Advanced Pacting

Beth Skurrie / Yousaf Nabi
@bethesque / @you54f

PACTFLOW
A SMARTBEAR COMPANY

Topics

- Writing consumer tests
- Verifying Pacts
- Using branches and environments
- CI/CD
- Organisational Scale
- Questions

Writing consumer tests

Writing good consumer tests

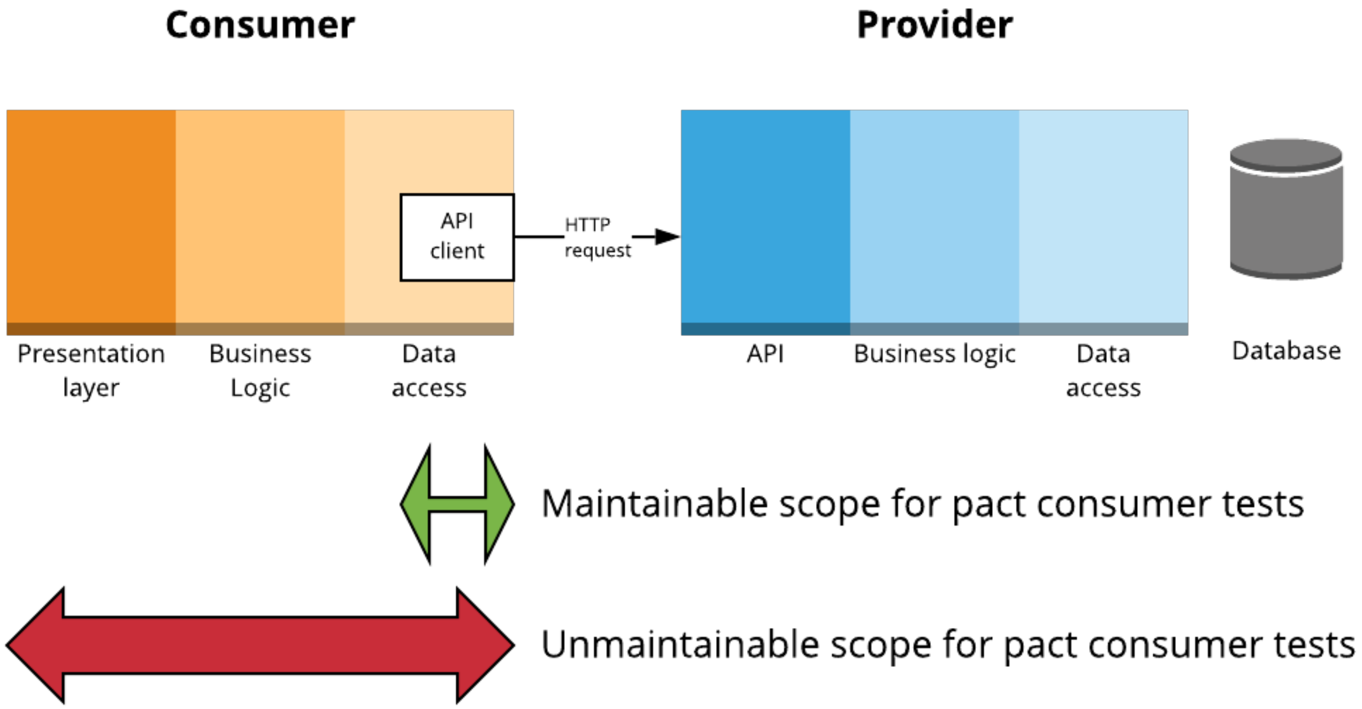
- The key is knowing what *not* to test.
- Restraint!
- It is easy to write Pact tests that are
  - Brittle
  - Overly strict
  - Burdensome

Writing good Pact tests
is the difference between a
useful Pact implementation
and one which makes everyone wish
they'd stuck with integration tests.

What does a good Pact test look like?

- There should be sufficient assertions to ensure that the response from the provider will not break the consumer while also keeping:
  - The scope of the consumer code under test as small as it can be.
  - The expectations on the response as loose as they can be.
  - The focus on the messages (request/response), not the implementation.

Scope, scope, scope



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

SMARTBEAR

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.

