Testing Flask Applications

Something that is untested is broken.

The origin of this quote is unknown and while it is not entirely correct, it is also not far from the truth. Untested applications make it hard to improve existing code and developers of untested applications tend to become pretty paranoid. If an application has automated tests, you can safely make changes and instantly know if anything breaks.

Flask provides a way to test your application by exposing the Werkzeug test `Client` and handling the context locals for you. You can then use that with your favourite testing solution.

In this documentation we will use the pytest package as the base framework for our tests. You can install it with `pip`, like so:

```
$ pip install pytest
```

The Application

First, we need an application to test; we will use the application from the Tutorial. If you don't have that application yet, get the source code from the examples.

The Testing Skeleton

We begin by adding a tests directory under the application root. Then create a Python file to store our tests (`test_flaskr.py`). When we format the filename like `test_*.py`, it will be auto-discoverable by pytest.

Next, we create a pytest fixture called **`client()`** that configures the application for testing and initializes a new database:

```
import os
import tempfile

import pytest

from flaskr import flaskr


@pytest.fixture
def client():
    db_fd, flaskr.app.config['DATABASE'] = tempfile.mkstemp()
    flaskr.app.config['TESTING'] = True

    with flaskr.app.test_client() as client:
        with flaskr.app.app_context():
            flaskr.init_db()
        yield client

    os.close(db_fd)
    os.unlink(flaskr.app.config['DATABASE'])
```

This client fixture will be called by each individual test. It gives us a simple interface to the application, where we can trigger test requests to the application. The client will also keep track of cookies for us.

During setup, the `TESTING` config flag is activated. What this does is disable error catching during request handling, so that you get better error reports when performing test requests against the application.

Because SQLite3 is filesystem-based, we can easily use the **`tempfile`** module to create a temporary database and initialize it. The **`mkstemp()`** function does two things for us: it returns a low-level file handle and a random file name, the latter we use as database name. We just have to keep the *db_fd* around so that we can use the **`os.close()`** function to close the file.

To delete the database after the test, the fixture closes the file and removes it from the filesystem.

If we now run the test suite, we should see the following output:

```
$ pytest

================ test session starts ================
rootdir: ./flask/examples/flaskr, inifile: setup.cfg
collected 0 items

=========== no tests ran in 0.07 seconds ============
```

Even though it did not run any actual tests, we already know that our `flaskr` application is syntactically valid, otherwise the import would have died with an exception.

The First Test

Now it's time to start testing the functionality of the application. Let's check that the application shows "No entries here so far" if we access the root of the application (`/`). To do this, we add a new test function to `test_flaskr.py`, like this:

```
def test_empty_db(client):
    """Start with a blank database."""

    rv = client.get('/')
    assert b'No entries here so far' in rv.data
```

Notice that our test functions begin with the word *test*; this allows pytest to automatically identify the function as a test to run.

By using `client.get` we can send an HTTP `GET` request to the application with the given path. The return value will be a **`response_class`** object. We can now use the **`data`** attribute to inspect the return value (as string) from the application. In this case, we ensure that `'No entries here so far'` is part of the output.

Run it again and you should see one passing test:

```
$ pytest -v

================ test session starts ================
rootdir: ./flask/examples/flaskr, inifile: setup.cfg
collected 1 items

tests/test_flaskr.py::test_empty_db PASSED

============= 1 passed in 0.10 seconds ==============
```

Logging In and Out

The majority of the functionality of our application is only available for the administrative user, so we need a way to log our test client in and out of the application. To do this, we fire some requests to the login and logout pages with the required form data (username and password). And because the login and logout pages redirect, we tell the client to *follow_redirects*.

