

BUT HOW DO YOU KNOW
YOUR MOCK IS VALID?

VERIFIED FAKES OF WEB SERVICES

 @adamdangoor

 adamtheturtle

BUT HOW DO YOU KNOW
YOUR MOCK IS VALID?

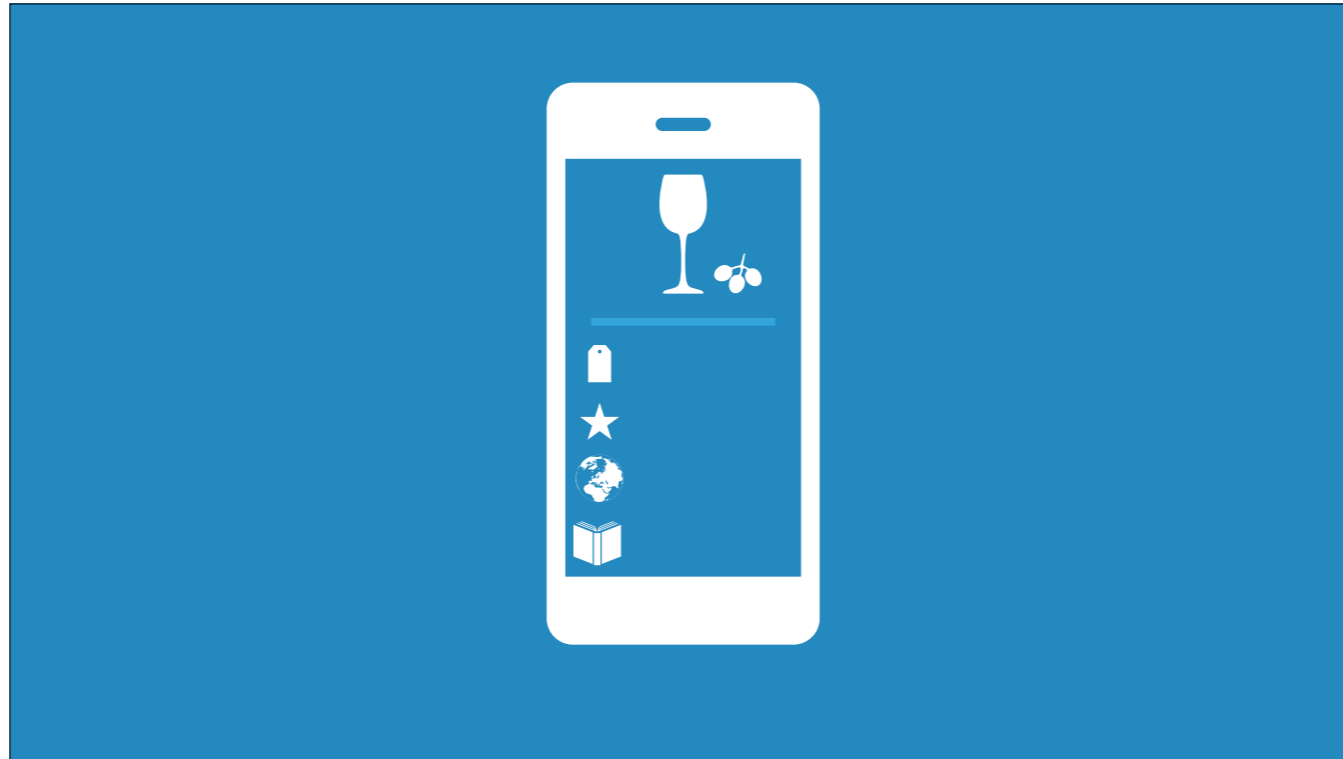
VERIFIED FAKES OF WEB SERVICES

 @adamdangoor

 adamtheturtle

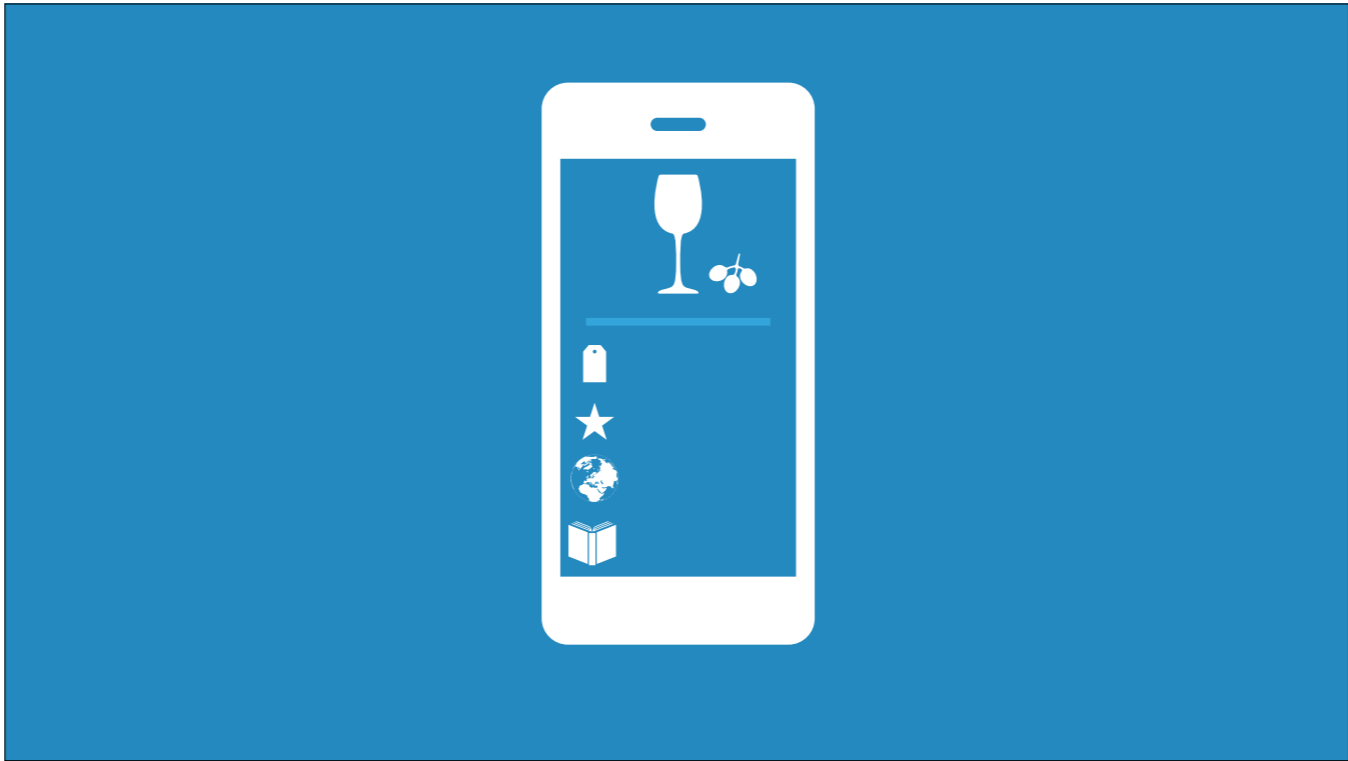
Hi, I'm Adam Dangoor,

I work at Mesosphere building an operating system for data centres.



