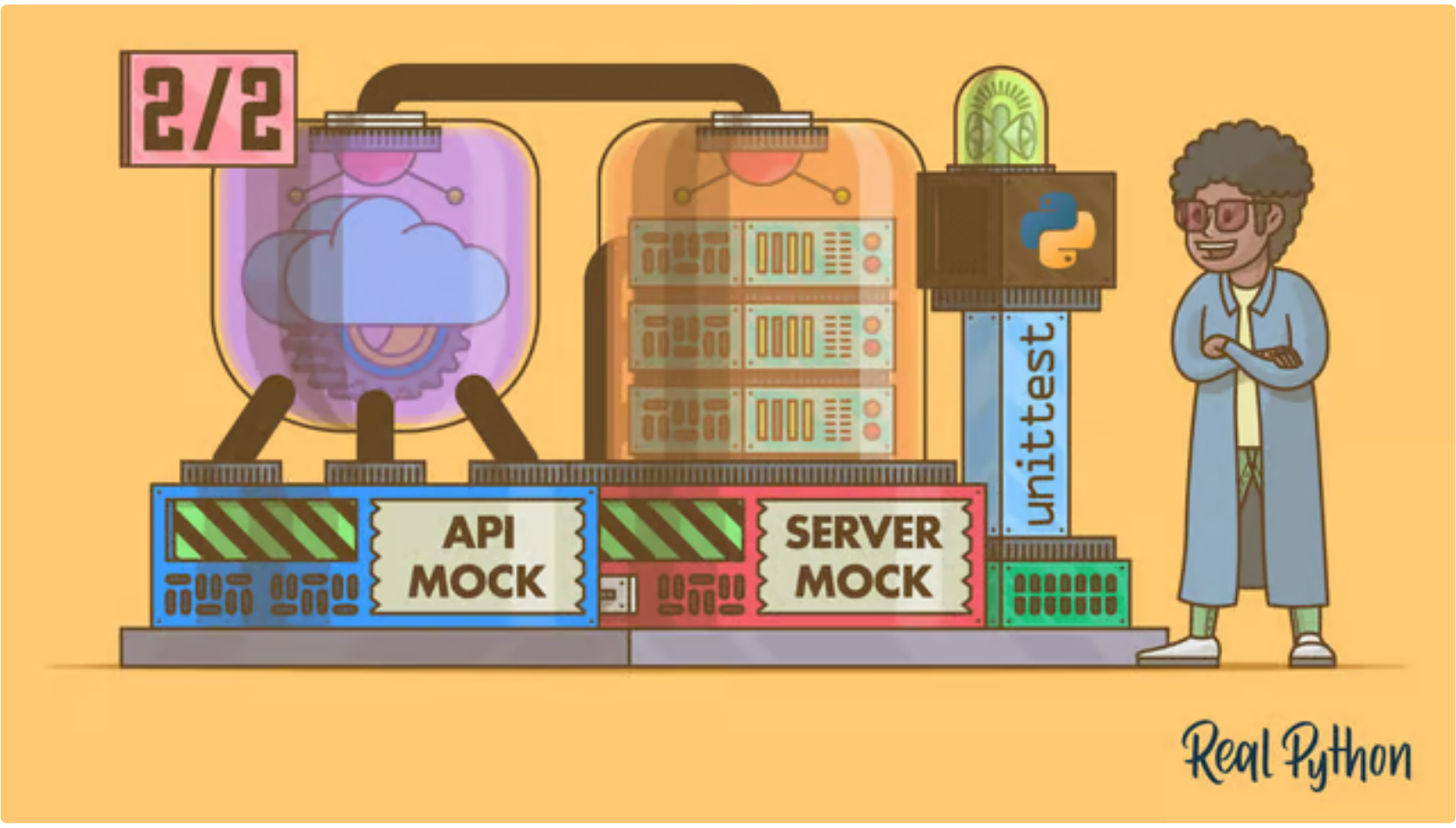


Testing External APIs With Mock Servers

by Real Python Jul 06, 2016 2 Comments

advanced api testing web-dev

Mark as Completed

Tweet Share Email

Table of Contents

Despite being so useful, external APIs can be a pain to test. When you hit an actual API, your tests are at the mercy of the external server, which can result in the following pain points:

- The request-response cycle can take several seconds. That might not seem like much at first, but the time compounds with each test. Imagine calling an API 10, 50, or even 100 times when testing your entire application.
- The API may have rate limits set up.
- The API server may be unreachable. Maybe the server is down for maintenance? Maybe it failed with an error and the development team is working to get it functional again> Do you really want the success of your tests to rely on the health of a server that you don't control?