SMARTBEAR

Pact Matchers - the main ones (v2)

- Type based matching
- Regular expressions

Type based matching - objects

```
like(“foo”) - match any string
like(1)       - match any number
like(true)  - match any boolean

like({
  “name”: “foo”,
  “address”: {
    “street”: “Flinders St”
    }
  })
```

Type based matching - arrays

```
eachLike("foo") - an array of strings
eachLike(1) - an array of numbers
eachLike(true) - an array of booleans

eachLike({
  "name": "foo",
  "address": {
    "street": "Flinders St"
    }
  })
```

SMARTBEAR

Type based matching - arrays

```
eachLike({
  “name”: “foo”,
  “address”: {
    “street”: “Flinders St”
    }
  }, { “min”: “2} )
```

Things to remember

- Responses can have extra keys without failing the verification, but requests cannot - Postel’s Law.
- You can be as strict as you like on requests - they’re your own tests.
- V3 matchers https://github.com/pact-foundation/pact-specification/tree/version-3
- Pact doesn’t support optional fields - [] or [{foo: “bar”}]
    - Verifications may validate against only [] and pass, but fail in production

?

Questions

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 doesn’t persist any data?

POP QUIZ

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

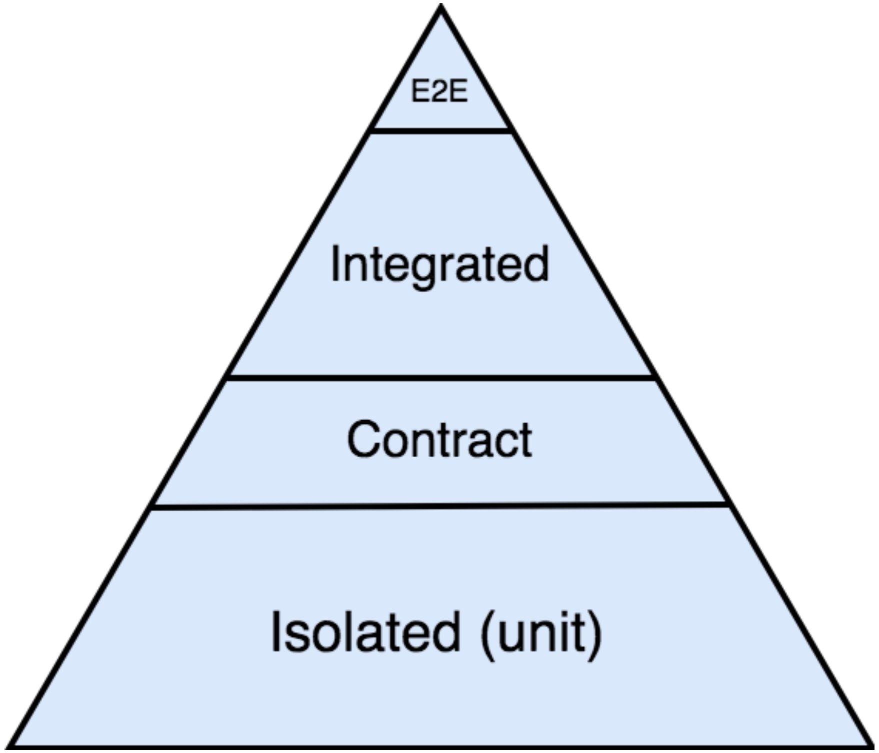
  YES!!!!

- What stops us doing this?

  The functional tests in the provider’s own codebase

SMARTBEAR

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**

Don’t add an assertion for a business rule just because you know it to be true

- “All plans IDs for Victoria should start with 8”
- “The customer ID should be 12 characters long”

