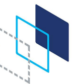
Advanced Pacting

Beth Skurrie
@bethesque

Topics

- Writing consumer tests
- Verifying Pacts
- Using tags
- CI/CD
- Questions



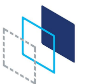
Writing consumer tests

Writing good consumer tests

- Understand the difference between contract tests and functional tests
- Getting the scope right

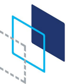
Functional vs contract tests

- Contract tests focus only on the *messages* (request and response)
  - When I send
    - POST /widgets
    - Request body containing widget properties
  - I receive
    - 200 OK
    - Location header with URL of new widget
    - Response body containing widget properties
- Functional tests also check for side effects
  - All of the above checks
  - Plus: is the widget stored correctly in the repository?



POP QUIZ

- Could you “hardcode” a provider implementation that passes the contract tests, but actually doe



POP QUIZ

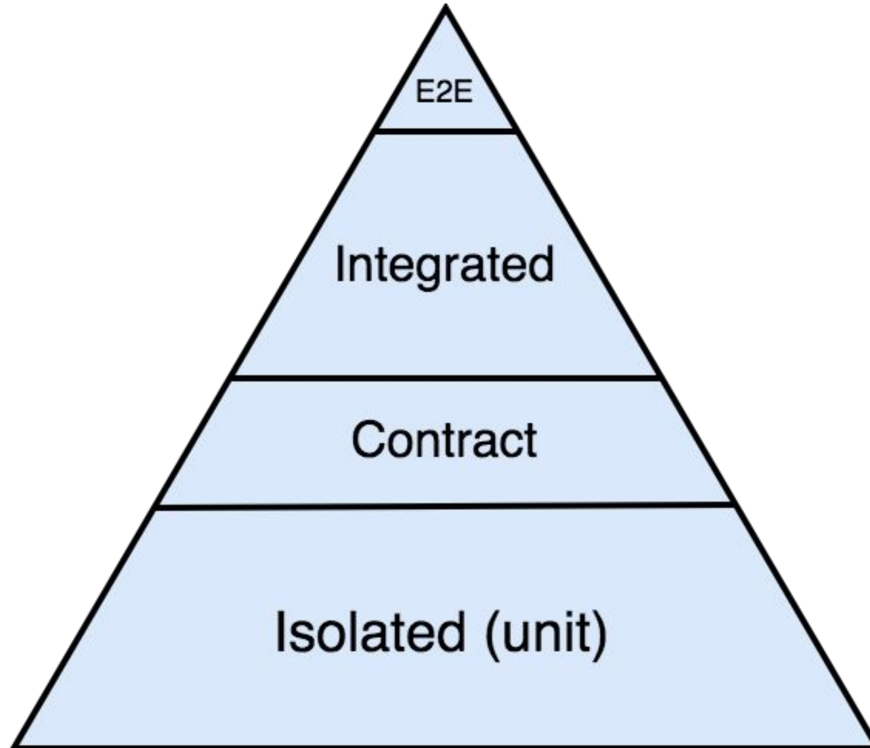
- Could you “hardcode” a provider implementation that passes the contract tests, but actually doesn’t persist any data?

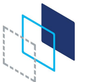
  YES!!!!

- What stops us doing this?

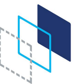
  The functional tests in the provider’s own codebase

Contract tests aren’t designed to operate alone



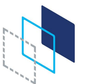


It’s not the job of the consumer to be a test harness for the provider



Is there duplication between functional and contract tests?

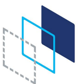
- Yes, but not entirely
- A contract test only covers the attributes of the request and response that a particular consumer cares about
- By design it excludes things that the consumer does not care about
- Functional test covers all the functionality available to all the consumers



Bad example - functional test

When "creating a user with a username with 21 characters"
    POST /users { "username": "thisisalooongusername" }
Then
    Expected Response is 400 Bad Request
    Expected Response body is { "error": "username cannot be more than 20 characters" }

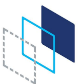
- **Focusses on the implementation**
- **Brittle**
- **Tempts you to try and write a test for every scenario**



Good example - contract test

When "creating a user with invalid data"
POST /users { "username": "thisisalooongusername" }
Then
Expected Response is 400 Bad Request
Expected Response body is { "error": Pact.like(“some error message”) }