Add the following two functions to your `test_flaskr.py` file:

```
def login(client, username, password):
    return client.post('/login', data=dict(
        username=username,
        password=password
    ), follow_redirects=True)


def logout(client):
    return client.get('/logout', follow_redirects=True)
```

Now we can easily test that logging in and out works and that it fails with invalid credentials. Add this new test function:

```
def test_login_logout(client):
    """Make sure login and logout works."""

    rv = login(client, flaskr.app.config['USERNAME'], flaskr.app.config
    assert b'You were logged in' in rv.data

    rv = logout(client)
    assert b'You were logged out' in rv.data

    rv = login(client, flaskr.app.config['USERNAME'] + 'x', flaskr.app.
    assert b'Invalid username' in rv.data

    rv = login(client, flaskr.app.config['USERNAME'], flaskr.app.config
    assert b'Invalid password' in rv.data
```

Test Adding Messages

We should also test that adding messages works. Add a new test function like this:

```
def test_messages(client):
    """Test that messages work."""

    login(client, flaskr.app.config['USERNAME'], flaskr.app.config['PAS
    rv = client.post('/add', data=dict(
        title='<Hello>',
        text='<strong>HTML</strong> allowed here'
    ), follow_redirects=True)
    assert b'No entries here so far' not in rv.data
    assert b'&lt;Hello&gt;' in rv.data
    assert b'<strong>HTML</strong> allowed here' in rv.data
```

Here we check that HTML is allowed in the text but not in the title, which is the intended behavior.

Running that should now give us three passing tests:

```
$ pytest -v

================ test session starts ================
rootdir: ./flask/examples/flaskr, inifile: setup.cfg
collected 3 items

tests/test_flaskr.py::test_empty_db PASSED
tests/test_flaskr.py::test_login_logout PASSED
tests/test_flaskr.py::test_messages PASSED

============= 3 passed in 0.23 seconds ==============
```

Other Testing Tricks

Besides using the test client as shown above, there is also the **`test_request_context()`** method that can be used in combination with the `with` statement to activate a request context temporarily. With this you can access the **`request`**, **`g`** and **`session`** objects like in view functions. Here is a full example that demonstrates this approach:

```
import flask

app = flask.Flask(__name__)

with app.test_request_context('/?name=Peter'):
    assert flask.request.path == '/'
    assert flask.request.args['name'] == 'Peter'
```

All the other objects that are context bound can be used in the same way.

If you want to test your application with different configurations and there does not seem to be a good way to do that, consider switching to application factories (see Application Factories).

Note however that if you are using a test request context, the **`before_request()`** and **`after_request()`** functions are not called automatically. However **`teardown_request()`** functions are indeed executed when the test request context leaves the `with` block. If you do want the **`before_request()`** functions to be called as well, you need to call **`preprocess_request()`** yourself:

```
app = flask.Flask(__name__)

with app.test_request_context('/?name=Peter'):
    app.preprocess_request()
    ...
```

This can be necessary to open database connections or something similar depending on how your application was designed.

If you want to call the `after_request()` functions you need to call into `process_response()` which however requires that you pass it a response object:

```
app = flask.Flask(__name__)

with app.test_request_context('/?name=Peter'):
    resp = Response('...')
    resp = app.process_response(resp)
    ...
```

This in general is less useful because at that point you can directly start using the test client.

Faking Resources and Context

▶ *Changelog*

A very common pattern is to store user authorization information and database connections on the application context or the `flask.g` object. The general pattern for this is to put the object on there on first usage and then to remove it on a teardown. Imagine for instance this code to get the current user:

```
def get_user():
    user = getattr(g, 'user', None)
    if user is None:
        user = fetch_current_user_from_database()
        g.user = user
    return user
```

For a test it would be nice to override this user from the outside without having to change some code. This can be accomplished with hooking the `flask.appcontext_pushed` signal:

```
from contextlib import contextmanager
from flask import appcontext_pushed, g

@contextmanager
def user_set(app, user):
    def handler(sender, **kwargs):
        g.user = user
    with appcontext_pushed.connected_to(handler, app):
        yield
```

