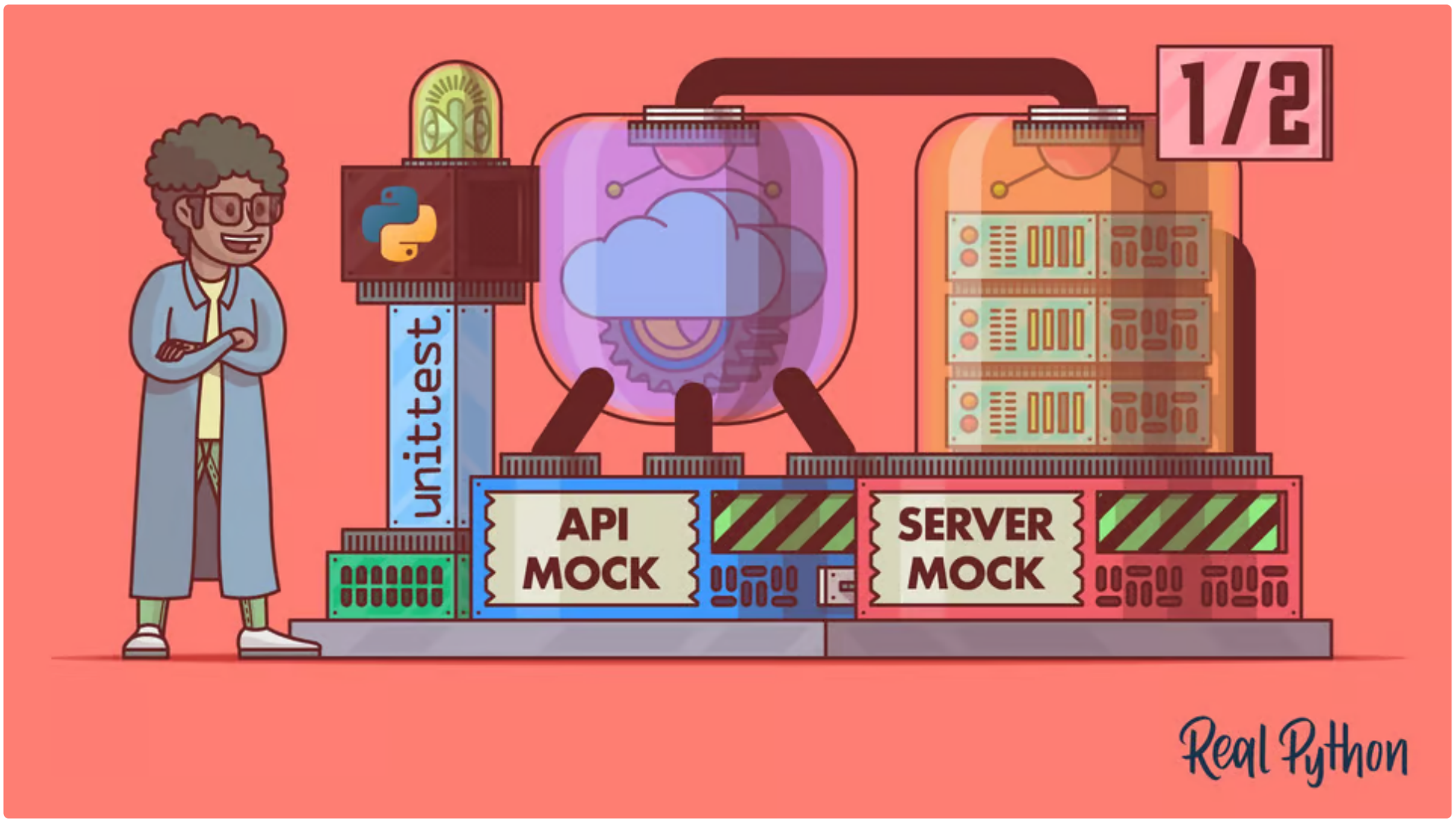

Mocking External APIs in Python

by Real Python Jun 20, 2016 18 Comments advanced api testing web-dev

Mark as Completed

Share Share Email

Table of Contents

The following tutorial demonstrates how to test the use of an external API using Python mock objects.