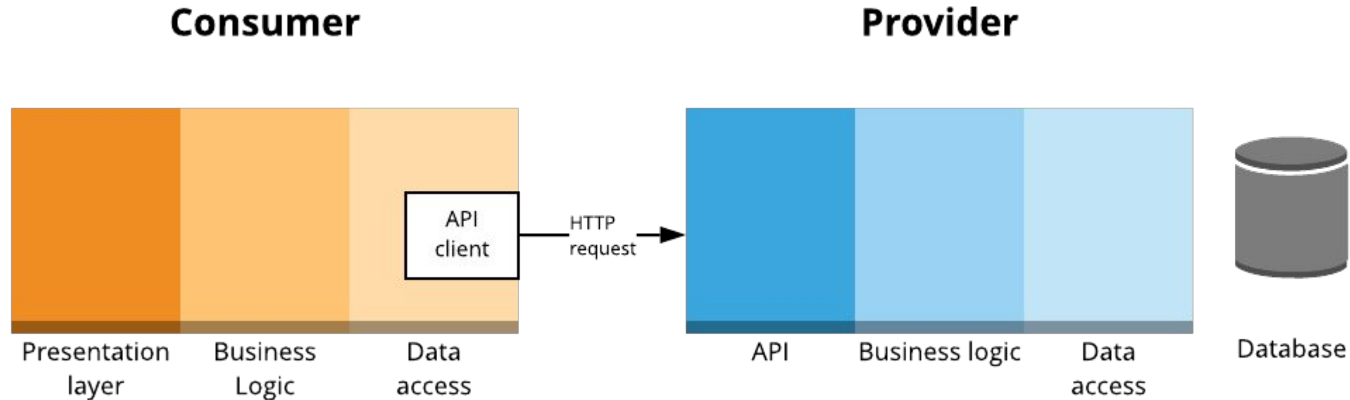
- **Focuses on the shape of the document**
- **Flexible**
- **Maintainable**



A good contract test aims to expose:

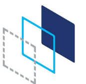
- bugs in the consumer code
- misunderstanding from the consumer about end-points or payload
- breaking changes by the provider on end-points or payload

Scope, scope, scope



Maintainable scope for pact consumer tests

Unmaintainable scope for pact consumer tests

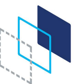


Why not include the UI/business logic layers?

- Maintainability
  - Using Pact to test UI concerns causes interactions with minor variations to be added to the contact that don’t meaningfully increase test coverage, but do increase the maintenance of the provider verifications.

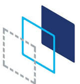
Provider API Client Responsibilities

- Converts back and forth between the business domain classes and concepts of the consumer and the HTTP requests and responses required to communicate with the provider
- Abstracts the HTTP-ishness of the provider
- Eg. 200 returns an object, 404 returns null, 401 raises a validation error



Options for “top to bottom” consumer tests

- Use the pact generated by the unit tests along with a pact *stub* server.
- Use a separate HTTP mock library and use shared fixtures between both the pact tests and the “top to bottom” tests.
- Use a separate HTTP mock library and parse the generated pact to initialise the mock.



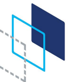
Other tips

- You should be able to construct a proper sentence using the description and the provider state(s).
  - Given <u>an alligator named Mary exists</u> upon receiving <u>a request to retrieve an alligator by name</u> the provider will respond with …
- Think of the readers of the generated documentation and try to use BDD style notation to describe the business actions rather than describing the HTTP mechanisms where possible.
  - “a request to activate a user” rather than “a request to set active to true”

Other tips

- Only use deterministic data - more on this later
- Reuse provider states where it makes sense, to ease the maintenance burden on the provider team.
- Make your response expectations as loose as possible
    - eg. `{“bar”: Pact.like(“foo”) }` rather than `{“bar”: “foo”}`

Writing consumer tests - Question time



Verifying pacts

Verifying pacts

- Where to stub
- Handling authentication and authorization

Scope of a provider verification test

Consumer

Provider

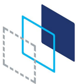
HTTP request

Presentation layer | Business Logic | Data access

API | Business logic | Data access | Database

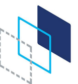
Safe scopes for pact provider verifications

Dangerous scope for pact provider verifications

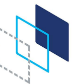


Stubbing

- Should be able to run on your local development machine
- Always stub external services
- Stub whichever layer(s) of your provider makes sense for you
  - Balance
    - Speed of feedback
    - Accuracy of feedback
    - Maintainability of tests
  - Microservice with SQLite database? Might not need to stub anything
  - Heavyweight proprietary database? Maybe stub DAO.
- Beware of stubbing business logic though
  - business logic can affect the response given, and hence, make the verification results unreliable



Be aware!!!