Your tests shouldn't assess whether an API server is running; they should test whether your code is operating as expected.

In the previous tutorial, we introduced the concept of mock objects, demonstrated how you could use them to tes code that interacts with external APIs. **This tutorial builds on the same topics, but here we walk you through how to actually build a mock server rather than mocking the APIs.** With a mock server in place, you can perform end-

to-end tests. You can use your application and get actual feedback from the mock server in real time.

When you finish working through the following examples, you will have programmed a basic mock server and two tests - one that uses the real API server and one that uses the mock server. Both tests will access the same service, an API that retrieves a list of users.

> NOTE: This tutorial uses Python v3.5.1.

Getting Started

Start by following the First steps section of the previous post. Or grab the code from the repository. Make sure the test passes before moving on:

Shell

```
$ nosetests --verbosity=2 project
test_todos.test_request_response ... ok

----------------------------------------------------------------------
Ran 1 test in 1.029s

OK
```


Testing the Mock API

With the set up complete, you can program your mock server. Write a test that describes the behavior:

project/tests/test_mock_server.py

Python

```
# Third-party imports...
from nose.tools import assert_true
import requests


def test_request_response():
    url = 'http://localhost:{port}/users'.format(port=mock_server_port)

    # Send a request to the mock API server and store the response.
    response = requests.get(url)

    # Confirm that the request-response cycle completed successfully.
    assert_true(response.ok)
```

Notice that it starts off looking almost identical to the real API test. The URL has changed and is now pointing to an API endpoint on *localhost* where the mock server will run.

Here is how to create a mock server in Python:

project/tests/test_mock_server.py

Python

```python
# Standard library imports...
from http.server import BaseHTTPRequestHandler, HTTPServer
import socket
from threading import Thread

# Third-party imports...
from nose.tools import assert_true
import requests


class MockServerRequestHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        # Process an HTTP GET request and return a response with an HTTP 200 status.
        self.send_response(requests.codes.ok)
        self.end_headers()
        return


def get_free_port():
    s = socket.socket(socket.AF_INET, type=socket.SOCK_STREAM)
    s.bind(('localhost', 0))
    address, port = s.getsockname()
    s.close()
    return port


class TestMockServer(object):
    @classmethod
    def setup_class(cls):
        # Configure mock server.
        cls.mock_server_port = get_free_port()
        cls.mock_server = HTTPServer(('localhost', cls.mock_server_port), MockServerRequestHandler)

        # Start running mock server in a separate thread.
        # Daemon threads automatically shut down when the main process exits.
        cls.mock_server_thread = Thread(target=cls.mock_server.serve_forever)
        cls.mock_server_thread.setDaemon(True)
        cls.mock_server_thread.start()

    def test_request_response(self):
        url = 'http://localhost:{port}/users'.format(port=self.mock_server_port)

        # Send a request to the mock API server and store the response.
        response = requests.get(url)

        # Confirm that the request-response cycle completed successfully.
        print(response)
        assert_true(response.ok)
```

First, create a subclass of `BaseHTTPRequestHandler`. This class captures the request and constructs the response to return. Override the `do_GET()` function to craft the response for an HTTP GET request. In this case, just return an OK status. Next, write a function to get an available port number for the mock server to use.

The next block of code actually configures the server. Notice how the code instantiates an `HTTPServer` instance and passes it a port number and a handler. Next create a thread so that the server can be run asynchronously and your main program thread can communicate with it. Make the thread a daemon, which tells the thread to stop when the main program exits. Finally, start the thread to serve the mock server forever (until the tests finish).

Create a test class and move the test function to it. You must add an additional method to ensure that the mock server is launched before any of the tests run. Notice that this new code lives within a special class-level function, `setup_class()`.

Run the tests and watch them pass:

Shell

```
$ nosetests --verbosity=2 project
```

Testing a Service that Hits the API

You probably want to call more than one API endpoint in your code. As you design your app, you will likely create service functions to send requests to an API and then process the responses in some way. Maybe you will store the response data in a database. Or you will pass the data to a user interface.

