# Contract Testing with Pact

By Tibs / Tony Ibbs

Presented at CamPUG, virtually, 12th January 2021
Written in reStructuredText, converted to PDF using rst2pdf.
Source and examples at https://github.com/tibs/pact-talk

# What we shall cover

- The problem space
- The testing approaches we might take
- A very simple example
- A second version that is a little more complex
- A third version that has non-static response data
- A few end items

# The problem space

We have two processes that communicate over HTTP

We have control of testing of both of them

We want to test that they agree on that communication

# Note on terminology

Pact talks about "producer" and "consumer" processes.

For historical reasons, I'm going to talk about "server" and "client".

But really, the Pact terms are better.

# Interlude

# Unit tests

I'm going to assume we have unit tests within each service

We're only looking at the boundary between services

# 1. End-to-end tests

The client tests talk to a real instance of the server.

- We need a server running
- Hopefully not the production server
- What version of the server?
- What if it doesn't match what's in production?

Likely to be expensive and slow

# 2. Mock / fake the server responses

- What if we get it wrong - our tests may still pass!
- What version of the server?

Likely to be difficult and unreliable

But does gives us useful documentation of our expectations

# 3. Record real server responses

Use a library like VCR or Betamax to record the actual responses

- What version of the server?

Excellent approach, but we have to remember to update the recordings

# 4. Ideal approach - contract testing

If we're lucky enough to have at least some control over both client and server, we can do better.

Instead of recording the response to our request, we describe it

Or, actually, we describe what we care about in the response

We can then test the client using that description as if it were a recording

And we can test that the server agrees that it can honour that description

In other words, we have a **contract** between client and server.

# But what about the server version?

Different versions of the server may still behave differently

But we can:

- name the contracts
- note which contracts server A and client B should both honour
- test accordingly

# Maybe use a contract broker

In simple systems, it may be enough for the server and client just to agree where the contracts are stored, and retrieve them

But for the complete experience, when generating a contract, also store it in a contract broker

Client and server can then be explicit about exactly which contract versions they both support

We'll come back to this briefly at the end

# Interlude

# A very very simple example

Imagine we are producing a service to make virtual sandwiches

Since we like microservices (a lot) our sandwich assembly service will need a different service to "put butter on things"

---

Let's create `server1` and `client1`

# The "put butter on things" server

```
#!/usr/bin/env python3

from bottle import Bottle

app = Bottle()

@app.route('/butter/<substrate>')
def butter(substrate):
    return f'{substrate} and butter'

if __name__ == '__main__':
    app.run()
```

# and a test for the server

```
#!/usr/bin/env python3

from server1 import butter

def test_butter():
    assert butter('bread') == 'bread and butter'
```

# which passes

```
$ pytest server1_tests.py
============================= test session starts ==============================
platform darwin -- Python 3.8.6, pytest-6.2.1, py-1.10.0, pluggy-0.13.1
rootdir: /Users/tibs/Dropbox/talks/pact-talk/examples/server1
collected 1 item

server1_tests.py .                                                       [100%]

============================== 1 passed in 0.05s ===============================
```

# The sandwich making client

The client for the "put butter on things" server

1. makes a request to get butter on some bread
2. carries on with the rest of the sandwich assembly

---

We're not particularly interested in anything but that first request

And actually, we're really only interested in the *test* for that request

I haven't actually bothered to write the client at all...

# and our test

```
#!/usr/bin/env python3

import requests

BASE_URL = 'http://localhost:8080'

def test_buttering():
    result = requests.get(f'{SERVER_BASE_URL}/butter/bread')
    assert(result.status_code) == 200
    assert(result.text) == 'bread and butter'
```

# which passes

```
$ pytest client1_tests.py
============================= test session starts ==============================
platform darwin -- Python 3.8.6, pytest-6.2.1, py-1.10.0, pluggy-0.13.1
rootdir: /Users/tibs/Dropbox/talks/pact-talk/examples/client1
collected 1 item

client1_tests.py .                                                       [100%]

============================== 1 passed in 0.10s ===============================
```

(provided I remember to run the server process!)