Integrating with a third-party application is a great way to extend the functionality of your product.

However, the added value also comes with obstacles. You do not own the external library, which means that you cann the servers that host it, the code that comprises its logic, or the data that gets transferred between it and your app. C those issues, users are constantly manipulating the data through their interactions with the library.

Help

If you want to enhance the utility of your application with a third-party API, then you need to be confident that the two systems will play nice. You need to test that the two applications interface in predictable ways, and you need your tests to execute in a controlled environment.

At first glance, it might seem like you do not have any control over a third-party application. Many of them do not offer testing servers. You cannot test live data, and even if you could, the tests would return unreliable results as the data was updated through use. Also, you never want your automated tests to connect to an external server. An error on their side could bring a halt to your development if releasing your code depends on whether your tests pass. Luckily, there is a way to test the implementation of a third-party API in a controlled environment without needing to actually connect to an outside data source. The solution is to fake the functionality of the external code using something known as mocks.

A mock is a fake object that you construct to look and act like real data. You swap it with the actual object and trick the system into thinking that the mock is the real deal. Using a mock reminds me of a classic movie trope where the hero grabs a henchman, puts on his uniform, and steps into a line of marching enemies. Nobody notices the impostor and everybody keeps moving—business as usual.

Third-party authentication, such as OAuth, is a good candidate for mocking within your application. OAuth requires your application to communicate with an external server, it involves real user data, and your application relies on its success in order to gain access to its APIs. Mocking authentication allows you to test your system as an authorized user without having to go through the actual process of exchanging credentials. In this case you do not want to test whether your system successfully authenticates a user; you want to test how your application's functions behave *after* you have been authenticated.

> NOTE: This tutorial uses Python v3.5.1.

First steps

Begin by setting up a new development environment to hold your project code. Create a new virtual environment and then install the following libraries:

Shell

```
$ pip install nose requests
```

Here is a quick rundown of each library you are installing, in case you have never encountered them:

- The mock library is used for testing Python code by replacing parts of your system with mock objects. *NOTE: The `mock` library is part of `unittest` if you are using Python 3.3 or greater. If you are using an older version, please install the backport mock library.*
- The nose library extends the built-in Python `unittest` module to make testing easier. You can use `unittest` or other third-party libraries such as *pytest* to achieve the same results, but I prefer *nose*'s assertion methods.
- The requests library greatly simplifies HTTP calls in Python.

For this tutorial, you will be communicating with a fake online API that was built for testing - JSON Placeholder. Before you write any tests, you need to know what to expect from the API.

First, you should expect that the API you are targeting actually returns a response when you send it a request. Confirm this assumption by calling the endpoint with *cURL*:

Shell

```
$ curl -X GET 'http://jsonplaceholder.typicode.com/todos'
```

This call should return a JSON-serialized list of todo items. Pay attention to the structure of the todo data in the response. You should see a list of objects with the keys `userId`, `id`, `title`, and `completed`. You are now prepared to make your second assumption–you know what to expect the data to look like. The API endpoint is alive and functioning. You proved that by calling

it from the command line. Now, write a *nose* test so that you can confirm the life of the server in the future. Keep it simple. You should only be concerned with whether the server returns an OK response.

project/tests/test_todos.py

Python

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


def test_request_response():
    # Send a request to the API server and store the response.
    response = requests.get('http://jsonplaceholder.typicode.com/todos')

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

Run the test and watch it pass:

Shell

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

----------------------------------------------------------------------
Ran 1 test in 9.270s

OK
```


Refactoring your code into a service

Chances are good that you will call an external API many times throughout your application. Also, those API calls will likely involve more logic than simply making an HTTP request, such as data processing, error handling, and filtering. You should pull the code out of your test and refactor it into a service function that encapsulates all of that expected logic.

Rewrite your test to reference the service function and to test the new logic.

project/tests/test_todos.py

Python

```
# Third-party imports...
from nose.tools import assert_is_not_none

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


def test_request_response():
    # Call the service, which will send a request to the server.
    response = get_todos()

    # If the request is sent successfully, then I expect a response to be returned.
    assert_is_not_none(response)
```

Run the test and watch it fail, and then write the minimum amount of code to make it pass:

project/services.py

Python