Refactor your code to pull the hardcoded API base URL into a constant. Add this variable to a *constants.py* file:

project/constants.py

Python

```
BASE_URL = 'http://jsonplaceholder.typicode.com'
```

Next, encapsulate the logic to retrieve users from the API into a function. Notice how new URLs can be created by joining a URL path to the base.

project/services.py

Python

```
# Standard library imports...
from urllib.parse import urljoin

# Third-party imports...
import requests

# Local imports...
from project.constants import BASE_URL

USERS_URL = urljoin(BASE_URL, 'users')


def get_users():
    response = requests.get(USERS_URL)
    if response.ok:
        return response
    else:
        return None
```

Move the mock server code from the feature file to a new Python file, so that it can easily be reused. Add conditional logic to the request handler to check which API endpoint the HTTP request is targeting. Beef up the response by adding some simple header information and a basic response payload. The server creation and kick off code can be encapsulated in a convenience method, `start_mock_server()`.

project/tests/mocks.py

Python

```python
# Standard library imports...
from http.server import BaseHTTPRequestHandler, HTTPServer
import json
import re
import socket
from threading import Thread

# Third-party imports...
import requests


class MockServerRequestHandler(BaseHTTPRequestHandler):
    USERS_PATTERN = re.compile(r'/users')

    def do_GET(self):
        if re.search(self.USERS_PATTERN, self.path):
            # Add response status code.
            self.send_response(requests.codes.ok)

            # Add response headers.
            self.send_header('Content-Type', 'application/json; charset=utf-8')
            self.end_headers()

            # Add response content.
            response_content = json.dumps([])
            self.wfile.write(response_content.encode('utf-8'))
            return


def get_free_port():
    s = socket.socket(socket.AF_INET, type=socket.SOCK_STREAM)
    s.bind(('localhost', 0))
    address, port = s.getsockname()
    s.close()
    return port


def start_mock_server(port):
    mock_server = HTTPServer(('localhost', port), MockServerRequestHandler)
    mock_server_thread = Thread(target=mock_server.serve_forever)
    mock_server_thread.setDaemon(True)
    mock_server_thread.start()
```

With your changes to the logic completed, alter the tests to use the new service function. Update the tests to check the increased information that is being passed back from the server.

project/tests/test_real_server.py

Python

```python
# Third-party imports...
from nose.tools import assert_dict_contains_subset, assert_is_instance, assert_true

# Local imports...
from project.services import get_users


def test_request_response():
    response = get_users()

    assert_dict_contains_subset({'Content-Type': 'application/json; charset=utf-8'}, response.header
    assert_true(response.ok)
    assert_is_instance(response.json(), list)
```

project/tests/test_mock_server.py

Python

```python
# Third-party imports...
from unittest.mock import patch
from nose.tools import assert_dict_contains_subset, assert_list_equal, assert_true

# Local imports...
from project.services import get_users
from project.tests.mocks import get_free_port, start_mock_server


class TestMockServer(object):
    @classmethod
    def setup_class(cls):
        cls.mock_server_port = get_free_port()
        start_mock_server(cls.mock_server_port)

    def test_request_response(self):
        mock_users_url = 'http://localhost:{port}/users'.format(port=self.mock_server_port)

        # Patch USERS_URL so that the service uses the mock server URL instead of the real URL.
        with patch.dict('project.services.__dict__', {'USERS_URL': mock_users_url}):
            response = get_users()

        assert_dict_contains_subset({'Content-Type': 'application/json; charset=utf-8'}, response.he
        assert_true(response.ok)
        assert_list_equal(response.json(), [])
```

Notice a new technique being used in the *test_mock_server.py* code. The `response = get_users()` line is wrapped with a `patch.dict()` function from the *mock* library.

What does this statement do?

Remember, you moved the `requests.get()` function from the feature logic to the `get_users()` service function. Internally, `get_users()` calls `requests.get()` using the `USERS_URL` variable. The `patch.dict()` function temporarily replaces the value of the `USERS_URL` variable. In fact, it does so only within the scope of the `with` statement. After that code runs, the `USERS_URL` variable is restored to its original value. This code *patches* the URL to use the mock server address.

Run the tests and watch them pass.