But last year I was working on the back end of an iPhone app.

You would take a picture of a wine label with your phone and the app would tell you all kinds of details about the wine.



Or at least that's close enough to protect my NDA

FLASK



```
@app.route('/user/<username>')  
def show_user_profile(username):  
    return f'User {username}'
```

Our app was a Flask app.

If you don't know Flask, it is a pretty simple web framework, and it looks something like this.

FLASK TEST CLIENT



```
def test_add_image():
    test_client = flask_app.app.test_client()
    result = app.post(
        '/targets/',
        data=dict(image=b64encode(image_data)),
    )
    assert result.data == '{"status": "added"}'
```

And a really cool thing about Flask is that it provides a Werkzeug test client.

That means you can make requests against an in memory application and get response objects which you can inspect.

FLASK TEST CLIENT



```
def test_add_image():
    test_client = flask_app.app.test_client()
    result = app.post(
        '/targets/',
        data=dict(image=b64encode(image_data)),
    )
    assert result.data == '{"status": "added"}'
```

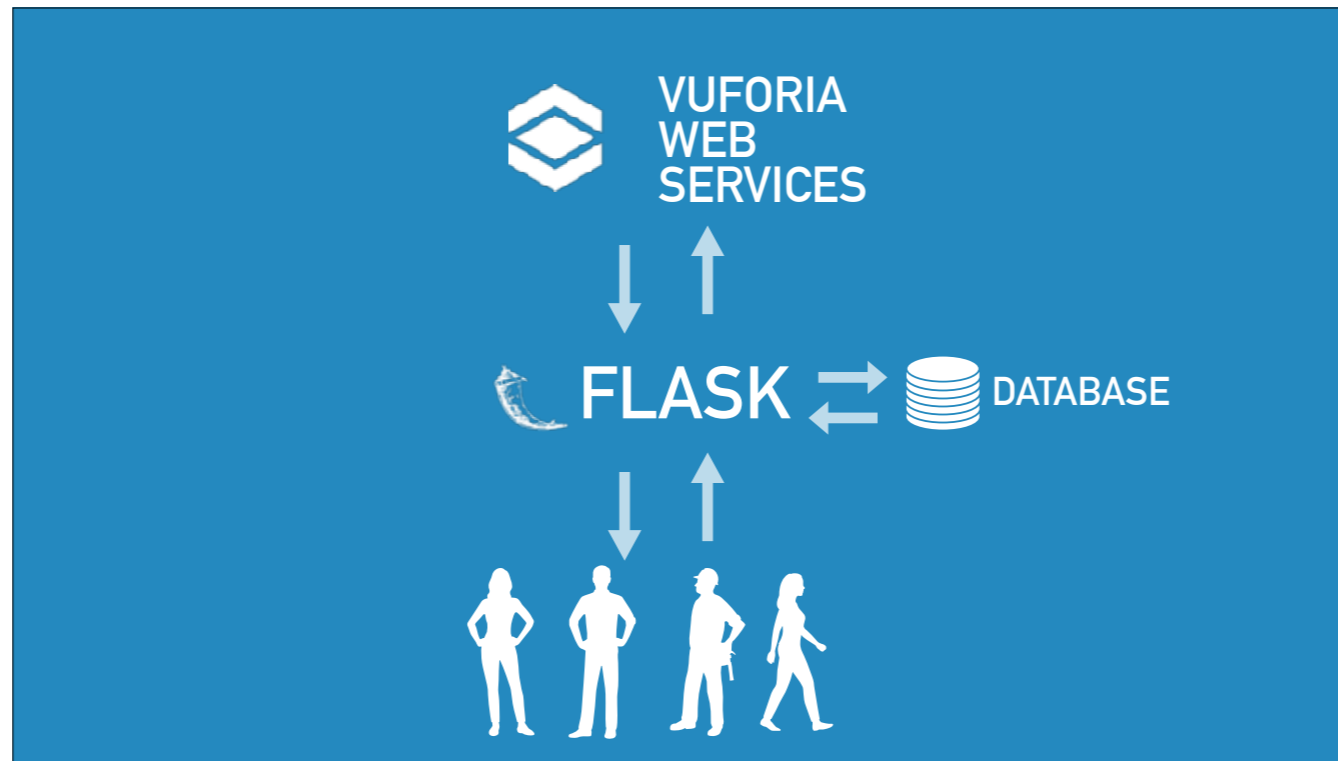
If we look at this test here, everything is happening in memory, but it kind of looks like we've made an HTTP request.