# Just the one test

Since this is the only request from our server to our client, we only need that one test

We assume the server's tests check for incorrect requests - that's not our responsibility

And if we test this request once, the rest of our tests can assume the result

# But - we're making a real request

Which we already said was a Bad Thing at the start of this talk

So let's look at how we can use Pact to describe our request and the response

# Let's write a test with pact - 1/2

Using pact-python

```
#!/usr/bin/env python3

import atexit
import requests

from pact import Consumer, Provider

pact = Consumer('sandwich-maker').has_pact_with(Provider('Butterer'))
pact.start_service()
atexit.register(pact.stop_service)

PACT_BASE_URL = 'http://localhost:1234'
```

# Let's write a test with pact - 2/2

```
BREAD_AND_BUTTER = 'bread and butter'

def test_buttering():

    (pact
    .given('We want to butter bread')
    .upon_receiving('a request to butter bread')
    .with_request('get', '/butter/bread')
    .will_respond_with(200, body=BREAD_AND_BUTTER))

    with pact:
        result = requests.get(f'{PACT_BASE_URL}/butter/bread')

    assert result.text == 'bread and butter'
```

# and it passes

```
$ pytest client1_contract_tests.py
============================= test session starts ==============================
platform darwin -- Python 3.8.6, pytest-6.2.1, py-1.10.0, pluggy-0.13.1
rootdir: /Users/tibs/Dropbox/talks/pact-talk/examples/client1
collected 1 item

client1_contract_tests.py .                                              [100%]

============================== 1 passed in 0.75s ===============================
```

# New files

Running the test creates two files:

- A log file: `pact-mock-service.log`
- A contract file: `sandwich-maker-butterer.json`

# Contract - 1/3

```
{
  "consumer": {
    "name": "sandwich-maker"
  },
  "provider": {
    "name": "Butterer"
  },
```

# Contract - 2/3

```
"interactions": [
  {
    "description": "a request to butter bread",
    "providerState": "We want to butter bread",
    "request": {
      "method": "get",
      "path": "/butter/bread"
    },
    "response": {
      "status": 200,
      "headers": {
      },
      "body": "bread and butter"
    }
  }
],
```

# Contract - 3/3

```
  "metadata": {
    "pactSpecification": {
      "version": "2.0.0"
    }
  }
}
```

# Testing the contract against the server

With the server running (at `http://localhost:8080`):

```
$ pact-verifier --provider-base-url=http://localhost:8080 \
  --pact-url=../client1/sandwich-maker-butterer.json
INFO: Reading pact at ../client1/sandwich-maker-butterer.json

Verifying a pact between sandwich-maker and Butterer
  Given We want to butter bread
    a request to butter bread
      with GET /butter/bread
        returns a response which
WARN: Skipping set up for provider state 'We want to butter bread' ...
          has status code 200
          has a matching body

1 interaction, 0 failures
```

# Interlude

# But buttering should be idempotent

If we ask to butter the same piece of bread more than once,

we still want to get back "bread and butter".

---

Let's update our code to give `server2` and `client2`

# Idempotent buttering

```
@app.route('/butter/<substrate>')
def butter(substrate):
    if substrate.endswith('butter'):
        return substrate
    else:
        return f'{substrate} and butter'
```

# A new server test

```
def test_already_buttered():
    assert butter('bread and butter') == 'bread and butter'
```

# Our server tests still pass

```
$ pytest server2_tests.py
============================= test session starts ==============================
platform darwin -- Python 3.8.6, pytest-6.2.1, py-1.10.0, pluggy-0.13.1
rootdir: /Users/tibs/Dropbox/talks/pact-talk/examples/server2
collected 2 items

server2_tests.py ..                                                      [100%]

============================== 2 passed in 0.04s ===============================
```

# We still honour the contract with client1

```
$ pact-verifier \
  --provider-base-url=http://localhost:8080 \
  --pact-url=../client1/sandwich-maker-butterer.json
INFO: Reading pact at ../client1/sandwich-maker-butterer.json

Verifying a pact between sandwich-maker and Butterer
  Given We want to butter bread
    a request to butter bread
      with GET /butter/bread
        returns a response which
WARN: Skipping set up for provider state 'We want to butter bread' ...
          has status code 200
          has a matching body

1 interaction, 0 failures
```

# client2 wants to use the new ability

An appropriate test against the server would be:

```
def test_buttering_twice():
    result = requests.get(f'{BASE_URL}/butter/bread%20and%20butter')
    assert(result.status_code) == 200
    assert(result.text) == 'bread and butter'
```