```
# Standard library imports...
try:
    from urllib.parse import urljoin
except ImportError:
    from urlparse import urljoin

# Third-party imports...
import requests

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

TODOS_URL = urljoin(BASE_URL, 'todos')


def get_todos():
    response = requests.get(TODOS_URL)
    if response.ok:
        return response
    else:
        return None
```

project/constants.py

Python

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

The first test that you wrote expected a response to be returned with an OK status. You refactored your programming logic into a service function that returns the response itself when the request to the server is successful. A `None` value is returned if the request fails. The test now includes an assertion to confirm that the function does not return `None`.

Notice how I instructed you to create a `constants.py` file and then I populated it with a `BASE_URL`. The service function extends the `BASE_URL` to create the `TODOS_URL`, and since all of the API endpoints use the same base, you can continue to create new ones without having to rewrite that bit of code. Putting the `BASE_URL` in a separate file allows you to edit it in one place, which will come in handy if multiple modules reference that code.

Run the test and watch it pass.

Shell

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

----------------------------------------------------------------------
Ran 1 test in 1.475s

OK
```

Your first mock

The code is working as expected. You know this because you have a passing test. Unfortunately, you have a problem–your service function is still accessing the external server directly. When you call `get_todos()`, your code is making a request to the API endpoint and returning a result that depends on that server being live. Here, I will demonstrate how to detach your programming logic from the actual external library by swapping the real request with a fake one that returns the same data.

project/tests/test_todos.py

Python

```
# Standard library imports...
from unittest.mock import Mock, patch

# Third-party imports...
from nose.tools import assert_is_not_none

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


@patch('project.services.requests.get')
def test_getting_todos(mock_get):
    # Configure the mock to return a response with an OK status code.
    mock_get.return_value.ok = True

    # Call the service, which will send a request to the server.
    response = get_todos()

    # If the request is sent successfully, then I expect a response to be returned.
    assert_is_not_none(response)
```

Notice that I did not change the service function at all. The only part of the code that I edited was the test itself. First, I imported the `patch()` function from the `mock` library. Next, I modified the test function with the `patch()` function as a decorator, passing in a reference to `project.services.requests.get`. In the function itself, I passed in a parameter `mock_get`, and then in the body of the test function, I added a line to set `mock_get.return_value.ok = True`.

Great. So what actually happens now when the test is run? Before I dive into that, you need to understand something about the way the `requests` library works. When you call the `requests.get()` function, it makes an HTTP request behind the scenes and then returns an HTTP response in the form of a `Response` object. The `get()` function itself communicates with the external server, which is why you need to target it. Remember the image of the hero swapping places with the enemy while wearing his uniform? You need to dress the mock to look and act like the `requests.get()` function.

When the test function is run, it finds the module where the `requests` library is declared, `project.services`, and it replaces the targeted function, `requests.get()`, with a mock. The test also tells the mock to behave the way the service function expects it to act. If you look at `get_todos()`, you see that the success of the function depends on `if response.ok:` returning `True`. That is what the line `mock_get.return_value.ok = True` is doing. When the `ok` property is called on the mock, it will return `True` just like the actual object. The `get_todos()` function will return the response, which is the mock, and the test will pass because the mock is not `None`.

Run the test to see it pass.

Shell

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


Other ways to patch

Using a decorator is just one of several ways to patch a function with a mock. In the next example, I explicitly patch a function within a block of code, using a context manager. The `with` statement patches a function used by any code in the code block. When the code block ends, the original function is restored. The `with` statement and the decorator accomplish the same goal: Both methods patch `project.services.request.get`.

project/tests/test_todos.py

Python

```python
# Standard library imports...
from unittest.mock import patch

# Third-party imports...
from nose.tools import assert_is_not_none

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


def test_getting_todos():
    with patch('project.services.requests.get') as mock_get:
        # Configure the mock to return a response with an OK status code.
        mock_get.return_value.ok = True

        # Call the service, which will send a request to the server.
        response = get_todos()

    # If the request is sent successfully, then I expect a response to be returned.
    assert_is_not_none(response)
```

Run the tests to see that they still pass.