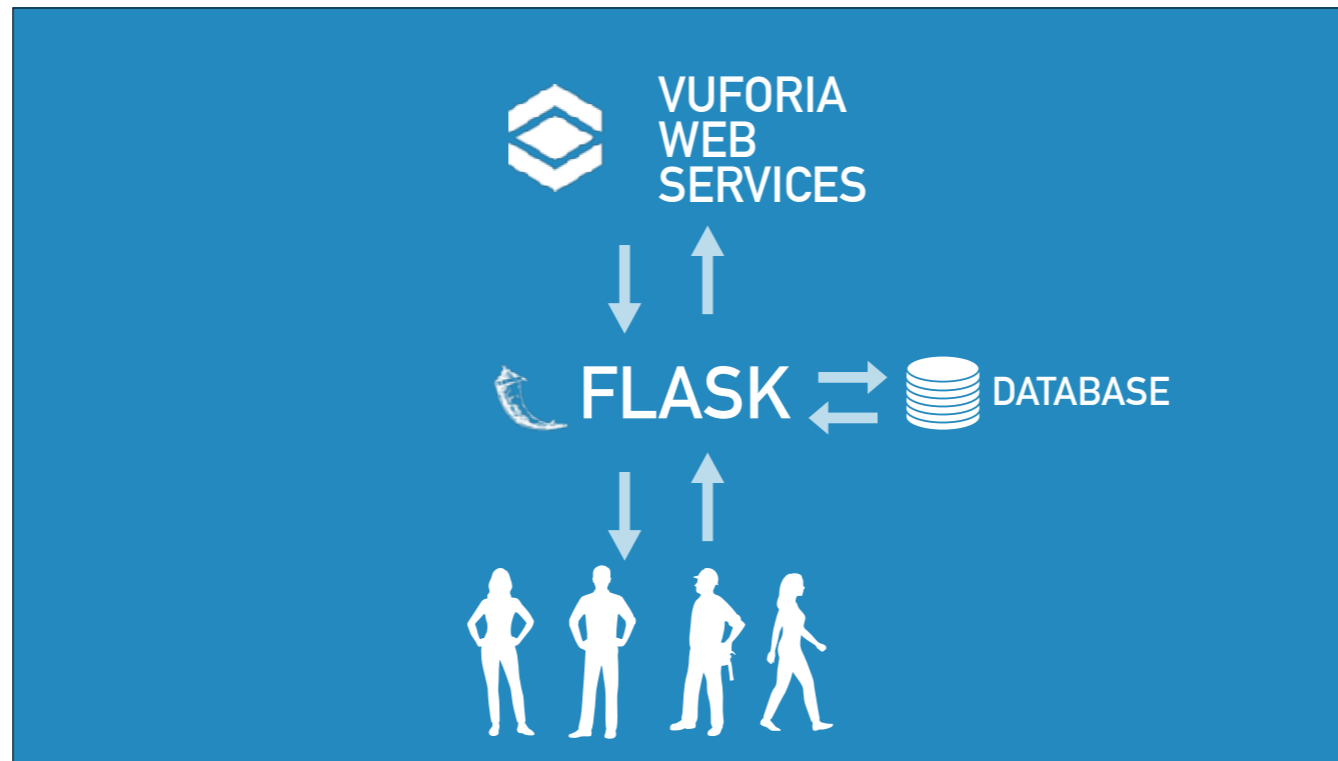
In our app we used something called Vuforia Web Services.



Basically it let us upload a whole bunch of images of wine labels and then when a user uploaded a photo to us, we could send that image to Vuforia and Vuforia would tell us which of our previously uploaded images their photo most closely matched.

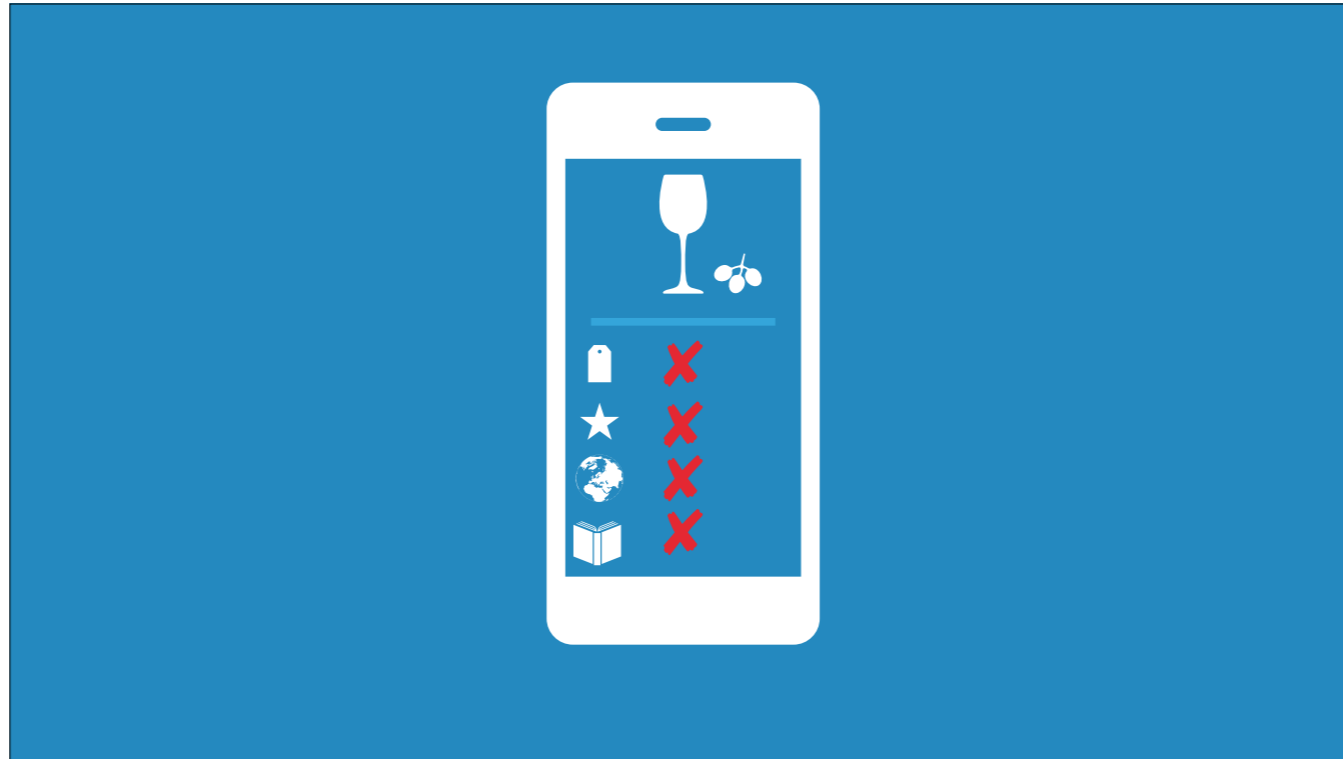


Then we could fetch the data from our database and tell the user details about the wine like how much it should cost, how well it was rated.