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

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

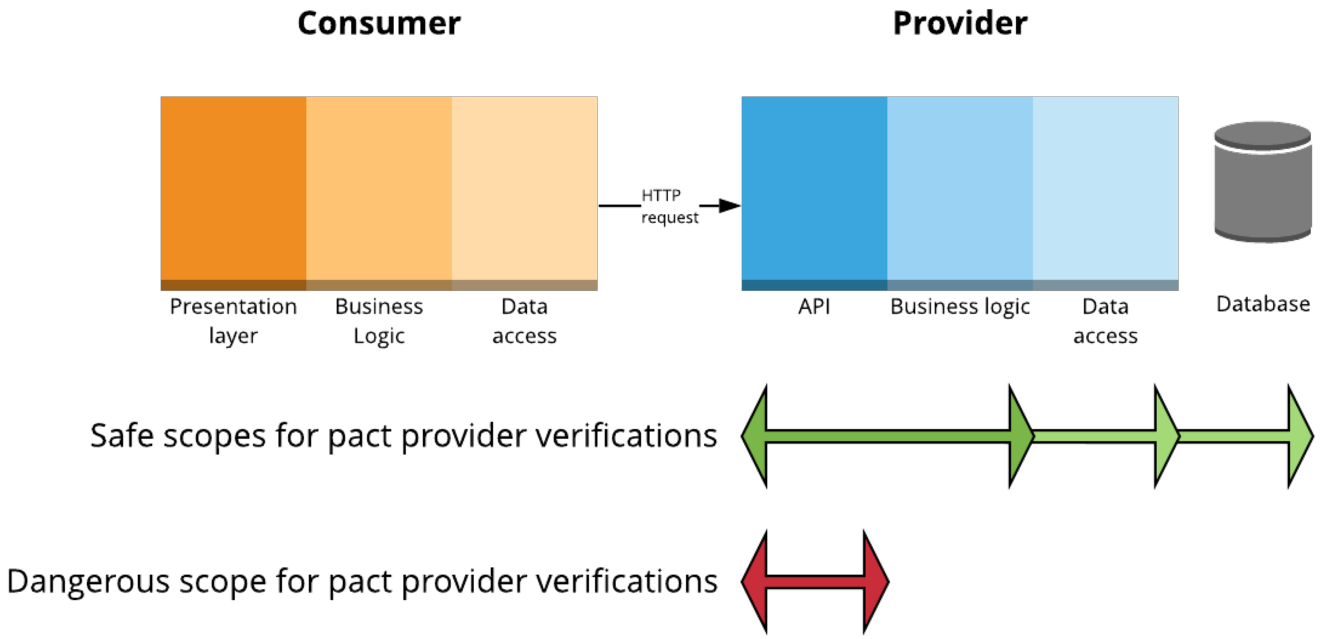
SMARTBEAR

Verifying pacts

Verifying pacts

- Where to stub
- Handling authentication and authorization

Scope of a provider verification test



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 vs 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