Another way to patch a function is to use a patcher. Here, I identify the source to patch, and then I explicitly start using the mock. The patching does not stop until I explicitly tell the system to stop using the mock.

project/tests/test_todos.py

Python

```python
# Standard library imports...
from unittest.mock import patch

# Third-party imports...
from nose.tools import assert_is_not_none

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


def test_getting_todos():
    mock_get_patcher = patch('project.services.requests.get')

    # Start patching `requests.get`.
    mock_get = mock_get_patcher.start()

    # Configure the mock to return a response with an OK status code.
    mock_get.return_value.ok = True

    # Call the service, which will send a request to the server.
    response = get_todos()

    # Stop patching `requests.get`.
    mock_get_patcher.stop()

    # If the request is sent successfully, then I expect a response to be returned.
    assert_is_not_none(response)
```

Run the tests again to get the same successful result.

Now that you have seen three ways to patch a function with a mock, when should you use each one? The short answer: it is entirely up to you. Each patching method is completely valid. That being said, I have found that specific coding patterns work especially well with the following patching methods.

1. **Use a decorator** when all of the code in your test function body uses a mock.

2. **Use a context manager** when some of the code in your test function uses a mock and other code references the actual function.
3. **Use a patcher** when you need to explicitly start and stop mocking a function across multiple tests (e.g. the `setUp()` and `tearDown()` functions in a test class).

I use each of these methods in this tutorial, and I will highlight each one as I introduce it for the first time.

Mocking the complete service behavior

In the previous examples, you implemented a basic mock and tested a simple assertion–whether the `get_todos()` function returned `None`. The `get_todos()` function calls the external API and receives a response. If the call is successful, the function returns a response object, which contains a JSON-serialized list of todos. If the request fails, `get_todos()` returns `None`. In the following example, I demonstrate how to mock the entire functionality of `get_todos()`. At the beginning of this tutorial, the initial call you made to the server using *cURL* returned a JSON-serialized list of dictionaries, which represented todo items. This example will show you how to mock that data.

Remember how `@patch()` works: You provide it a path to the function you want to mock. The function is found, `patch()` creates a `Mock` object, and the real function is temporarily replaced with the mock. When `get_todos()` is called by the test, the function uses the `mock_get` the same way it would use the real `get()` method. That means that it calls `mock_get` like a function and expects it to return a response object.

In this case, the response object is a `requests` library `Response` object, which has several attributes and methods. You faked one of those properties, `ok`, in a previous example. The `Response` object also has a `json()` function which converts its JSON-serialized string content into a Python datatype (e.g. a `list` or a `dict`).

project/tests/test_todos.py

Python

```
# Standard library imports...
from unittest.mock import Mock, patch

# Third-party imports...
from nose.tools import assert_is_none, assert_list_equal

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


@patch('project.services.requests.get')
def test_getting_todos_when_response_is_ok(mock_get):
    todos = [{
        'userId': 1,
        'id': 1,
        'title': 'Make the bed',
        'completed': False
    }]

    # Configure the mock to return a response with an OK status code. Also, the mock should have
    # a `json()` method that returns a list of todos.
    mock_get.return_value = Mock(ok=True)
    mock_get.return_value.json.return_value = todos

    # Call the service, which will send a request to the server.
    response = get_todos()

    # If the request is sent successfully, then I expect a response to be returned.
    assert_list_equal(response.json(), todos)


@patch('project.services.requests.get')
def test_getting_todos_when_response_is_not_ok(mock_get):
    # Configure the mock to not return a response with an OK status code.
    mock_get.return_value.ok = False

    # Call the service, which will send a request to the server.
    response = get_todos()

    # If the response contains an error, I should get no todos.
    assert_is_none(response)
```

I mentioned in a previous example that when you ran the `get_todos()` function that was patched with a mock, the function returned a mock object "response". You might have noticed a pattern: whenever the `return_value` is added to a mock, that mock is modified to be run as a function, and by default it returns another mock object. In this example, I made that a little more clear by explicitly declaring the `Mock` object, `mock_get.return_value = Mock(ok=True)`. The `mock_get()` mirrors `requests.get()`, and `requests.get()` returns a `Response` whereas `mock_get()` returns a `Mock`. The `Response` object has an `ok` property, so you added an `ok` property to the `Mock`.

