


@pytest.fixture
def app():
    db_fd, db_path = tempfile.mkstemp()

    app = create_app({
        'TESTING': True,
        'DATABASE': db_path,
    })

    with app.app_context():
        init_db()
        get_db().executescript(_data_sql)

    yield app

    os.close(db_fd)
    os.unlink(db_path)


@pytest.fixture
def client(app):
    return app.test_client()


@pytest.fixture
```

```
def runner(app):
    return app.test_cli_runner()
```

`tempfile.mkstemp()` creates and opens a temporary file, returning the file object and the path to it. The `DATABASE` path is overridden so it points to this temporary path instead of the instance folder. After setting the path, the database tables are created and the test data is inserted. After the test is over, the temporary file is closed and removed.

`TESTING` tells Flask that the app is in test mode. Flask changes some internal behavior so it's easier to test, and other extensions can also use the flag to make testing them easier.

The `client` fixture calls **`app.test_client()`** with the application object created by the `app` fixture. Tests will use the client to make requests to the application without running the server.

The `runner` fixture is similar to `client`. **`app.test_cli_runner()`** creates a runner that can call the Click commands registered with the application.

Pytest uses fixtures by matching their function names with the names of arguments in the test functions. For example, the `test_hello` function you'll write next takes a `client` argument. Pytest matches that with the `client` fixture function, calls it, and passes the returned value to the test function.

Factory

There's not much to test about the factory itself. Most of the code will be executed for each test already, so if something fails the other tests will notice.

The only behavior that can change is passing test config. If config is not passed, there should be some default configuration, otherwise the configuration should be overridden.

`tests/test_factory.py`

```
from flaskr import create_app


def test_config():
    assert not create_app().testing
    assert create_app({'TESTING': True}).testing


def test_hello(client):
    response = client.get('/hello')
    assert response.data == b'Hello, World!'
```

You added the `hello` route as an example when writing the factory at the beginning of the tutorial. It returns “Hello, World!”, so the test checks that the response data matches.

Database

Within an application context, `get_db` should return the same connection each time it’s called. After the context, the connection should be closed.

tests/test_db.py

```
import sqlite3

import pytest
from flaskr.db import get_db


def test_get_close_db(app):
    with app.app_context():
        db = get_db()
        assert db is get_db()

    with pytest.raises(sqlite3.ProgrammingError) as e:
        db.execute('SELECT 1')

    assert 'closed' in str(e.value)
```

The `init-db` command should call the `init_db` function and output a message.

tests/test_db.py

```
def test_init_db_command(runner, monkeypatch):
    class Recorder(object):
        called = False

    def fake_init_db():
        Recorder.called = True

    monkeypatch.setattr('flaskr.db.init_db', fake_init_db)
    result = runner.invoke(args=['init-db'])
    assert 'Initialized' in result.output
    assert Recorder.called
```

This test uses Pytest’s `monkeypatch` fixture to replace the `init_db` function with one that records that it’s been called. The `runner` fixture you wrote above is used to call the `init-`

db command by name.

Authentication

For most of the views, a user needs to be logged in. The easiest way to do this in tests is to make a POST request to the login view with the client. Rather than writing that out every time, you can write a class with methods to do that, and use a fixture to pass it the client for each test.

tests/conftest.py

```
class AuthActions(object):
    def __init__(self, client):
        self._client = client

    def login(self, username='test', password='test'):
        return self._client.post(
            '/auth/login',
            data={'username': username, 'password': password}
        )

    def logout(self):
        return self._client.get('/auth/logout')


@pytest.fixture
def auth(client):
    return AuthActions(client)
```

With the auth fixture, you can call auth.login() in a test to log in as the test user, which was inserted as part of the test data in the app fixture.

The register view should render successfully on GET. On POST with valid form data, it should redirect to the login URL and the user's data should be in the database. Invalid data should display error messages.

tests/test_auth.py