# A new contract test

```
def test_buttering_twice():

    (pact
    .given('We want to butter bread again')
    .upon_receiving('a request to butter buttered bread')
    .with_request('get', '/butter/bread%20and%20butter')
    .will_respond_with(200, body=BREAD_AND_BUTTER))

    with pact:
        result = requests.get(f'{PACT_BASE_URL}/butter/bread%20and%20butter')

    assert result.text == 'bread and butter'
```

# which passes

```
pytest client2_contract_tests.py
============================= test session starts ==============================
platform darwin -- Python 3.8.6, pytest-6.2.1, py-1.10.0, pluggy-0.13.1
rootdir: /Users/tibs/Dropbox/talks/pact-talk/examples/client2
collected 2 items

client2_contract_tests.py ..                                             [100%]

============================== 2 passed in 0.79s ===============================
```

# And here is the new interaction

In `client2/sandwich-maker-butterer.json`

```
{
  "description": "a request to butter buttered bread",
  "providerState": "We want to butter bread again",
  "request": {
    "method": "get",
    "path": "/butter/bread%20and%20butter"
  },
  "response": {
    "status": 200,
    "headers": {
    },
    "body": "bread and butter"
  }
}
```

# server2 is happy - 1/2

While running server2 at `http://localhost:8080`

```
$ pact-verifier \
  --provider-base-url=http://localhost:8080 \
  --pact-url=../client2/sandwich-maker-butterer.json
INFO: Reading pact at ../client2/sandwich-maker-butterer.json

Verifying a pact between sandwich-maker and Butterer
  Given We want to butter bread
    a request to butter bread
      with GET /butter/bread
        returns a response which
WARN: Skipping set up for provider state 'We want to butter bread' ...
          has status code 200
          has a matching body
```

# server2 is happy - 2/2

```
  Given We want to butter bread again
    a request to butter buttered bread
      with GET /butter/bread%20and%20butter
        returns a response which
WARN: Skipping set up for provider state 'We want to butter bread again' ...
          has status code 200
          has a matching body

2 interactions, 0 failures
```

# But the old server and the new contract...

```
$ pact-verifier \
  --provider-base-url=http://localhost:8080 \
  --pact-url=../client2/sandwich-maker-butterer.json
INFO: Reading pact at ../client2/sandwich-maker-butterer.json

Verifying a pact between sandwich-maker and Butterer
  Given We want to butter bread
    a request to butter bread
      with GET /butter/bread
        returns a response which
WARN: Skipping set up for provider state 'We want to butter bread' ...
          has status code 200
          has a matching body
```

# fails - 2/4

```
  Given We want to butter bread again
    a request to butter buttered bread
      with GET /butter/bread%20and%20butter
        returns a response which
WARN: Skipping set up for provider state 'We want to butter bread again' ...
          has status code 200
          has a matching body (FAILED - 1)

Failures:
```

# with details - 3/4

```
1) Verifying a pact between sandwich-maker and Butterer Given We want to butter bread
   again a request to butter buttered bread with GET /butter/bread%20and%20butter
   returns a response which has a matching body
  Failure/Error: expect(response_body).to match_term expected_response_body, ...

    Actual: bread and butter and butter

    Diff
    ---------------------------------------
    Key: - is expected
         + is actual
    Matching keys and values are not shown

    -bread and butter
    +bread and butter and butter
```

# and summary - 4/4

```
      Description of differences
      ------------------------------------
      * Expected "bread and butter" but got "bread and butter and butter" at $

2 interactions, 1 failure

Failed interactions:

PACT_DESCRIPTION='a request to butter buttered bread' PACT_PROVIDER_STATE='We want to
butter bread again' /Users/tibs/Library/Caches/pypoetry/virtualenvs/pact-talk-zwt4AdHO-py
--pact-url=../client2/sandwich-maker-butterer.json --provider-base-url=http://localhost:8
# A request to butter buttered bread given We want to butter bread again
```

# Which is good!

`server1` does not support the contract required by `client2`

# Interlude

# What if it's not that simple

What if we have response data that may change?