The `Response` object also has a `json()` function, so I added `json` to the `Mock` and appended it with a `return_value`, since it will be called like a function. The `json()` function returns a list of todo objects. Notice that the test now includes an assertion that checks the value of `response.json()`. You want to make sure that the `get_todos()` function returns a list of todos, just like the actual server does. Finally, to round out the testing for `get_todos()`, I add a test for failure.

Run the tests and watch them pass.

Shell

```
$ nosetests --verbosity=2 project
test_todos.test_getting_todos_when_response_is_not_ok ... ok
test_todos.test_getting_todos_when_response_is_ok ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.285s

OK
```

Mocking integrated functions

The examples I have show you have been fairly straightforward, and in the next example, I will add to the complexity. Imagine a scenario where you create a new service function that calls `get_todos()` and then filters those results to return only the todo items that have been completed. Do you have to mock the `requests.get()` again? No, in this case you mock the `get_todos()` function directly! Remember, when you mock a function, you are replacing the actual object with the mock, and you only have to worry about how the service function interacts with that mock. In the case of `get_todos()`, you know that it takes no parameters and that it returns a response with a `json()` function that returns a list of todo objects. You do not care what happens under the hood; you just care that the `get_todos()` mock returns what you expect the real `get_todos()` function to return.

project/tests/test_todos.py

```
Python
```

```
# Standard library imports...
from unittest.mock import Mock, patch

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

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


@patch('project.services.get_todos')
def test_getting_uncompleted_todos_when_todos_is_not_none(mock_get_todos):
    todo1 = {
        'userId': 1,
        'id': 1,
        'title': 'Make the bed',
        'completed': False
    }
    todo2 = {
        'userId': 1,
        'id': 2,
        'title': 'Walk the dog',
        'completed': True
    }

    # Configure mock to return a response with a JSON-serialized list of todos.
    mock_get_todos.return_value = Mock()
    mock_get_todos.return_value.json.return_value = [todo1, todo2]

    # Call the service, which will get a list of todos filtered on completed.
    uncompleted_todos = get_uncompleted_todos()

    # Confirm that the mock was called.
    assert_true(mock_get_todos.called)

    # Confirm that the expected filtered list of todos was returned.
    assert_list_equal(uncompleted_todos, [todo1])


@patch('project.services.get_todos')
def test_getting_uncompleted_todos_when_todos_is_none(mock_get_todos):
    # Configure mock to return None.
    mock_get_todos.return_value = None

    # Call the service, which will return an empty list.
    uncompleted_todos = get_uncompleted_todos()

    # Confirm that the mock was called.
    assert_true(mock_get_todos.called)

    # Confirm that an empty list was returned.
    assert_list_equal(uncompleted_todos, [])
```

Notice that now I am patching the test function to find and replace `project.services.get_todos` with a mock. The mock function should return an object that has a `json()` function. When called, the `json()` function should return a list of todo objects. I also add an assertion to confirm that the `get_todos()` function is actually called. This is useful to establish that when the service function accesses the actual API, the real `get_todos()` function will execute. Here, I also include a test to verify that if `get_todos()` returns `None`, the `get_uncompleted_todos()` function returns an empty list. Again, I confirm that the `get_todos()` function is called.

Write the tests, run them to see that they fail, and then write the code necessary to make them pass.

project/services.py

Python

```
def get_uncompleted_todos():
    response = get_todos()
    if response is None:
        return []
    else:
        todos = response.json()
        return [todo for todo in todos if todo.get('completed') == False]
```

The tests now pass.

Refactoring tests to use classes

You probably noticed that some of the tests seem to belong together in a group. You have two tests that hit the `get_todos()` function. Your other two tests focus on `get_uncompleted_todos()`. Whenever I start to notice trends and similarities between tests, I refactor them into a test class. This refactoring accomplishes several goals:

1. Moving common test functions to a class allows you to more easily test them together as a group. You can tell *nose* to target a list of functions, but it is easier to target a single class.
2. Common test functions often require similar steps for creating and destroying data that is used by each test. These steps can be encapsulated in the `setup_class()` and `teardown_class()` functions respectively in order to execute code at the appropriate stages.
3. You can create utility functions on the class to reuse logic that is repeated among test functions. Imagine having to call the same data creation logic in each function individually. What a pain!