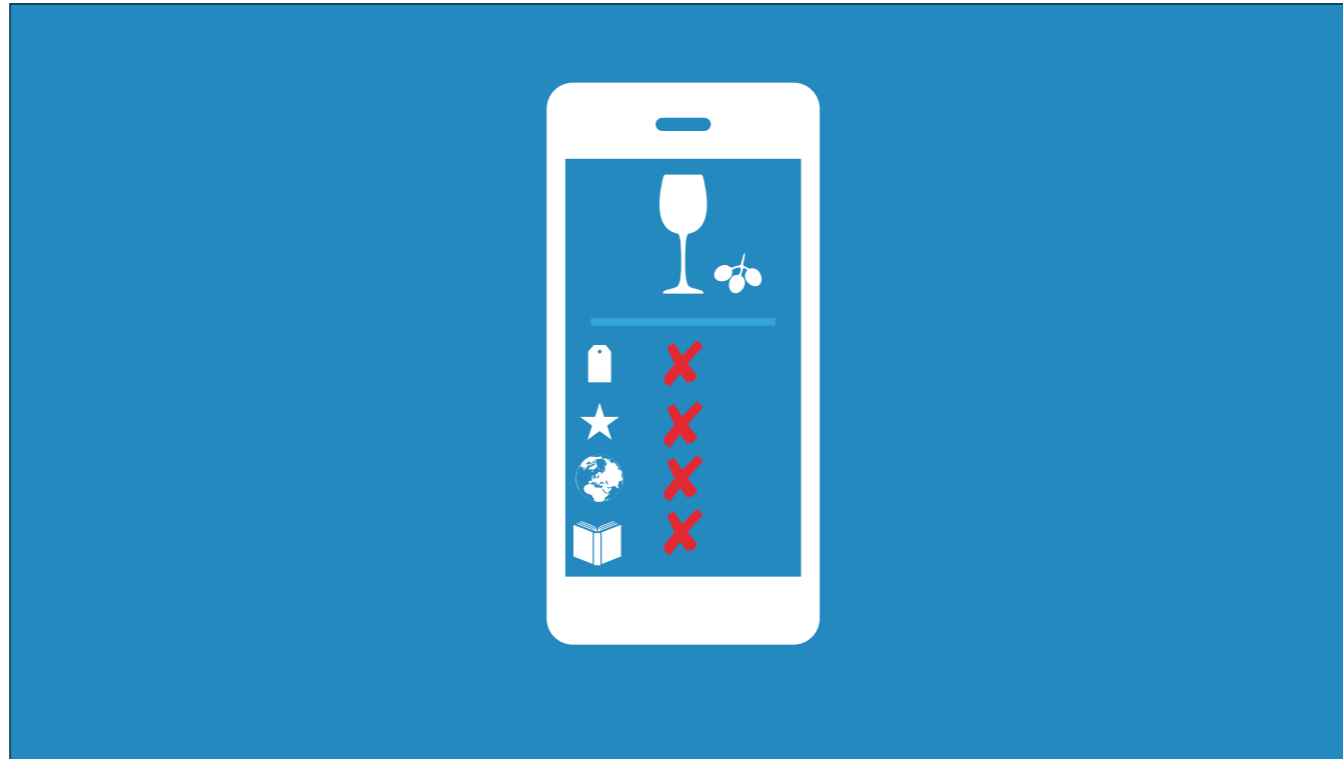


But when we built our prototype we kept finding problems.

In particular, we had made assumptions about Vuforia which weren't quite right.

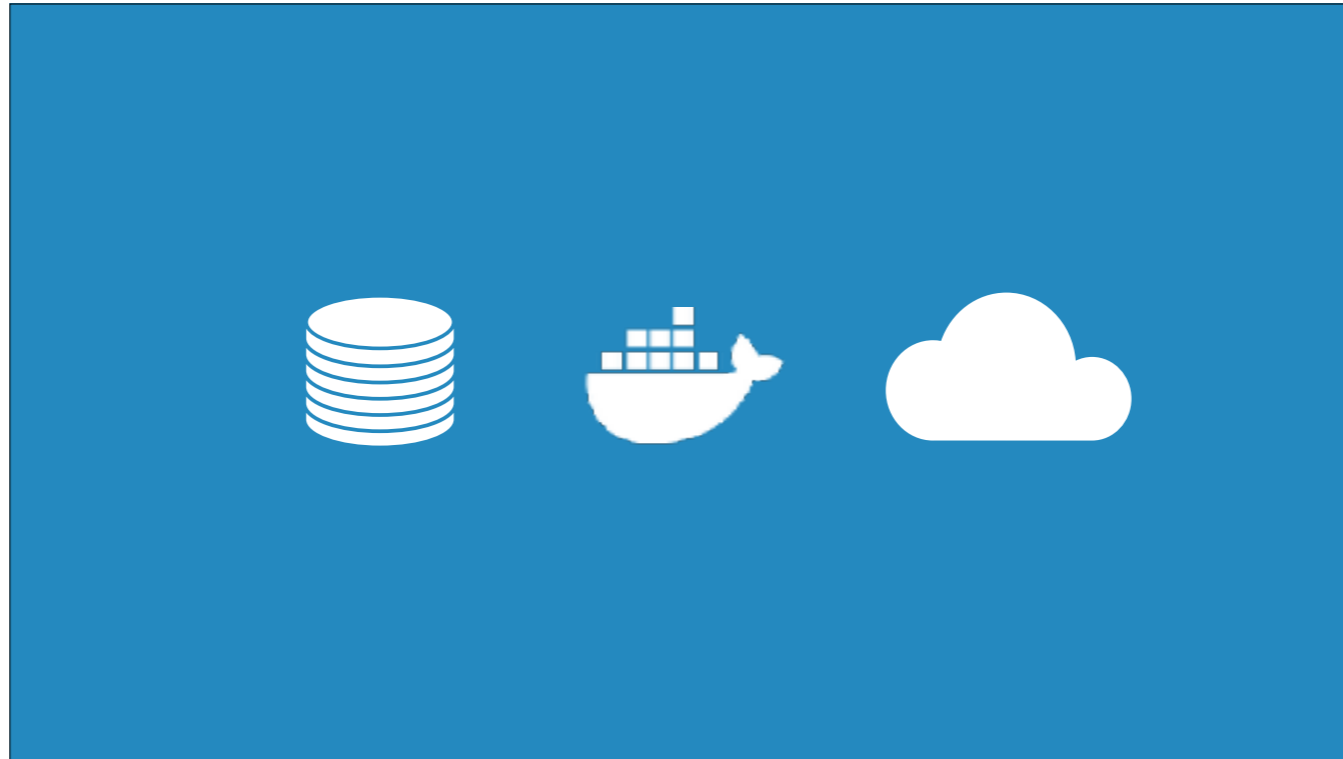


And so we wanted to add tests for our matching workflow, which used Vuforia, to the test suite.



Vuforia here was accessed over HTTP, and that's what I'm going to focus on today.

But the general ideas aren't specific to HTTP.



You might want to test code which uses a database for local storage.

Or you might want to test deployment tooling which uses Docker.

Or you might want to test code which uses an S3 backend.

INITIAL TEST CASE

```
def get_match(image):  
    ...  
    response = requests.request(  
        method='GET',  
        url=urljoin('https://vws.vuforia.com/', request_path),  
        data=data,  
    )  
    return wine_from_db(id=response.json()['id'])  
  
def test_match_wine():  
    add_wine(image=blue_nun, name="Blue Nun")  
    add_wine(image=cristal, name="Cristal")  
  
    result = get_match(image=blue_nun)  
    assert result.name == "Blue Nun"
```

We had a clear idea of what we wanted the first test to be.