Shell

```
$ nosetests --verbosity=2
test_mock_server.TestMockServer.test_request_response ... 127.0.0.1 - - [05/Jul/2016 20:45:30] "GET
ok
test_real_server.test_request_response ... ok
test_todos.test_request_response ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.871s

OK
```


Skipping Tests that Hit the Real API

We began this tutorial describing the merits of testing a mock server instead of a real one, however, your code currently tests both. How do you configure the tests to ignore the real server? The Python ‘unittest’ library provides several functions that allow you to skip tests. You can use the conditional skip function ‘skipIf’ along with an

environment variable to toggle the real server tests on and off. In the following example, we pass a tag name that should be ignored:

Shell

```
$ export SKIP_TAGS=real
```

project/constants.py

Python

```
# Standard-library imports...
import os


BASE_URL = 'http://jsonplaceholder.typicode.com'
SKIP_TAGS = os.getenv('SKIP_TAGS', '').split()
```

project/tests/test_real_server.py

Python

```
# Standard library imports...
from unittest import skipIf

# Third-party imports...
from nose.tools import assert_dict_contains_subset, assert_is_instance, assert_true

# Local imports...
from project.constants import SKIP_TAGS
from project.services import get_users


@skipIf('real' in SKIP_TAGS, 'Skipping tests that hit the real API server.')
def test_request_response():
    response = get_users()

    assert_dict_contains_subset({'Content-Type': 'application/json; charset=utf-8'}, response.header
    assert_true(response.ok)
    assert_is_instance(response.json(), list)
```

Run the tests and pay attention to how the real server test is ignored:

Shell

```
$ nosetests --verbosity=2 project
test_mock_server.TestMockServer.test_request_response ... 127.0.0.1 - - [05/Jul/2016 20:52:18] "GET
ok
test_real_server.test_request_response ... SKIP: Skipping tests that hit the real API server.
test_todos.test_request_response ... ok

----------------------------------------------------------------------
Ran 3 tests in 1.196s

OK (SKIP=1)
```

Next Steps

Now that you have created a mock server to test your external API calls, you can apply this knowledge to your own projects. Build upon the simple tests created here. Expand the functionality of the handler to mimic the behavior of the real API more closely.

Try the following exercises to level up:

- Return a response with a status of HTTP 404 (not found) if a request is sent with an unknown path.
- Return a response with a status of HTTP 405 (method not allowed) if a request is sent with a method that is not allowed (POST, DELETE, UPDATE).

- Return actual user data for a valid request to `/users`.
- Write tests to capture those scenarios.

Grab the code from the repo.

Mark as Completed

🐍 Python Tricks 💌

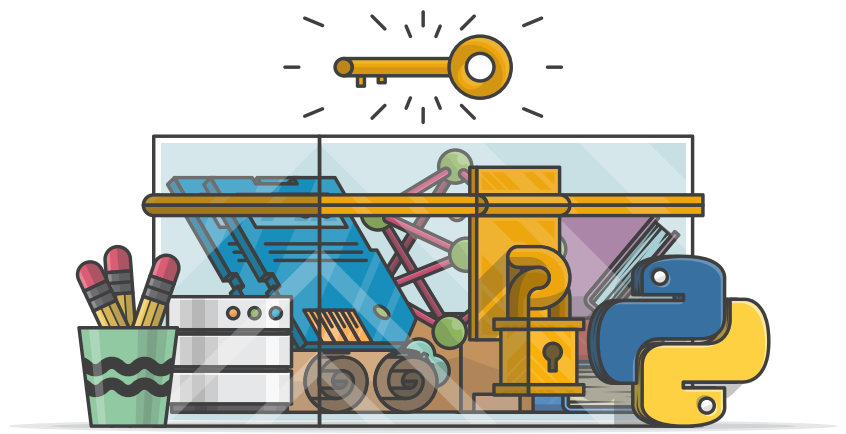
Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Email Address

Send Me Python Tricks »

Master Real-World Python Skills With Unlimited Access to Real Python



Join us and get access to thousands of tutorials, hands-on video courses, and a community of expert Pythonistas:

Level Up Your Python Skills »

What Do You Think?

Rate this article:

Tweet Share Share Email

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. Get tips for asking good questions and get answers to common questions in our support portal.

Looking for a real-time conversation? Visit the Real Python Community Chat or join the next "Office Hours" Live Q&A Session. Happy Pythoning!

Keep Learning

Related Tutorial Categories: advanced api testing web-dev