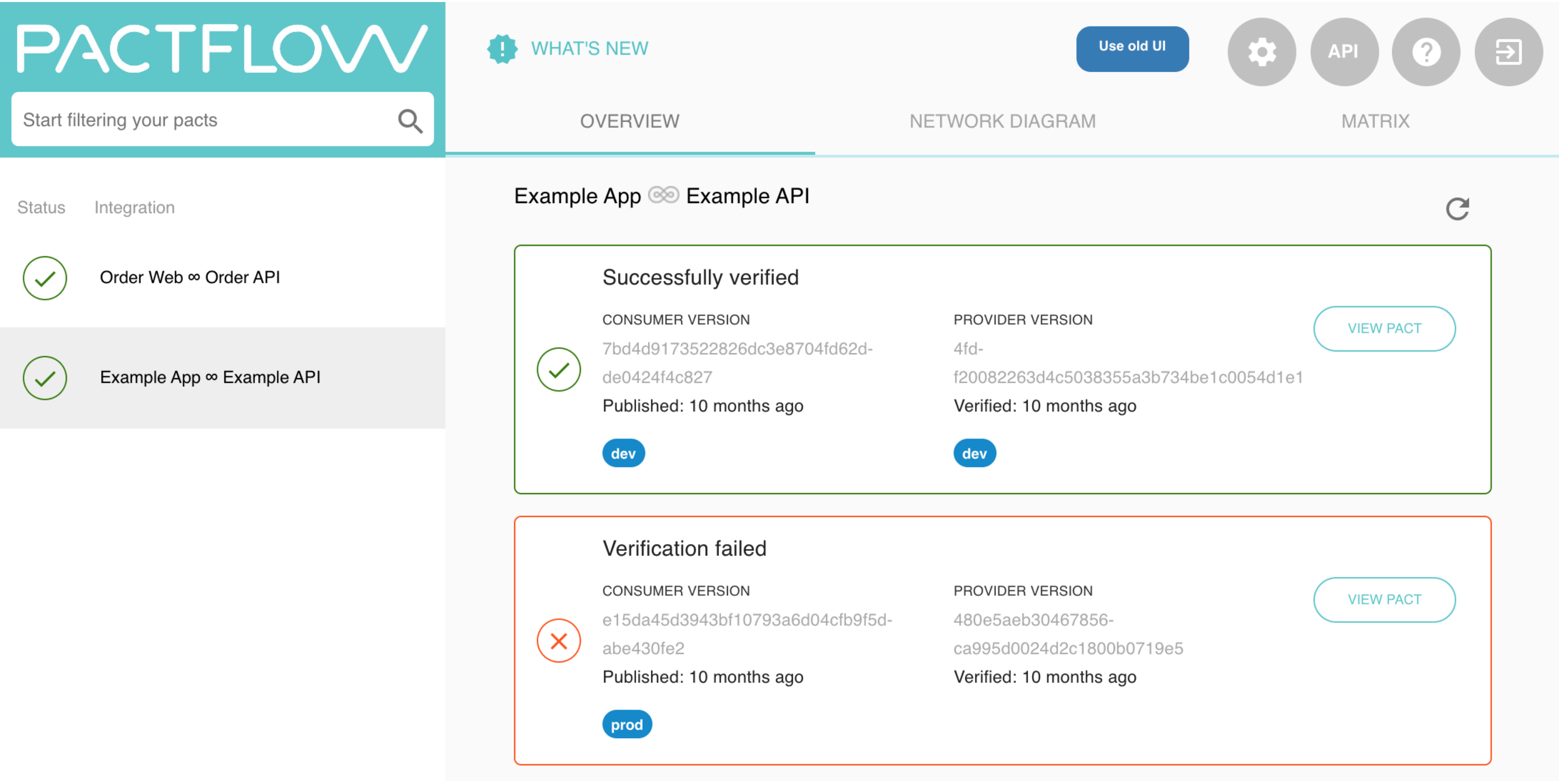
Communicate

and

collaborate

Verifying Pacts - Question time

Pactflow/Pact Broker



Status Integration

✓ Order Web ∞ Order API

✓ Example App ∞ Example API

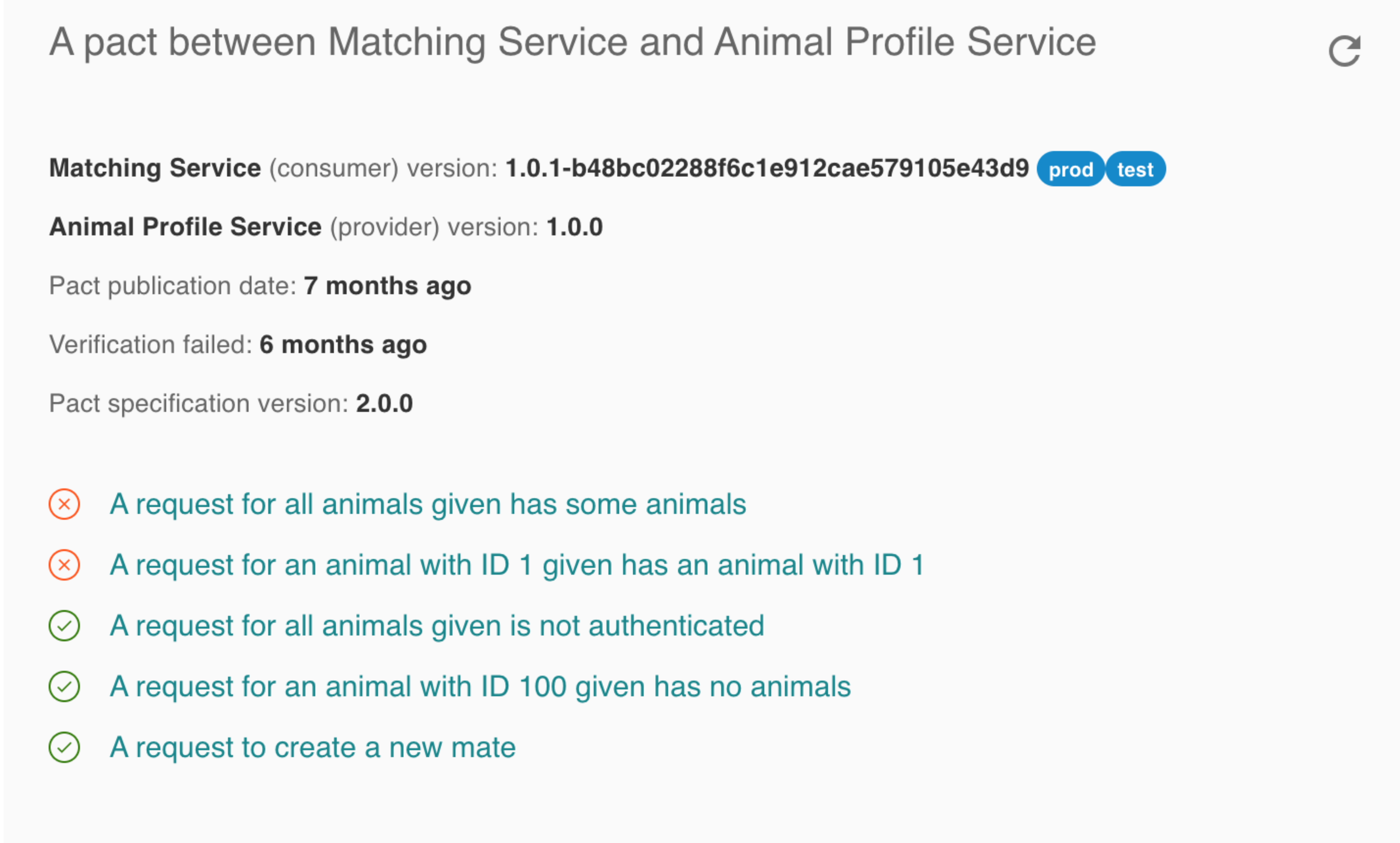
Example App ∞ Example API

Successfully verified