INITIAL TEST CASE

```
def get_match(image):
    ...
    response = requests.request(
        method='GET',
        url=urljoin('https://vws.vuforia.com/', request_path),
        data=data,
    )
    return wine_from_db(id=response.json()['id'])

def test_match_wine():
    add_wine(image=blue_nun, name="Blue Nun")
    add_wine(image=cristal, name="Cristal")

    result = get_match(image=blue_nun)
    assert result.name == "Blue Nun"
```

We wanted to test that when a user uploaded a photo of a wine label which matched a photo we'd already added, they would get the details of the right wine.

So I wrote a test that looked a little bit like this.

INITIAL TEST CASE

```
def get_match(image):  
    ...  
    response = requests.request(  
        method='GET',  
        url=urljoin('https://vws.vuforia.com/', request_path),  
        data=data,  
    )  
    return wine_from_db(id=response.json()['id'])  
  
def test_match_wine():  
    add_wine(image=blue_nun, name="Blue Nun")  
    add_wine(image=cristal, name="Cristal")  
  
    result = get_match(image=blue_nun)  
    assert result.name == "Blue Nun"
```

I add two wines.

That adds them to the database, and it also uploads them to Vuforia

INITIAL TEST CASE

```
def get_match(image):  
    ...  
    response = requests.request(  
        method='GET',  
        url=urljoin('https://vws.vuforia.com/', request_path),  
        data=data,  
    )  
    return wine_from_db(id=response.json()['id'])  
  
def test_match_wine():  
    add_wine(image=blue_nun, name="Blue Nun")  
    add_wine(image=cristal, name="Cristal")  
  
    result = get_match(image=blue_nun)  
    assert result.name == "Blue Nun"
```

And then I check that I get the right one back when I query the match function.

Now with some third party tools, you might be completely fine just calling the real tool in your test suite.

PROBLEMS

But when we called Vuforia in the tests, we hit some problems.

PROBLEMS

◎ FLAKY TESTS

First of all we were at the mercy of the network.

If our CI system had a network glitch, our test suite would fail because our tests made HTTP requests over the internet.

PROBLEMS

- ◎ FLAKY TESTS

But also we were at the mercy of Vuforia - if their service went down, our test suite would fail.

PROBLEMS

- ◎ FLAKY TESTS

- ◎ € € €

And if you're using a real service like S3 in your test suite, something which charges you per megabyte, you might have to pay a decent amount of money to run your tests.

PROBLEMS

- ◎ FLAKY TESTS

- ◎ € € €

Some services also have resource limits which can make it hard to run tests against them.

PROBLEMS

- ⦿ FLAKY TESTS
- ⦿ € € €
- ⦿ SLOW TESTS

And even when those weren't problems, everything was slow.

Vuforia does some processing magic and so it takes a few minutes before you can match images, but we didn't want to have to wait a few minutes in our test suite to know if our matching code worked.

INTEGRATION
TESTS

UNIT
TESTS

So we called these tests integration tests - they tested the integration of our software with Vuforia's.

And these were useful, they helped us track down bugs.

INTEGRATION
TESTS

UNIT
TESTS

But we also wanted unit tests because unit tests give us a lot of benefits over integration tests.

INTEGRATION TESTS

- ▶ COVER A LOT OF CODE PER TEST

UNIT TESTS

- ▶ COVER LESS CODE PER TEST

They tell us if our code calls Vuforia correctly, even when Vuforia is down.

And unit tests are small in scope so if one fails then you often know exactly which part of the code failed.

INTEGRATION TESTS

- ▶ COVER A LOT OF CODE PER TEST
- ▶ CAN BE SLOW

UNIT TESTS

- ▶ COVER LESS CODE PER TEST
- ▶ FAST

You can even have a tool like Hypothesis generate a whole bunch of them.

INTEGRATION TESTS

- ▶ COVER A LOT OF CODE PER TEST
- ▶ CAN BE SLOW

UNIT TESTS

- ▶ COVER LESS CODE PER TEST
- ▶ FAST

You can even have a tool like Hypothesis generate a whole bunch of them.

INTEGRATION TESTS

- ▶ COVER A LOT OF CODE PER TEST
- ▶ CAN BE SLOW

UNIT TESTS

- ▶ COVER LESS CODE PER TEST
- ▶ FAST

So we want to turn a codebase which can currently be tested only by integration tests into one which can also be tested with unit tests.

MOCKS

And one way that people achieve this is by using mocks.

A mock is some code which provides the same interface as something your code calls, but it reduces or removes some cost.

MOCKS

In this case the main costs we cared about were time and flakiness.

But again you might want to avoid financial costs, or resource limits.

MOCKS

So my goal was that wherever code under test made a request to Vuforia, my tests would make sure that that request was handled by a mock function.

MOCKING REQUESTS

```
$ pip install requests-mock
```

```
with requests_mock.mock() as mock:  
    mock.get('http://test.com', text='data')  
    print(requests.get('http://test.com').text)  
    # prints 'data'
```

Luckily we were using the `requests` library and there are a few ways to get requests made with `requests` to point to some mock code.

The tool I chose to use was requests-mock.

MOCKING REQUESTS

```
$ pip install requests-mock
```

```
with requests_mock.mock() as mock:  
    mock.get('http://test.com', text='data')  
    print(requests.get('http://test.com').text)  
    # prints 'data'
```

Now the simple requests mock example is this.

You can say "If there's a GET request to test.com, give back data"

PYTEST FIXTURES



```
$ pip install pytest
```

```
@pytest.fixture()
def mock_vuforia() -> Generator:
    with requests_mock.mock() as mock:
        mock.get('http://test.com', text='data')
        ...
    yield

@pytest.mark.usefixture('mock_vuforia')
def test_add_target():
    ...
```

And we also used pytest.

Pytest is a test runner which gives you a neat way to do set up and tear down for test requirements.

PYTEST FIXTURES

```
$ pip install pytest
```

```
@pytest.fixture()
def mock_vuforia() -> Generator:
    with requests_mock.mock() as mock:
        mock.get('http://test.com', text='data')
        ...
    yield

@pytest.mark.usefixture('mock_vuforia')
def test_add_target():
    ...
```

That feature is called fixtures and we have one here.

PYTEST FIXTURES

```
$ pip install pytest
```

```
@pytest.fixture()
def mock_vuforia() -> Generator:
    with requests_mock.mock() as mock:
        mock.get('http://test.com', text='data')
        ...
    yield

@pytest.mark.usefixture('mock_vuforia')
def test_add_target():
    ...
```

This one says - If I use this fixture, requests made in the test will be handled by mock code.