Notice that I use the **patcher** technique to mock the targeted functions in the test classes. As I mentioned before, this patching method is great for creating a mock that spans over several functions. The code in the `teardown_class()` method explicitly restores the original code when the tests finish.

project/tests/test_todos.py

Python

```
# Standard library imports...
from unittest.mock import Mock, patch

# Third-party imports...
from nose.tools import assert_is_none, assert_list_equal, assert_true

# Local imports...
from project.services import get_todos, get_uncompleted_todos


class TestTodos(object):
    @classmethod
    def setup_class(cls):
        cls.mock_get_patcher = patch('project.services.requests.get')
        cls.mock_get = cls.mock_get_patcher.start()

    @classmethod
    def teardown_class(cls):
        cls.mock_get_patcher.stop()

    def test_getting_todos_when_response_is_ok(self):
        # Configure the mock to return a response with an OK status code.
        self.mock_get.return_value.ok = True

        todos = [{
            'userId': 1,
            'id': 1,
            'title': 'Make the bed',
            'completed': False
        }]

        self.mock_get.return_value = Mock()
        self.mock_get.return_value.json.return_value = todos

        # Call the service, which will send a request to the server.
        response = get_todos()

        # If the request is sent successfully, then I expect a response to be returned.
        assert_list_equal(response.json(), todos)

    def test_getting_todos_when_response_is_not_ok(self):
        # Configure the mock to not return a response with an OK status code.
        self.mock_get.return_value.ok = False

        # Call the service, which will send a request to the server.
        response = get_todos()

        # If the response contains an error, I should get no todos.
        assert_is_none(response)


class TestUncompletedTodos(object):
    @classmethod
    def setup_class(cls):
        cls.mock_get_todos_patcher = patch('project.services.get_todos')
        cls.mock_get_todos = cls.mock_get_todos_patcher.start()

    @classmethod
    def teardown_class(cls):
        cls.mock_get_todos_patcher.stop()

    def test_getting_uncompleted_todos_when_todos_is_not_none(self):
        todo1 = {
            'userId': 1,
            'id': 1,
            'title': 'Make the bed',
            'completed': False
        }
        todo2 = {
```

```
            'userId': 2,
            'id': 2,
            'title': 'Walk the dog',
            'completed': True
        }

        # Configure mock to return a response with a JSON-serialized list of todos.
        self.mock_get_todos.return_value = Mock()
        self.mock_get_todos.return_value.json.return_value = [todo1, todo2]

        # Call the service, which will get a list of todos filtered on completed.
        uncompleted_todos = get_uncompleted_todos()

        # Confirm that the mock was called.
        assert_true(self.mock_get_todos.called)

        # Confirm that the expected filtered list of todos was returned.
        assert_list_equal(uncompleted_todos, [todo1])

    def test_getting_uncompleted_todos_when_todos_is_none(self):
        # Configure mock to return None.
        self.mock_get_todos.return_value = None

        # Call the service, which will return an empty list.
        uncompleted_todos = get_uncompleted_todos()

        # Confirm that the mock was called.
        assert_true(self.mock_get_todos.called)

        # Confirm that an empty list was returned.
        assert_list_equal(uncompleted_todos, [])
```

Run the tests. Everything should pass because you did not introduce any new logic. You merely moved code around.

Shell

```
$ nosetests --verbosity=2 project
test_todos.TestTodos.test_getting_todos_when_response_is_not_ok ... ok
test_todos.TestTodos.test_getting_todos_when_response_is_ok ... ok
test_todos.TestUncompletedTodos.test_getting_uncompleted_todos_when_todos_is_none ... ok
test_todos.TestUncompletedTodos.test_getting_uncompleted_todos_when_todos_is_not_none ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.300s

OK
```

Testing for updates to the API data

Throughout this tutorial I have been demonstrating how to mock data returned by a third-party API. That mock data is based on an assumption that the real data uses the same data contract as what you are faking. Your first step was making a call to the actual API and taking note of the data that was returned. You can be fairly confident that the structure of the data has not changed in the short time that you have been working through these examples, however, you should not be confident that the data will remain unchanged forever. Any good external library is updated regularly. While developers aim to make new code backwards-compatible, eventually there comes a time where code is deprecated.