When you stub

- Be aware of the tradeoffs
    - Stubbing improves reliability, but reduces confidence
- Make sure you have a matching “contract” test with the thing you’re stubbing to make sure you’re stubbing it right.

Handling authentication and authorisation

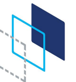
- Should authentication and authorization be part of the contract?
- No straight answer, it’s about the tradeoffs
- Yes
  - Increase in certainty
  - Less to cover in any integrated tests
  - Good if the auth code is custom and likely to change
- No
  - Simpler
  - If using stable standards, there may be little benefit
  - May be more easily covered in e2e tests

Options

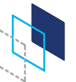
- Ignore auth (test using other types of tests)
- Stub your auth services (client code or implementation)
- Use provider states to create real users with matching credentials
- Modify the request before sending it using live credentials (using Pact framework)
  - Make sure the credentials you're replacing "match", otherwise, there's no point in including them in the contract
- Use your own custom middleware or proxy to modify the response with live credentials
- Use a 100 year token!

Required

Communicate
and
collaborate



Verifying Pacts - Question time



Pactflow/Pact Broker

PACTFLOW

Start filtering your pacts

WHAT'S NEW

Use old UI

OVERVIEW | NETWORK DIAGRAM | MATRIX

Status | Integration

Order Web ∞ Order API

Example App ∞ Example API

Example App ∞ Example API

Successfully verified

CONSUMER VERSION
7bd4d9173522826dc3e8704fd62d-de0424f4c827
Published: 10 months ago
dev

PROVIDER VERSION
4fd-f20082263d4c5038355a3b734be1c0054d1e1
Verified: 10 months ago
dev

VIEW PACT

Verification failed

CONSUMER VERSION
e15da45d3943bf10793a6d04cfb9f5d-abe430fe2
Published: 10 months ago
prod

PROVIDER VERSION
480e5aeb30467856-ca995d0024d2c1800b0719e5
Verified: 10 months ago

VIEW PACT

A pact between Matching Service and Animal Profile Service

Matching Service (consumer) version: **1.0.1-b48bc02288f6c1e912cae579105e43d9** prod test

Animal Profile Service (provider) version: **1.0.0**

Pact publication date: **7 months ago**

Verification failed: **6 months ago**

Pact specification version: **2.0.0**

- ⊗ A request for all animals given has some animals
- ⊗ A request for an animal with ID 1 given has an animal with ID 1
- ✓ A request for all animals given is not authenticated
- ✓ A request for an animal with ID 100 given has no animals
- ✓ A request to create a new mate

A request for orders given there are orders

Request

Method:GET
Path:/orders

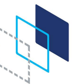
Expected response

Status:200
Headers:


```
{
  "Content-Type": "application/json; charset=utf-8"
}
```


Body:


```
[
  {
    "id": 1,
    "items": [
      {
        "name": "burger",
        "quantity": 2,
        "value": 100
      }
    ]
  }
]
```

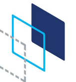


A request for an animal with ID 1 given has an animal with ID 1

Mismatches (1)

Body:

Could not find key "first_name" (keys present are: last_name, animal, age, available_from, gender, location, eligibility, interests, id) at $



Using tags

Pact Broker BDM - Pact publication

Consumer

1

0..*

Consumer Version

1

0..*

Tag

0..*

1

Pact Publication

0..*

1..*

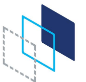
1

Pact Version

This is the actual pact content

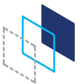
1

Provider



Pact Broker BDM - with verification results

Consumer
1
0..*
Provider
1
1
0..*
0..*
Consumer Version
1
0..*
Pact Publication
1..*
1
Pact Version
1
0..*
Verification Results
0..*
1
Provider Version
1
0..*
1
0..*
Tag
This is the actual pact content
Tag



Pact Broker Class diagram

Pact Publication

1..*

1

Pact Version

This is the actual pact content

0..*

0..*

0..*

has a **Consumer**

has a **Provider**

has a **Consumer Version**

1

1

1

"Consumer" and "Provider" are roles that a pacticipant takes in a pact

Pacticipant

1

0..*

Pacticipant Version

1

0..*

Tag

1

1

has a **Provider**

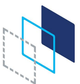
has a **Consumer**

0..*

0..*

Verification results belong to the pact **content**, and the content may belong to many pact publications

Verification results



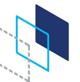
What are tags?