```
import pytest
from flask import g, session
from flaskr.db import get_db


def test_register(client, app):
    assert client.get('/auth/register').status_code == 200
    response = client.post(
        '/auth/register', data={'username': 'a', 'password': 'a'}
    )
    assert 'http://localhost/auth/login' == response.headers['Location'

    with app.app_context():
        assert get_db().execute(
            "select * from user where username = 'a'",
        ).fetchone() is not None


@pytest.mark.parametrize(('username', 'password', 'message'), (
    ('', '', b'Username is required.'),
    ('a', '', b'Password is required.'),
    ('test', 'test', b'already registered'),
))
def test_register_validate_input(client, username, password, message):
    response = client.post(
        '/auth/register',
        data={'username': username, 'password': password}
    )
    assert message in response.data
```

`client.get()` makes a `GET` request and returns the **`Response`** object returned by Flask. Similarly, **`client.post()`** makes a `POST` request, converting the `data` dict into form data.

To test that the page renders successfully, a simple request is made and checked for a `200 OK` **`status_code`**. If rendering failed, Flask would return a `500 Internal Server Error` code.

`headers` will have a `Location` header with the login URL when the register view redirects to the login view.

`data` contains the body of the response as bytes. If you expect a certain value to render on the page, check that it's in `data`. Bytes must be compared to bytes. If you want to compare Unicode text, use **`get_data(as_text=True)`** instead.

`pytest.mark.parametrize` tells Pytest to run the same test function with different arguments. You use it here to test different invalid input and error messages without writing the same code three times.

The tests for the `login` view are very similar to those for `register`. Rather than testing the data in the database, **`session`** should have `user_id` set after logging in.

`tests/test_auth.py`

```
def test_login(client, auth):
    assert client.get('/auth/login').status_code == 200
    response = auth.login()
    assert response.headers['Location'] == 'http://localhost/'

    with client:
        client.get('/')
        assert session['user_id'] == 1
        assert g.user['username'] == 'test'


@pytest.mark.parametrize(('username', 'password', 'message'), (
    ('a', 'test', b'Incorrect username.'),
    ('test', 'a', b'Incorrect password.'),
))
def test_login_validate_input(auth, username, password, message):
    response = auth.login(username, password)
    assert message in response.data
```

Using `client` in a `with` block allows accessing context variables such as **`session`** after the response is returned. Normally, accessing `session` outside of a request would raise an error.

Testing `logout` is the opposite of `login`. **`session`** should not contain `user_id` after logging out.

`tests/test_auth.py`

```
def test_logout(client, auth):
    auth.login()

    with client:
        auth.logout()
        assert 'user_id' not in session
```

Blog

All the blog views use the `auth` fixture you wrote earlier. Call `auth.login()` and subsequent requests from the client will be logged in as the `test` user.

The `index` view should display information about the post that was added with the test data. When logged in as the author, there should be a link to edit the post.

You can also test some more authentication behavior while testing the `index` view. When not logged in, each page shows links to log in or register. When logged in, there's a link to log out.

tests/test_blog.py

```
import pytest
from flaskr.db import get_db


def test_index(client, auth):
    response = client.get('/')
    assert b"Log In" in response.data
    assert b"Register" in response.data

    auth.login()
    response = client.get('/')
    assert b'Log Out' in response.data
    assert b'test title' in response.data
    assert b'by test on 2018-01-01' in response.data
    assert b'test\nbody' in response.data
    assert b'href="/1/update"' in response.data
```

A user must be logged in to access the `create`, `update`, and `delete` views. The logged in user must be the author of the post to access `update` and `delete`, otherwise a `403 Forbidden` status is returned. If a `post` with the given `id` doesn't exist, `update` and `delete` should return `404 Not Found`.

tests/test_blog.py