As you can imagine, relying entirely on fake data is dangerous. Since you are testing your code without communicating with the actual server, you can easily become overconfident in the strength of your tests. When the time comes to use your application with real data, everything falls apart. The following strategy should be used to confirm that the data you are expecting from the server matches the data that you are testing. *The goal here is to compare the data structure (e.g. the keys in an object) rather than the actual data.*

Notice how I am using the **context manager** patching technique. Here, you need to call the real server *and* you need to mock it separately.

project/tests/test_todos.py

Python

```
def test_integration_contract():
    # Call the service to hit the actual API.
    actual = get_todos()
    actual_keys = actual.json().pop().keys()

    # Call the service to hit the mocked API.
    with patch('project.services.requests.get') as mock_get:
        mock_get.return_value.ok = True
        mock_get.return_value.json.return_value = [{
            'userId': 1,
            'id': 1,
            'title': 'Make the bed',
            'completed': False
        }]

        mocked = get_todos()
        mocked_keys = mocked.json().pop().keys()

    # An object from the actual API and an object from the mocked API should have
    # the same data structure.
    assert_list_equal(list(actual_keys), list(mocked_keys))
```

Your tests should pass. Your mocked data structure matches the one from the actual API.

Conditionally testing scenarios

Now that you have a test to compare the actual data contracts with the mocked ones, you need to know when to run it. The test that hits the real server should not be automated because a failure does not necessarily mean your code is bad. You might not be able to connect to the real server at the time of your test suite execution for a dozen reasons that are outside of your control. Run this test separately from your automated tests, but also run it fairly frequently. One way to selectively skip tests is to use an environment variable as a toggle. In the example below, all tests run unless the `SKIP_REAL` environment variable is set to `True`. When the `SKIP_REAL` variable is toggled on, any test with the `@skipIf(SKIP_REAL)` decorator will be skipped.

project/tests/test_todos.py

Python

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

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


@skipIf(SKIP_REAL, 'Skipping tests that hit the real API server.')
def test_integration_contract():
    # Call the service to hit the actual API.
    actual = get_todos()
    actual_keys = actual.json().pop().keys()

    # Call the service to hit the mocked API.
    with patch('project.services.requests.get') as mock_get:
        mock_get.return_value.ok = True
        mock_get.return_value.json.return_value = [{
            'userId': 1,
            'id': 1,
            'title': 'Make the bed',
            'completed': False
        }]

        mocked = get_todos()
        mocked_keys = mocked.json().pop().keys()

    # An object from the actual API and an object from the mocked API should have
    # the same data structure.
    assert_list_equal(list(actual_keys), list(mocked_keys))
```

project/constants.py

Python

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


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

Shell

```
$ export SKIP_REAL=True
```

Run the tests and pay attention to the output. One test was ignored and the console displays the message, “Skipping tests that hit the real API server.” Excellent!

Shell

```
$ nosetests --verbosity=2 project
test_todos.TestTodos.test_getting_todos_when_response_is_not_ok ... ok
test_todos.TestTodos.test_getting_todos_when_response_is_ok ... ok
test_todos.TestUncompletedTodos.test_getting_uncompleted_todos_when_todos_is_none ... ok
test_todos.TestUncompletedTodos.test_getting_uncompleted_todos_when_todos_is_not_none ... ok
test_todos.test_integration_contract ... SKIP: Skipping tests that hit the real API server.

----------------------------------------------------------------------
Ran 5 tests in 0.240s

OK (SKIP=1)
```

Next steps

At this point, you have seen how to test the integration of your app with a third-party API using mocks. Now that you know how to approach the problem, you can continue practicing by writing service functions that for the other API endpoints in JSON Placeholder (e.g. posts, comments, users).

Improve your skills even more by connecting your app to a real external library such as Google, Facebook, or Evernote and see if you can write tests that use mocks. Keep producing clean and reliable code and stay tuned for the next tutorial, which will describe how to take testing to the next level with mock servers!

Grab the code from the repo.