CONSUMER VERSION	PROVIDER VERSION
7bd4d9173522826dc3e8704fd62d-de0424f4c827	4fd-f20082263d4c5038355a3b734be1c0054d1e1
Published: 10 months ago	Verified: 10 months ago
dev	dev

Verification failed

CONSUMER VERSION	PROVIDER VERSION
e15da45d3943bf10793a6d04cfb9f5d-abe430fe2	480e5aeb30467856-ca995d0024d2c1800b0719e5
Published: 10 months ago	Verified: 10 months ago
prod	



A pact between Matching Service and Animal Profile Service

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

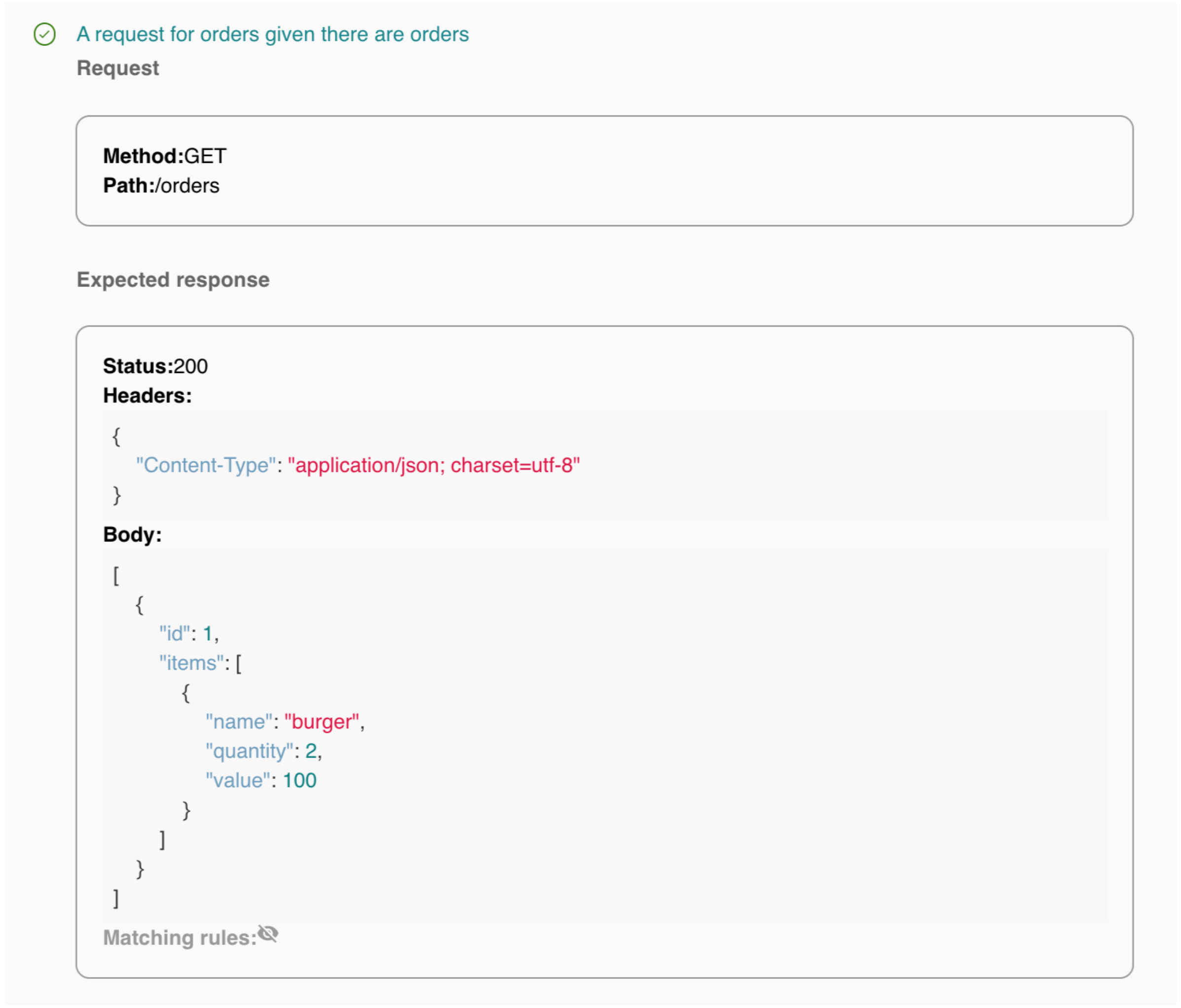
Animal Profile Service (provider) version: **1.0.0**