REQUESTS-MOCK DYNAMIC RESPONSES

```
def add_target_to_database(request, context):
    context.headers['Date'] = rfc_1123_date()
    return json.dumps({'status': 'added'})

@pytest.fixture()
def mock_vuforia() -> Generator:
    with requests_mock.mock() as mock:
        m.register_uri(
            'POST',
            re.compile('http://vws.vuforia.com/targets/\w+'),
            text=add_target_to_database,
        )
        ...
    yield
```

Now I wanted some quite advanced features in my mock.

In particular, I wanted to have a stateful mock to allow me to give different responses based on previous requests.

REQUESTS-MOCK DYNAMIC RESPONSES

```
def add_target_to_database(request, context):
    context.headers['Date'] = rfc_1123_date()
    return json.dumps({'status': 'added'})

@pytest.fixture()
def mock_vuforia() -> Generator:
    with requests_mock.mock() as mock:
        m.register_uri(
            'POST',
            re.compile('http://vws.vuforia.com/targets/\w+'),
            text=add_target_to_database,
        )
        ...
    yield
```

So I used a `requests_mock` feature which let me use a callable instead of a predefined response, and that callable takes a request object.

REQUESTS-MOCK DYNAMIC RESPONSES

```
def add_target_to_database(request, context):
    context.headers['Date'] = rfc_1123_date()
    return json.dumps({'status': 'added'})

@pytest.fixture()
def mock_vuforia() -> Generator:
    with requests_mock.mock() as mock:
        m.register_uri(
            'POST',
            re.compile('http://vws.vuforia.com/targets/\w+'),
            text=add_target_to_database,
        )
        ...
    yield
```

So we created some small mock functions for every endpoint we used.

And at this point we had achieved our initial goal.

We could test our code without touching the real Vuforia.

PROBLEMS WITH MOCKS

But when we were using the mock, we hit some major problems.

And these are problems that a lot of mocks face.

PROBLEMS WITH MOCKS

- ◎ HUMAN ERROR

Sometimes we found that we had copied the interface incorrectly.

PROBLEMS WITH MOCKS

- ◎ HUMAN ERROR
- ◎ BECOME OUTDATED

But even when we were very careful, we found that the mock quickly became outdated when Vuforia changed.

PROBLEMS WITH MOCKS

- ◎ HUMAN ERROR
- ◎ BECOME OUTDATED

And when you have an outdated mock, your tests can pass while your software fails in production, and it can be hard to track down where your mock needs to change when the third party code changes.

VWS-PYTHON

COMING SOON TO PYPI

So that contract ended and I felt that I'd built an OK solution which worked for the client, but I knew that it could be done better because we kept hitting those issues.

VWS-PYTHON

COMING SOON TO PYPI

And at the same time, I believed that Vuforia could be genuinely useful for a bunch of people if it were easier to develop with.

So I set out to make VWS-Python, an open source library for using Vuforia Web Services with Python.

GOALS

- ◎ PYTHON LIBRARY FOR VUFORIA
- ◎ MOCK ANYONE CAN USE

And I also started testing it with an open source mock.

I wanted people to be able to use my mock to test their code whether or not they were using my library.

TRAVIS CI

- ◎ FREE FOR OSS
- ◎ SUPPORT FOR SECRETS

So I wrote some integration tests for the library and some unit tests with a mock.

And I put the test suite on Travis CI because I knew it and because it was free for open source projects.

TRAVIS CI

- ◎ FREE FOR OSS
- ◎ SUPPORT FOR SECRETS

One nice feature of Travis is that I can give it credentials to Vuforia without having those show up in my codebase or in the logs.

So every time I made a change to the library, the tests were run and the integration tests used the real Vuforia.

CROSS- LANGUAGE SUPPORT



Now if you remember the goal I set.

I wanted people to be able to use my mock to test their code whether or not they were using my library.

CROSS- LANGUAGE SUPPORT



And there's a cool way to let even people using different languages use the mock while keeping the interface nice for Python users.

CROSS- LANGUAGE SUPPORT



That way is to build the mock in a way that it can be run as a standalone server.

That means ditching the requests mock syntax we saw before.

CROSS- LANGUAGE SUPPORT



But I wanted to keep the conveniences of using requests-mock for a completely in memory mock.

```
for rule in flask_app.url_map.iter_rules():
    pattern = urljoin(
        'http://vws.vuforia.com',
        re.sub(pattern='<.+>', repl='.', string=rule.rule),
    )

    for method in rule.methods:
        responses.add_callback(
            method=getattr(responses, method),
            url=re.compile(pattern),
            callback=self.request_callback,
        )

def request_callback(self, request):
    response = getattr(flask_app, request.method.lower())(
        request.path_url,
        content_type=request.headers['Content-Type'],
        data=request.body,
    )

    return (
        response.status_code,
        {key: value for (key, value) in response.headers},
        response.data,
    )
```

So I wrote a little bit of code that let me rewrite the mock as a flask app and keep using requests_mock.

```
for rule in flask_app.url_map.iter_rules():
    pattern = urljoin(
        'http://vws.vuforia.com',
        re.sub(pattern='<.+>', repl='.+', string=rule.rule),
    )

    for method in rule.methods:
        responses.add_callback(
            method=getattr(responses, method),
            url=re.compile(pattern),
            callback=self.request_callback,
        )

def request_callback(self, request):
    response = getattr(flask_app, request.method.lower())(
        request.path_url,
        content_type=request.headers['Content-Type'],
        data=request.body,
    )

    return (
        response.status_code,
        {key: value for (key, value) in response.headers},
        response.data,
    )
```

I won't go into it too much because it is a bit of a hairy hack, but basically this translates requests in the form `requests_mock` gives them to you into calls to a flask test client, and then you translate the responses from the test client into something `requests_mock` likes the look of.


```
for rule in flask_app.url_map.iter_rules():
    pattern = urljoin(
        'http://vws.vuforia.com',
        re.sub(pattern='<.+>', repl='.+', string=rule.rule),
    )

    for method in rule.methods:
        responses.add_callback(
            method=getattr(responses, method),
            url=re.compile(pattern),
            callback=self.request_callback,
        )

def request_callback(self, request):
    response = getattr(flask_app, request.method.lower())(
        request.path_url,
        content_type=request.headers['Content-Type'],
        data=request.body,
    )

    return (
        response.status_code,
        {key: value for (key, value) in response.headers},
        response.data,
    )
```

So with this, I could continue to have a completely in memory mock for Python users, and keep the Flask app for other users, or users of old Python versions.