- Simple string values
- Belong to pacticipant (application) version resources in the Pact Broker
- Tell us metadata about the pacticipant version
  - Git branch eg. “master”, “feat/xyz”
  - Deployment stage eg. “test”, “prod”
- To create:

```
PUT /pacticipants/PACTICIPANT/versions/VERSION/tags/TAG
```

When are tags created?

- During pact publication - this tells us which branch the pacticipant version (and hence, the pact) was published from
- During verification results publication - this tells us which branch the pacticipant version(and hence, the verification results) were published from
- After deployment - this tells us which environment the application version is deployed to



Tagging over time

Time

test prod master

version 0cdaa54

pacts

The latest "prod" pacts

master

version 58183a2

pacts

feat/new-thing

version 78dde00

pacts

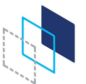
The latest "feat/new-thing" pacts

test master

version 0d99a49

pacts

The latest "master" and "test" pacts

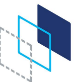


Things to note

- “latest” is not a tag itself (unlike docker tags). It is a dynamically calculated reference to the actual latest resource.
- Ordering for calculating the “latest” is done by the creation date of the pacticipant versions, not the tags.
    - If you rollback to a previous prod version, delete the tag on the undeployed version
- You can think of the the time ordered series of pacticipant versions that belong to a particular tag as forming a “pseudo branch”.

What are tags used for?

- Identifying the resources in the Pact Broker that belong to a given branch or stage.
- Examples
  - The latest production version of Foo
    `/pacticipants/Foo/latest-version/`**`prod`**
  - The pact belonging to the latest production version of Foo, and Bar
    `/pacts/provider/Foo/consumer/Bar/latest/`**`prod`**
  - All production versions of a mobile consumer

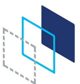


What does this allow us to do?

1. Ensure we are verifying the right pacts
2. Ensure backwards compatibility
3. Provides a mechanism for introducing changes to pacts
4. Easily ensure safe deployments

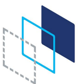
1. Ensure the right pacts are verified

- The example used for default provider verification configurations usually specifies to verify the “overall latest pact”
- What if the latest pact came from a feature branch?
- Tag with the branch name when you publish pacts
- Configure the provider to verify the “latest master” (or whatever the name of your main line of development is).



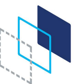
2. Ensure backwards compatibility

- Verifying “latest master” ensures our provider is compatible with the current consumer code.
- Microservices -> decouple release cycles of consumer and provider
- Need to ensure provider is compatible with *production* consumer as well as latest
- Tag with the stage name when you deploy application
- Configure the provider to verify the latest test/prod pacts as well as the latest master.



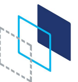
3. Introduce changes without breaking builds

- If following “consumer driven” pacts, pact is changed *before* provider
- This would break provider build
- Do changes on branch of consumer, and tag with branch name OR do changes with feature toggle and tag with toggle name
- Collaborate with provider team!
- Once feature pact is successfully verified, merge to master/turn toggle on



4. Easily ensure safe deployments

- Each pact publication is associated with a consumer version
- Each pact verification is associated with a provider version
- The pact publication is linked to the verification results through the pact (content) version
- There is a **many to many** relationship between consumer version and provider version thought pact publication/pact version/verification results



Quick tangent! Pre-verification

- If pact with same content published multiple times with different consumer versions:
  - New pact publication resource each time
  - Reuses existing pact version
  - Inherits existing verification results
- This is how pacts are “pre-verified”
- This is why it’s best to use deterministic data

The Matrix

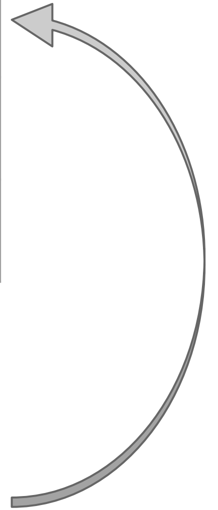
Consumer (Foo) version	**Provider (Bar) version**	**Verification result**
11	54 **prod**	success
12	54	failure
12	55	success

The can-i-deploy CLI

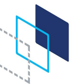
- Queries the matrix to determine if a set of pacticipant versions can be safely deployed together
  - ie. is there a pact with a successful verification result between the specified consumer and provider versions

can-i-deploy

Consumer (Foo) version	Provider (Bar) version	Verification result
11	54 **prod**	success
12	54	failure
12	55	success

