**Flask-Testing**

The **Flask-Testing** extension provides unit testing utilities for Flask.

## Installing Flask-Testing

Install with **pip** and **easy\_install**:

pip install Flask-Testing

or download the latest version from version control:

git clone https**:**//github.com/jarus/flask-testing.git

cd flask-testing

python setup.py develop

If you are using **virtualenv**, it is assumed that you are installing **Flask-Testing** in the same virtualenv as your Flask application(s).

## Writing tests

Simply subclass the TestCase class:

**from** flask\_testing **import** TestCase

**class** MyTest**(**TestCase**):**

**pass**

You must specify the create\_app method, which should return a Flask instance:

**from** flask **import** Flask

**from** flask\_testing **import** TestCase

**class** MyTest**(**TestCase**):**

**def** create\_app**(**self**):**

app = Flask**(**\_\_name\_\_**)**

app.config**[**'TESTING'**]** = **True**

**return** app

If you don’t define create\_app a NotImplementedError will be raised.

### Testing with LiveServer

If you want your tests done via Selenium or other headless browser like PhantomJS you can use the LiveServerTestCase:

**import** urllib2

**from** flask **import** Flask

**from** flask\_testing **import** LiveServerTestCase

**class** MyTest**(**LiveServerTestCase**):**

**def** create\_app**(**self**):**

app = Flask**(**\_\_name\_\_**)**

app.config**[**'TESTING'**]** = **True**

*# Default port is 5000*

app.config**[**'LIVESERVER\_PORT'**]** = 8943

*# Default timeout is 5 seconds*

app.config**[**'LIVESERVER\_TIMEOUT'**]** = 10

**return** app

**def** test\_server\_is\_up\_and\_running**(**self**):**

response = urllib2.urlopen**(**self.get\_server\_url**())**

self.assertEqual**(**response.code**,** 200**)**

The method get\_server\_url will return [http://localhost:8943](http://localhost:8943/) in this case.

### Dynamic LiveServerTestCase port

By default, LiveServerTestCase will use the pre-defined port for running the live server. If multiple tests need to run in parallel, the LIVESERVER\_PORT can be set to 0 to have the underlying operating system pick an open port for the server. The full address of the running server can be accessed via the get\_server\_url call on the test case:

**import** urllib2

**from** flask **import** Flask

**from** flask\_testing **import** LiveServerTestCase

**class** MyTest**(**LiveServerTestCase**):**

**def** create\_app**(**self**):**

app = Flask**(**\_\_name\_\_**)**

app.config**[**'TESTING'**]** = **True**

*# Set to 0 to have the OS pick the port.*

app.config**[**'LIVESERVER\_PORT'**]** = 0

**return** app

**def** test\_server\_is\_up\_and\_running**(**self**):**

response = urllib2.urlopen**(**self.get\_server\_url**())**

self.assertEqual**(**response.code**,** 200**)**

### Testing JSON responses

If you are testing a view that returns a JSON response, you can test the output using a special json attribute appended to the Response object:

@app.route**(**"/ajax/"**)**

**def** some\_json**():**

**return** jsonify**(**success=**True)**

**class** TestViews**(**TestCase**):**

**def** test\_some\_json**(**self**):**

response = self.client.get**(**"/ajax/"**)**

self.assertEquals**(**response.json**,** dict**(**success=**True))**

### Opt to not render the templates

When testing with mocks the template rendering can be a problem. If you don’t want to render the templates in the tests you can use the render\_templates attribute:

**class** TestNotRenderTemplates**(**TestCase**):**

render\_templates = **False**

**def** test\_assert\_not\_process\_the\_template**(**self**):**

response = self.client.get**(**"/template/"**)**

**assert** "" == response.data

The signal will be sent anyway so that you can check if the template was rendered using the assert\_template\_used method:

**class** TestNotRenderTemplates**(**TestCase**):**

render\_templates = **False**

**def** test\_assert\_mytemplate\_used**(**self**):**

response = self.client.get**(**"/template/"**)**

self.assert\_template\_used**(**'mytemplate.html'**)**

When the template rendering is turned off the tests will also run faster and the view logic can be tested in isolation.

### Using with Twill

[Twill](http://twill.idyll.org/) is a simple language for browsing the Web through a command line interface.

Note

Please note that Twill only supports Python 2.x and therefore cannot be used with Python 3 or above.

Flask-Testing comes with a helper class for creating functional tests using Twill:

**def** test\_something\_with\_twill**(**self**):**

**with** Twill**(**self.app**,** port=3000**)** **as** t**:**

t.browser.go**(**t.url**(**"/"**))**

The older TwillTestCase has been deprecated.

### Testing with SQLAlchemy

This covers a couple of points if you are using **Flask-Testing** with [SQLAlchemy](http://sqlalchemy.org/). It is assumed that you are using the [Flask-SQLAlchemy](http://packages.python.org/Flask-SQLAlchemy/) extension, but if not the examples should not be too difficult to adapt to your own particular setup.

First, ensure you set the database URI to something other than your production database ! Second, it’s usually a good idea to create and drop your tables with each test run, to ensure clean tests:

**from** flask\_testing **import** TestCase

**from** myapp **import** create\_app**,** db

**class** MyTest**(**TestCase**):**

SQLALCHEMY\_DATABASE\_URI = "sqlite://"

TESTING = **True**

**def** create\_app**(**self**):**

*# pass in test configuration*

**return** create\_app**(**self**)**

**def** setUp**(**self**):**

db.create\_all**()**

**def** tearDown**(**self**):**

db.session.remove**()**

db.drop\_all**()**

Notice also that db.session.remove() is called at the end of each test, to ensure the SQLAlchemy session is properly removed and that a new session is started with each test run - this is a common “gotcha”.

Another gotcha is that Flask-SQLAlchemy **also** removes the session instance at the end of every request (as should any thread safe application using SQLAlchemy with **scoped\_session**). Therefore the session is cleared along with any objects added to it every time you call client.get() or another client method.

For example:

**class** SomeTest**(**MyTest**):**

**def** test\_something**(**self**):**

user = User**()**

db.session.add**(**user**)**

db.session.commit**()**

*# this works*

**assert** user **in** db.session

response = self.client.get**(**"/"**)**

*# this raises an AssertionError*

**assert** user **in** db.session

You now have to re-add the “user” instance back to the session with db.session.add(user), if you are going to make any further database operations on it.

Also notice that for this example the SQLite in-memory database is used : while it is faster for tests, if you have database-specific code (e.g. for MySQL or PostgreSQL) it may not be applicable.

You may also want to add a set of instances for your database inside of a setUp() once your database tables have been created. If you want to work with larger sets of data, look at [Fixture](http://farmdev.com/projects/fixture/index.html) which includes support for SQLAlchemy.