DOCKER



If you're not using Python, can spin up the Flask app in a Docker container for every test, and route requests to that container.

So if you're thinking about writing a mock in your organisation, and you use multiple languages, this can be one way to do it.

WRITING A MOCK

So back to writing the mock.

This time around the mock was part of my product and so I wrote unit tests that confirmed it was doing what I wanted.

And if you think about it, this is probably duplicating the work that the Vuforia authors did.

WRITING A MOCK




I'm rewriting a small part of their service and now I'm testing it.

I'm thinking of edge cases and I'm manually making requests to the Vuforia servers with those edge cases and then copying the response results into my mock.

WRITING A MOCK

I test things that aren't mentioned in the documentation like giving the image a negative width, and then I'm copying the error messages into my mock so that my library can give a nice Python exception for that error.

TEST LAYERS

- LIBRARY INTEGRATION TESTS 
- LIBRARY UNIT TESTS 
- TESTS FOR THE MOCK 

So at this point I have three sets of tests:

- * I have few integration tests which use test library with the real Vuforia
- * I have a lot unit tests for the library which use the mock.
- * And I have some unit tests for the mock itself

PROBLEMS WITH MOCKS

- ◎ COPYING ERRORS
- ◎ BECOME OUTDATED

But, I'm still vulnerable to the problems I mentioned earlier.

I might be making mistakes copying Vuforia, and Vuforia might change, rendering my mock inaccurate and my library possibly broken.

VERIFIED FAKE

[A verified fake] is verified against (a subset of) the same test suite as the real implementation, as well as providing an introspection API that allows test cases to verify that it did the right thing. – Glyph

Turning the mock into a verified fake is all about avoiding those problems.

A verified fake is a fake implementation which is verified against a subset of the same test suite as the real implementation.

VERIFIED FAKE

[A verified fake] is verified against (a subset of) the same test suite as the real implementation, as well as providing an introspection API that allows test cases to verify that it did the right thing. – Glyph

I don't have access to the Vuforia code or its test suite.

So I want to make a verified fake, I needed to have my own test suite.

So turning VWS-Python's mock into a verified fake involved making a test suite to run against both the mock and the real Vuforia.

PARAMETRIZING PYTEST FIXTURES

```
@pytest.fixture(
    params=[True, False],
    ids=['Real Vuforia', 'Mock Vuforia'])
def verify_fake_vuforia(request):
    use_real_vuforia = request.param

    if use_real_vuforia:
        yield
    else:
        with requests_mock.mock() as mock:
            m.post(
                re.compile('http://vws.vuforia.com/targets/\w+'),
                text=add_target_to_database,
            )
            ...
            yield
```

And so if you recall the pytest fixture from before, I expanded it.

Pytest has this pretty neat feature where you can parametrize fixtures, so that tests which use those fixtures are run once with each parameter option.

PARAMETRIZING PYTEST FIXTURES

```
@pytest.fixture(
    params=[True, False],
    ids=['Real Vuforia', 'Mock Vuforia'])
def verify_fake_vuforia(request):
    use_real_vuforia = request.param

    if use_real_vuforia:
        yield
    else:
        with requests_mock.mock() as mock:
            m.post(
                re.compile('http://vws.vuforia.com/targets/\w+'),
                text=add_target_to_database,
            )
            ...
            yield
```

In this example each test which uses this fixture gets run twice, once with the real vuforia and once with the mock vuforia.

TEST RESULTS

```
test_add_target.py::test_existing_target_name[Real Vuforia] PASSED  
test_add_target.py::test_existing_target_name[Mock Vuforia] PASSED
```

And the results show that each test is run twice, and you see the result on the real Vuforia and the mock

So I used that fixture on the tests which tested the mock.

TEST RESULTS

```
test_add_target.py::test_existing_target_name[Real Vuforia] PASSED  
test_add_target.py::test_existing_target_name[Mock Vuforia] PASSED
```

That means that the mock has been turned into a verified fake.

We now have a test suite which runs against both the fake implementation and the real implementation.

BENEFITS

- ◎ WE TRUST THE MOCK...SO WE TRUST THE TESTS WHICH USE THE MOCK

Now, because the mock has been turned into a verified fake, we trust that it is representative of the real Vuforia.

And what this gives me is confidence that the hundreds of tests I have for the library, which use the mock, are valid.

BENEFITS

- ◎ WE TRUST THE MOCK...SO WE TRUST THE TESTS WHICH USE THE MOCK

That is, they don't rely on an unrealistic mock.

BENEFITS

- ◎ TESTS TELL US IF THE MOCK IS OUTDATED

But we also had another problem.

Remember we were worried that Vuforia would change and that would make our mock inaccurate.

BENEFITS

- ◎ TESTS TELL US IF THE MOCK IS OUTDATED

Well now, whenever the tests pass, I know that the mock is still a faithful representation of Vuforia.

We only incur the cost of running a hundred tests against Vuforia, but we get the benefit of running thousands of tests against Vuforia.

SCHEDULED TESTS WITH TRAVIS CI CRON JOBS

But our tests only run when we make a change to the code, which might not be that often.

A cool feature of Travis CI is that you can set tests to run on a schedule.

SCHEDULED TESTS WITH TRAVIS CI CRON JOBS

Remember there's a cost associated with running them, but I do want to know when the mock becomes inaccurate.

SCHEDULED TESTS WITH TRAVIS CI CRON JOBS

So there's a trade-off between running them all the time and running them very rarely.

The trade off I chose was to trigger them every night.

SCHEDULED TESTS WITH TRAVIS CI CRON JOBS

But you can choose to run them every week, just before releases, every four hours, whatever suits.