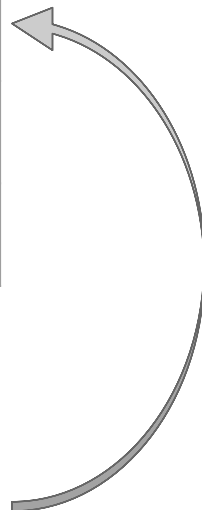


```
$ pact-broker can-i-deploy
--pacticipant Foo --version 11
--pacticipant Bar --version 54
```

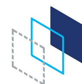


can-i-deploy - better

Consumer (Foo) version	Provider (Bar) version	Verification result
11	54 **prod**	success
12	54	failure
12	55	success

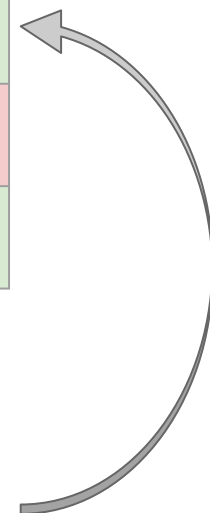


```
$ pact-broker can-i-deploy
--pacticipant Foo --version 11
--pacticipant Bar --latest prod
```

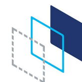


can-i-deploy - best

Consumer (Foo) version	Provider (Bar) version	Verification result
11	54 **prod**	success
12	54	failure
12	55	success

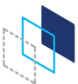


```
$ pact-broker can-i-deploy
--pacticipant Foo --version 11
--to prod
```



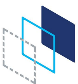
Tagging with feature toggles

- The current Pact Broker workflow best suits branch based development
- Expects one pact per consumer version, but feature toggles mean there might be multiple variations of the pact for the same git sha.
- Conceptually though, these can be thought of as different versions



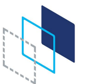
Feature toggle flow

- Consumer
  - Create pact with toggles off
    - Consumer version - sha eg. `effe8a07`
    - Tag with ‘base’
  - Create pact with toggle A on
    - Consumer version sha+toggle_name eg. `effe8a07+feat_a`
    - Tag with toggle name
- Provider
  - Verify pacts with toggles off
    - Provider version - sha eg. `d3092627`
    - Tag with ‘base’
  - Verify pacts with toggle B on
    - Provider version - sha+toggle_name eg. `d3092627+feat_b`



Feature toggle matrix

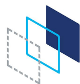
Consumer (Foo) version	Provider (Bar) version	Verification result
effe8a07	d3092627	success
effe8a07	d3092627+feat_b	success
effe8a07+feat_a	d3092627	failure
effe8a07+feat_a	d3092627+feat_b	success



can-i-deploy for deployment

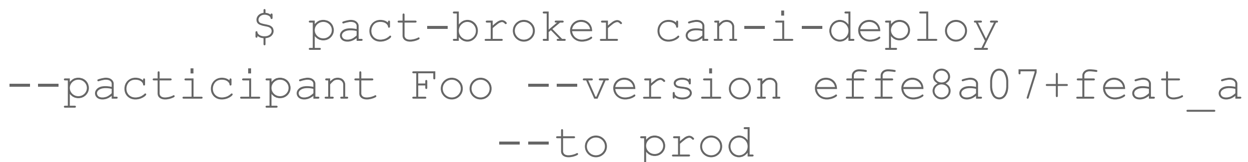
Consumer (Foo) version	Provider (Bar) version	Verification result
effe8a07	d3092627 **prod**	success
effe8a07	d3092627+feat_b	success
effe8a07+feat_a	d3092627 **prod**	failure
effe8a07+feat_a	d3092627+feat_b	success

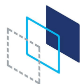
```
$ pact-broker can-i-deploy
--pacticipant Foo --version effe8a07
--to prod
```



can-i-deploy for enabling dynamic toggle

Consumer (Foo) version	Provider (Bar) version	Verification result
effe8a07	d3092627 **prod**	success
effe8a07	d3092627+feat_b	success
effe8a07+feat_a	d3092627 **prod**	failure
effe8a07+feat_a	d3092627+feat_b	success