```
@pytest.mark.parametrize('path', (
    '/create',
    '/1/update',
    '/1/delete',
))
def test_login_required(client, path):
    response = client.post(path)
    assert response.headers['Location'] == 'http://localhost/auth/login


def test_author_required(app, client, auth):
    # change the post author to another user
    with app.app_context():
        db = get_db()
        db.execute('UPDATE post SET author_id = 2 WHERE id = 1')
        db.commit()

    auth.login()
    # current user can't modify other user's post
```

```python
    assert client.post('/1/update').status_code == 403
    assert client.post('/1/delete').status_code == 403
    # current user doesn't see edit link
    assert b'href="/1/update"' not in client.get('/').data


@pytest.mark.parametrize('path', (
    '/2/update',
    '/2/delete',
))
def test_exists_required(client, auth, path):
    auth.login()
    assert client.post(path).status_code == 404
```

The `create` and `update` views should render and return a `200 OK` status for a `GET` request. When valid data is sent in a `POST` request, `create` should insert the new post data into the database, and `update` should modify the existing data. Both pages should show an error message on invalid data.

tests/test_blog.py

```python
def test_create(client, auth, app):
    auth.login()
    assert client.get('/create').status_code == 200
    client.post('/create', data={'title': 'created', 'body': ''})

    with app.app_context():
        db = get_db()
        count = db.execute('SELECT COUNT(id) FROM post').fetchone()[0]
        assert count == 2


def test_update(client, auth, app):
    auth.login()
    assert client.get('/1/update').status_code == 200
    client.post('/1/update', data={'title': 'updated', 'body': ''})

    with app.app_context():
        db = get_db()
        post = db.execute('SELECT * FROM post WHERE id = 1').fetchone()
        assert post['title'] == 'updated'


@pytest.mark.parametrize('path', (
    '/create',
    '/1/update',
))
```

```python
def test_create_update_validate(client, auth, path):
    auth.login()
    response = client.post(path, data={'title': '', 'body': ''})
    assert b'Title is required.' in response.data
```

The `delete` view should redirect to the index URL and the post should no longer exist in the database.

tests/test_blog.py

```python
def test_delete(client, auth, app):
    auth.login()
    response = client.post('/1/delete')
    assert response.headers['Location'] == 'http://localhost/'

    with app.app_context():
        db = get_db()
        post = db.execute('SELECT * FROM post WHERE id = 1').fetchone()
        assert post is None
```

Running the Tests

Some extra configuration, which is not required but makes running tests with coverage less verbose, can be added to the project's `setup.cfg` file.

setup.cfg

```
[tool:pytest]
testpaths = tests

[coverage:run]
branch = True
source =
    flaskr
```

To run the tests, use the `pytest` command. It will find and run all the test functions you've written.

```
$ pytest

========================= test session starts ==========================
platform linux -- Python 3.6.4, pytest-3.5.0, py-1.5.3, pluggy-0.6.0
rootdir: /home/user/Projects/flask-tutorial, inifile: setup.cfg
```

```
collected 23 items

tests/test_auth.py ........                                      [ 34%]
tests/test_blog.py ............                                  [ 86%]
tests/test_db.py ..                                              [ 95%]
tests/test_factory.py ..                                         [100%]

====================== 24 passed in 0.64 seconds ======================
```

If any tests fail, pytest will show the error that was raised. You can run `pytest -v` to get a list of each test function rather than dots.

To measure the code coverage of your tests, use the `coverage` command to run pytest instead of running it directly.

```
$ coverage run -m pytest
```

You can either view a simple coverage report in the terminal:

```
$ coverage report

Name                 Stmts   Miss Branch BrPart  Cover
------------------------------------------------------
flaskr/__init__.py      21      0      2      0   100%
flaskr/auth.py          54      0     22      0   100%
flaskr/blog.py          54      0     16      0   100%
flaskr/db.py            24      0      4      0   100%
------------------------------------------------------
TOTAL                  153      0     44      0   100%
```

An HTML report allows you to see which lines were covered in each file:

```
$ coverage html
```

This generates files in the `htmlcov` directory. Open `htmlcov/index.html` in your browser to see the report.

Continue to Deploy to Production.