And then to use it:

```
from flask import json, jsonify

@app.route('/users/me')
def users_me():
    return jsonify(username=g.user.username)

with user_set(app, my_user):
    with app.test_client() as c:
        resp = c.get('/users/me')
        data = json.loads(resp.data)
        self.assert_equal(data['username'], my_user.username)
```

Keeping the Context Around

▶ *Changelog*

Sometimes it is helpful to trigger a regular request but still keep the context around for a little longer so that additional introspection can happen. With Flask 0.4 this is possible by using the **`test_client()`** with a `with` block:

```
app = flask.Flask(__name__)

with app.test_client() as c:
    rv = c.get('/?tequila=42')
    assert request.args['tequila'] == '42'
```

If you were to use just the **`test_client()`** without the `with` block, the `assert` would fail with an error because *request* is no longer available (because you are trying to use it out-side of the actual request).

Accessing and Modifying Sessions

▶ *Changelog*

Sometimes it can be very helpful to access or modify the sessions from the test client. Gen-erally there are two ways for this. If you just want to ensure that a session has certain keys set to certain values you can just keep the context around and access **`flask.session`**:

```
with app.test_client() as c:
    rv = c.get('/')
    assert flask.session['foo'] == 42
```

This however does not make it possible to also modify the session or to access the session before a request was fired. Starting with Flask 0.8 we provide a so called "session transaction" which simulates the appropriate calls to open a session in the context of the test client and to modify it. At the end of the transaction the session is stored and ready to be used by the test client. This works independently of the session backend used:

```
with app.test_client() as c:
    with c.session_transaction() as sess:
        sess['a_key'] = 'a value'

    # once this is reached the session was stored and ready to be used
    c.get(...)
```

Note that in this case you have to use the `sess` object instead of the **`flask.session`** proxy. The object however itself will provide the same interface.

Testing JSON APIs

▶ *Changelog*

Flask has great support for JSON, and is a popular choice for building JSON APIs. Making requests with JSON data and examining JSON data in responses is very convenient:

```
from flask import request, jsonify

@app.route('/api/auth')
def auth():
    json_data = request.get_json()
    email = json_data['email']
    password = json_data['password']
    return jsonify(token=generate_token(email, password))

with app.test_client() as c:
    rv = c.post('/api/auth', json={
        'email': 'flask@example.com', 'password': 'secret'
    })
    json_data = rv.get_json()
    assert verify_token(email, json_data['token'])
```

Passing the `json` argument in the test client methods sets the request data to the JSON-serialized object and sets the content type to `application/json`. You can get the JSON data from the request or response with `get_json`.

Testing CLI Commands

Click comes with utilities for testing your CLI commands. A **`CliRunner`** runs commands in isolation and captures the output in a **`Result`** object.

Flask provides **`test_cli_runner()`** to create a **`FlaskCliRunner`** that passes the Flask app to the CLI automatically. Use its **`invoke()`** method to call commands in the same way they would be called from the command line.

```
import click

@app.cli.command('hello')
@click.option('--name', default='World')
def hello_command(name)
    click.echo(f'Hello, {name}!')

def test_hello():
    runner = app.test_cli_runner()

    # invoke the command directly
    result = runner.invoke(hello_command, ['--name', 'Flask'])
    assert 'Hello, Flask' in result.output

    # or by name
    result = runner.invoke(args=['hello'])
    assert 'World' in result.output
```

In the example above, invoking the command by name is useful because it verifies that the command was correctly registered with the app.

If you want to test how your command parses parameters, without running the command, use its **`make_context()`** method. This is useful for testing complex validation rules and custom types.

```
def upper(ctx, param, value):
    if value is not None:
        return value.upper()

@app.cli.command('hello')
@click.option('--name', default='World', callback=upper)
def hello_command(name)
```

```python
    click.echo(f'Hello, {name}!')

def test_hello_params():
    context = hello_command.make_context('hello', ['--name', 'flask'])
    assert context.params['name'] == 'FLASK'
```