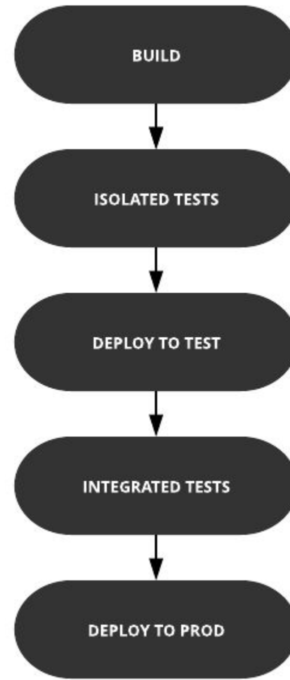


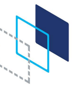
CI/CD

The CI/CD/Pact Broker touchpoints

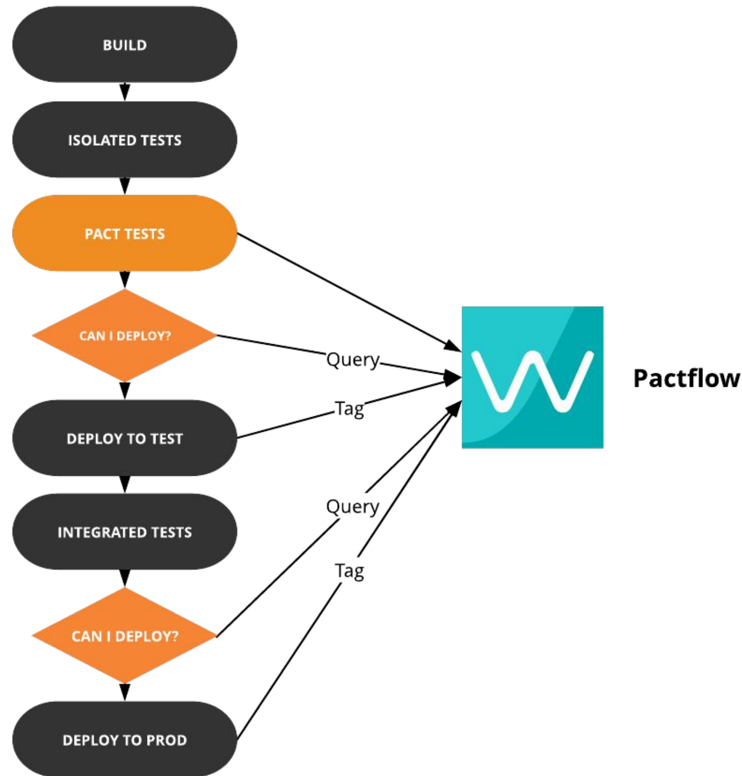
1. Pact changed (CI)
2. Provider changed (CI)
3. Release workflow (CD)

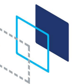
Build pipeline without Pact





Build pipeline with Pact





Build pipeline with Pact - Consumer

BUILD
ISOLATED TESTS
GENERATE PACT
CAN I DEPLOY?
DEPLOY TO TEST
INTEGRATED TESTS
CAN I DEPLOY?
DEPLOY TO PROD

Consumer

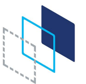
1. Publish pact
Query
Tag
Query
Tag

Pactflow

2. Webhook
3. Publish results

Provider

VERIFY PACT



Build pipeline with Pact - Provider

Provider

BUILD

ISOLATED TESTS

VERIFY PACTS

CAN I DEPLOY?

DEPLOY TO TEST

INTEGRATED TESTS

CAN I DEPLOY?

DEPLOY TO PROD

Publish results

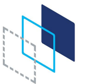
Query

Tag

Query

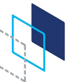
Tag

Pactflow

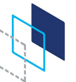


For extra brownie points

- Git statuses
- Slack updates



CI/CD - Question Time



Pending pacts - the problem

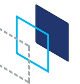
- Changes to the pact can break the provider's build

Pending pacts - the solution

- If the pact content has not yet been successfully verified:
  - It is considered “pending”
  - If verification fails, it will not fail the build
- Once it has been successfully verified:
  - It is no longer “pending”
  - Any failure can only be due to a change in the provider
  - If verification fails, it will fail the build

Pending pacts - something to note

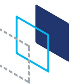
- The pending status is calculated based on the tags that will be applied to the provider version when the results are published



WIP Pacts - the problem

1. Changed pact published with tag ‘feat/foo’
2. ‘Changed contract’ webhook triggers verification - failure
3. Provider implements required changes
4. Provider runs verification for consumer tags ‘master’ and ‘prod’

Unless provider team changes the consumer tags to verify in the configuration, the ‘feat/foo’ pact won’t get a successful verification result.



WIP Pacts - the solution

- Changed pact published with tag ‘feat/foo’
- ‘Changed contract’ webhook triggers verification - failure
- Provider implements required changes
- Provider runs verification for consumer tags ‘master’ and ‘prod’ - and also automatically verifies any “work in progress” pacts.

A “work in progress” pact is one which is the latest for its tag, and has not yet been successfully verified.