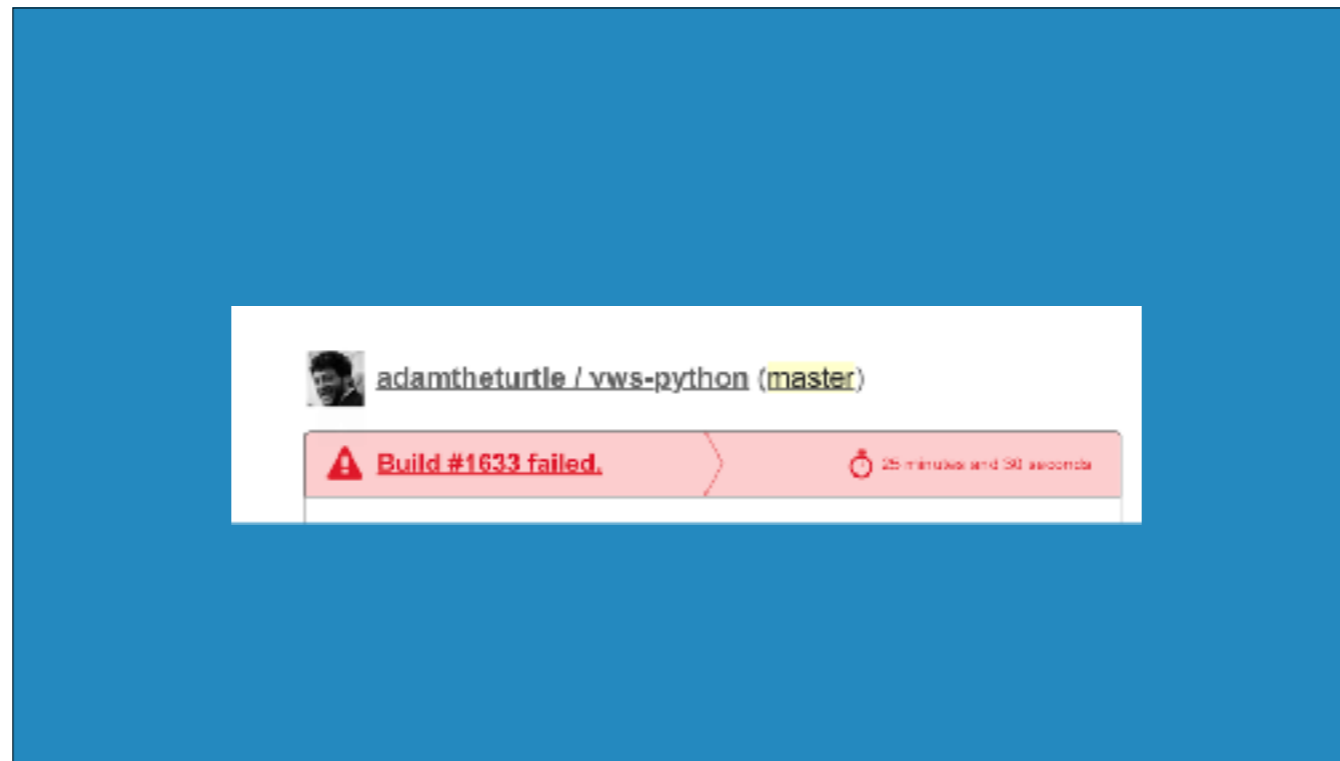
WHEN THE SERVICE CHANGES

In the wine application, we didn't care about the physical width of each wine label because the data was hard to get and it wasn't particularly useful. So we told Vuforia that the width was 0.

This worked and the mock supported it.

WHEN THE SERVICE CHANGES

Fast forward a few months and the new verified fake also supports setting the width to 0, and it checks to make sure that uploading an image with a width of 0 works.



One morning I get an email from Travis telling me that the build failed.

TEST RESULTS

```
test_add_target.py::test_zero_width[Real Vuforia] FAILED  
test_add_target.py::test_zero_width[Mock Vuforia] PASSED
```

When I look at the logs I see that I have a very precise data point of exactly what has changed in Vuforia.

The mock passes the test for setting the width to zero but the real Vuforia fails.

TEST RESULTS

```
test_add_target.py::test_zero_width[Real Vuforia] FAILED  
test_add_target.py::test_zero_width[Mock Vuforia] PASSED
```

I then changed the mock function and the test so that the new behaviour was represented in the mock.

That was easy enough - I just made the mock give an invalid width error when zero was given.

TEST RESULTS

```
test_add_target.py::test_zero_width[Real Vuforia] FAILED  
test_add_target.py::test_zero_width[Mock Vuforia] PASSED
```

But if you remember, the library's tests used the mock.

And the library also expected that a width of 0 was valid.

So as soon as I changed the mock, the library's tests started failing.

TEST RESULTS

```
test_add_target.py::test_zero_width[Real Vuforia] FAILED  
test_add_target.py::test_zero_width[Mock Vuforia] PASSED
```

So I changed the library to give a nice Python exception when you use a width of 0.

That means that within a few hours, Vuforia made an undocumented change which introduced an incompatibility with my library, and this incompatibility was found and fixed without any complex debugging.

TEST RESULTS

```
test_add_target.py::test_zero_width[Real Vuforia] FAILED  
test_add_target.py::test_zero_width[Mock Vuforia] PASSED
```

And I hope with that you see the value of a verified fake to a developer writing code which integrates with third party software.

TEST RESULTS

```
test_add_target.py::test_zero_width[Real Vuforia] FAILED  
test_add_target.py::test_zero_width[Mock Vuforia] PASSED
```

Now you can imagine that building a verified fake when you have the original source code is much simpler than when you don't.

A lot of the fake can share code with the real implementation.

TEST RESULTS

```
test_add_target.py::test_zero_width[Real Vuforia] FAILED  
test_add_target.py::test_zero_width[Mock Vuforia] PASSED
```

But hardly any web services are open source.

SHIPPING A VERIFIED FAKE ADDS VALUE TO YOUR PRODUCT

If you're shipping software which people might want to call in tests, you can add tremendous value to that software by shipping your own verified fake.

It might even cause someone to choose your software over a competitor's.

SHIPPING A VERIFIED FAKE ADDS VALUE TO YOUR PRODUCT

And if you make a verified fake as the author of software, it is much easier, because you can get told before merging code into your software that it would make the fake unrealistic.

That means you know when you need to change the fake to match your changes without the need for that once per day test run.

SHIPPING A VERIFIED FAKE ADDS VALUE TO YOUR PRODUCT

In the last decade, having an API has become vital for almost any web service.

Maybe in the next decade, having an API which is easily tested against will become table stakes as well.

SHIPPING A VERIFIED FAKE ADDS VALUE TO YOUR PRODUCT

And shipping a verified fake is one way to make your service more easily tested against.

You don't have to give your secret sauce away.

SHIPPING A VERIFIED FAKE ADDS VALUE TO YOUR PRODUCT

What I mean by that is, well say with a Vuforia fake, you can have just very basic image matching. In fact that's what my fake does - nothing too fancy.

The idea is to make sure that the API matches the implementation's.

- ◎ WHEN A MOCK IS USEFUL
- ◎ SOME DOWNSIDES OF MOCKS
- ◎ HOW TO TURN A MOCK INTO A VERIFIED FAKE
- ◎ HOW TO MAKE THAT CROSS-LANGUAGE

So I hope you now have an idea of what a verified fake is, why it might be useful, and how you can start making one for yourself or for your users.

CHEERS

 @adamdangoor

 adamtheturtle

Thank you