---

Let's update our code to give `server3` and `client3`

# Butter information

Let's provide information about the butter being used.

```
@app.route('/info')
def info():
    return {
            'salt': random.choice(['0%', '0.9%']),
            'lactose': random.choice([True, False]),
        }
    )
```

# A new server test

```
def test_info():
    result = info()
    assert result['salt'] in ('0%', '0.9%')
    assert result['lactose'] in (True, False)
```

# Which passes

```
$ pytest server3_tests.py
============================= test session starts ==============================
platform darwin -- Python 3.8.6, pytest-6.2.1, py-1.10.0, pluggy-0.13.1
rootdir: /Users/tibs/Dropbox/talks/pact-talk/examples/server3
collected 3 items

server3_tests.py ...                                                     [100%]

============================== 3 passed in 0.05s ===============================
```

# And in our client

```
def test_info():
    result = requests.get(f'{BASE_URL}/info')
    json_result = result.json()
    assert json_result['lactose'] in (True, False)
    salt = json_result['salt']
    assert salt[-1] == '%'
    assert float(salt[:-1]) >= 0.0
```

# Which passes

With server3 running at `http://localhost:8080`

```
$ pytest client3_tests.py
============================= test session starts ==============================
platform darwin -- Python 3.8.6, pytest-6.2.1, py-1.10.0, pluggy-0.13.1
rootdir: /Users/tibs/Dropbox/talks/pact-talk/examples/client3
collected 3 items

client3_tests.py ...                                                     [100%]

============================== 3 passed in 0.10s ===============================
```

# But we want a contract test

```
from pact import Like, Term

BUTTER_INFO = Like(
    {
        'salt': Term(r'\d+(\.\d+)?%', '0%'),
        'lactose': False,
    }
)
```

# And the test

```
def test_info():

    (pact
    .given('We want to know about the butter being used')
    .upon_receiving('a request for information')
    .with_request('get', '/info')
    .will_respond_with(200, body=BUTTER_INFO))

    with pact:
        result = requests.get(f'{PACT_BASE_URL}/info')

    json_result = result.json()
    assert json_result['lactose'] in (True, False)
    salt = json_result['salt']
    assert salt[-1] == '%'
    assert float(salt[:-1]) >= 0.0
```

# And here is the new interaction

In `client3/sandwich-maker-butterer.json`

```
{
  "description": "a request for information",
  "providerState": "We want to know about the butter being used",
  "request": {
    "method": "get",
    "path": "/info"
  },
  "response": {
    "status": 200,
    "headers": {
    },
    "body": {
      "salt": "0%",
      "lactose": false
    },
```

```
    "matchingRules": {
      "$.body": {
        "match": "type"
      },
      "$.body.salt": {
        "match": "regex",
        "regex": "\\d+(\\.\\d+)?%"
      }
    }
  }
}
```

# server3 is happy

with server3 running on `http://localhost:8080`

```
$ pact-verifier \
  --provider-base-url=http://localhost:8080 \
  --pact-url=../client3/sandwich-maker-butterer.json
INFO: Reading pact at ../client3/sandwich-maker-butterer.json

Verifying a pact between sandwich-maker and Butterer
  Given We want to butter bread
    a request to butter bread
      with GET /butter/bread
        returns a response which
WARN: Skipping set up for provider state 'We want to butter bread' ...
          has status code 200
          has a matching body
  Given We want to butter bread again
    a request to butter buttered bread
```

```
      with GET /butter/bread%20and%20butter
        returns a response which
WARN: Skipping set up for provider state 'We want to butter bread again' ...
          has status code 200
          has a matching body
  Given We want to know about the butter being used
    a request for information
      with GET /info
        returns a response which
WARN: Skipping set up for provider state 'We want to know about ...
          has status code 200
          has a matching body

3 interactions, 0 failures
```

# Interlude

# How to share the contract

- By copying the contract file - don't do this!
- By "reference" - e.g., via github
- Using a Pact broker - at https://pactflow.io/
- Using a Pact broker - run "locally" as described at https://github.com/pact-foundation/pact_broker

# Other benefits

- The server can tell what requests it needs to support, making dead code detection easier (assumes complete coverage!)
- Programmers can look at the contracts to learn about how requests and responses are structured
- Programmers can look at the contracts when trying to debug communication issues

and doubtless other things

# Multiple programming languages

Pact has a very active user community, and support for a variety of programming languages:

> .NET (for C#), Go, JavaScript, Python, Ruby, Rust, the JVM (for Java, Scala, Clojure, etc.),

with more in development. And if it is not directly supported for a language, there are ways around that.

*That means client and server need not be in the same language*

# Fin

- Pact: https://docs.pact.io/

*Remember, buttering should be idempotent.*

Written in reStructuredText, converted to PDF using rst2pdf

Source and examples at https://github.com/tibs/pact-talk

This slideshow and its related files are released under a Creative Commons Attribution-ShareAlike 4.0 International License.