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

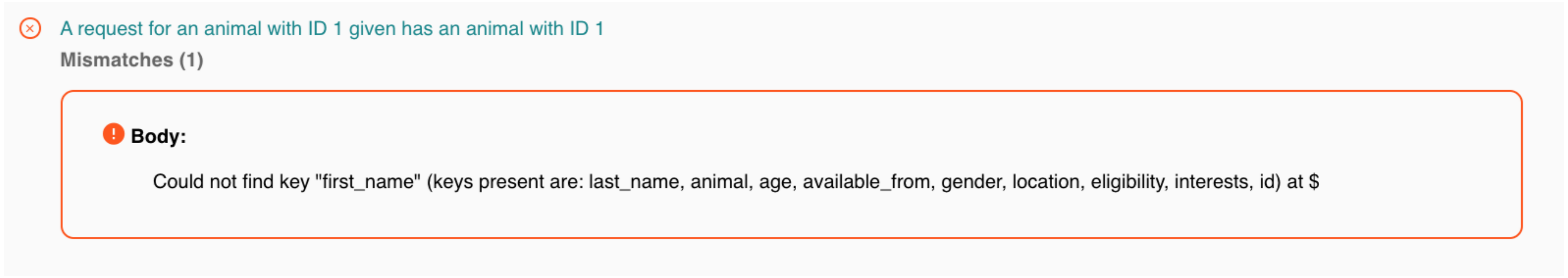
Status:200
Headers:

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

Body:

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

Matching rules:


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

Mismatches (1)

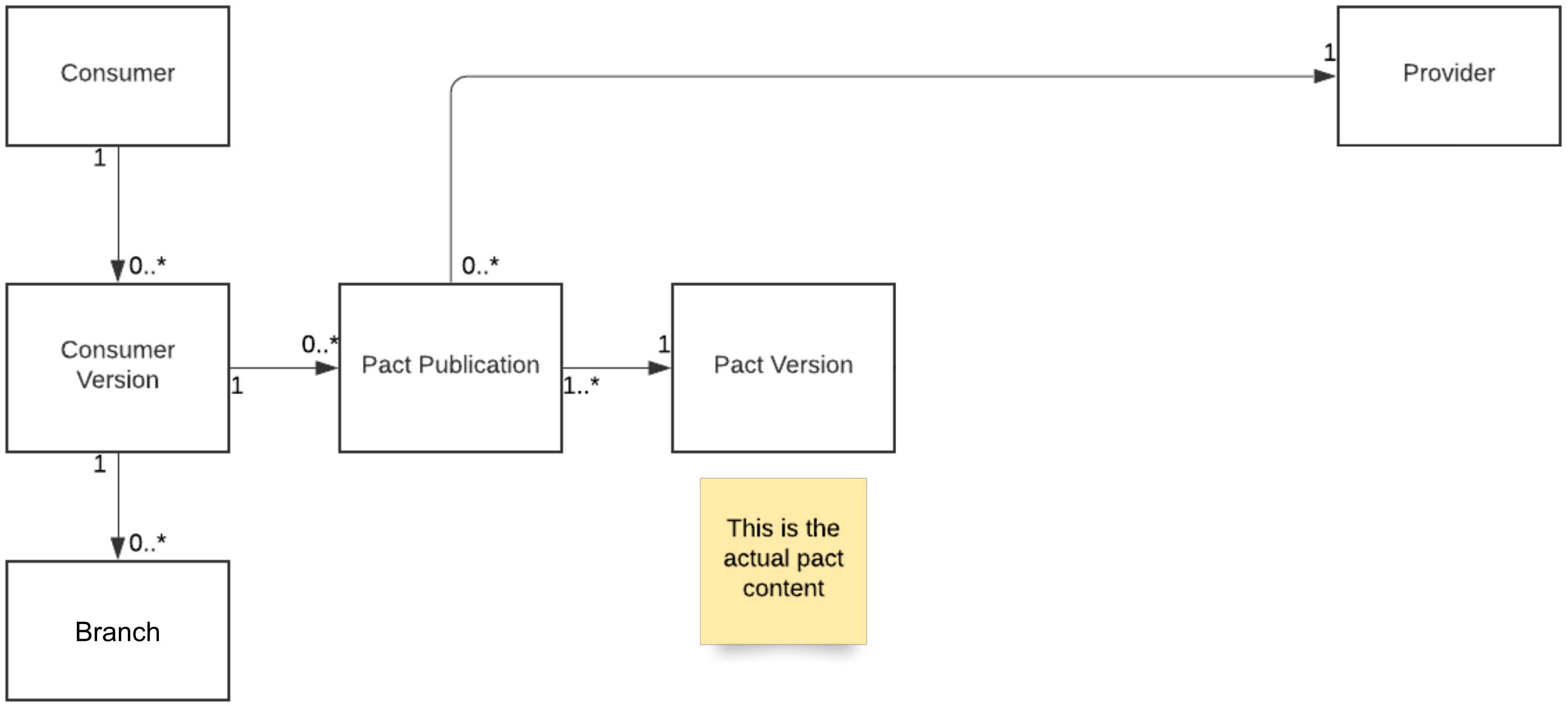
Body:

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

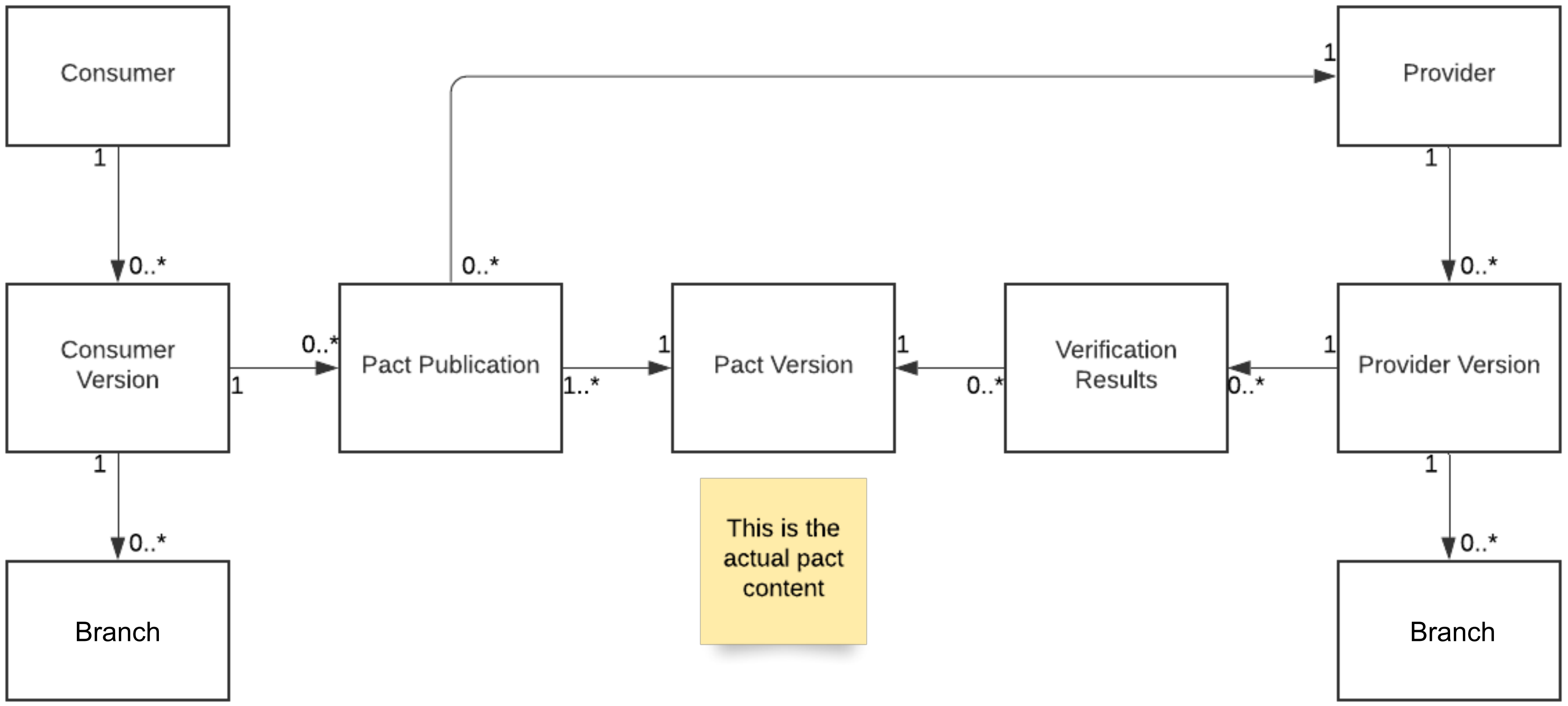
SMARTBEAR

Using branches / deployments

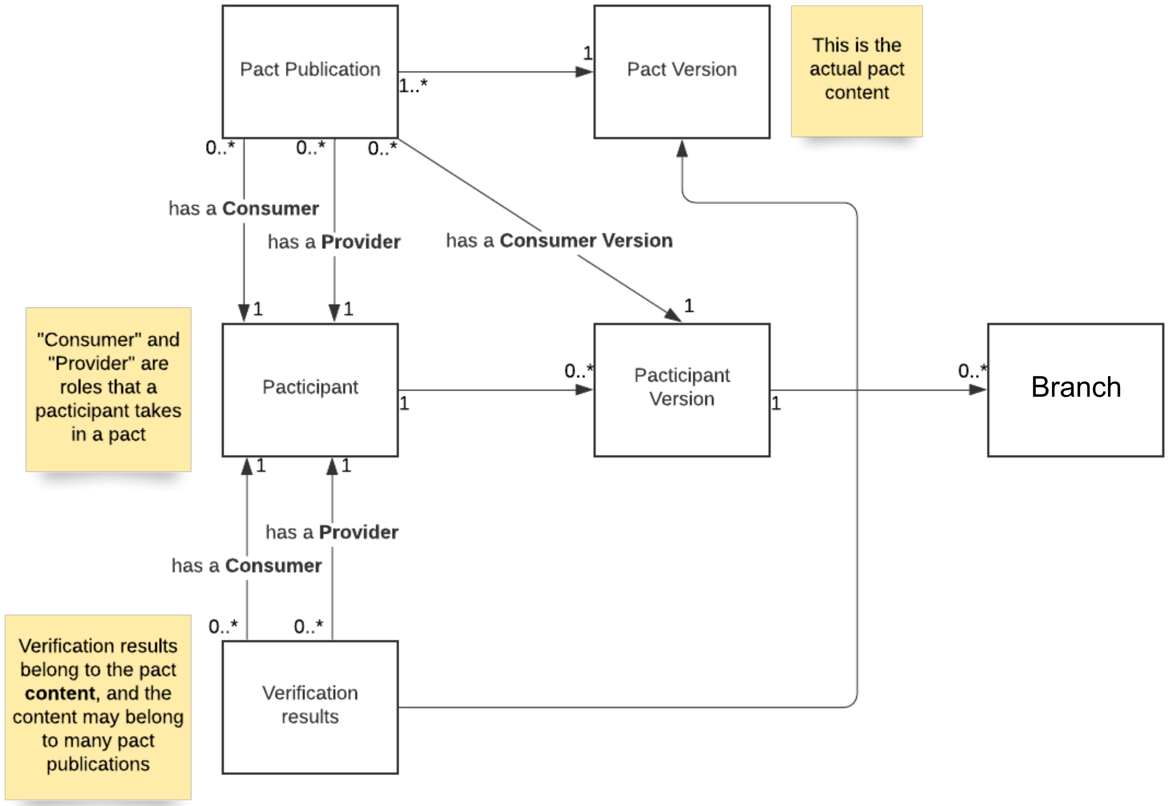
Pact Broker BDM - Pact publication



Pact Broker BDM - with verification results



Pact Broker Class diagram


Pact Publication
Pact Version
1
1..*
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..*
Branch
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 branches?

- Branches in the Pact Broker are designed to model repository (git, svn etc) branches
- Belong to pacticipant (application) version resources in the Pact Broker
- Tell us metadata about the pacticipant version
  - Git branch eg. “master”, “feat/xyz”
- A pacticipant version in the Pact Broker should map 1:1 to a commit in your repository. To facilitate this, the version number used to publish pacts and verification results should either *be* or *contain* the commit.

What are environments?

- For can-i-deploy to work correctly, every team and the Pact Broker must have the same shared understanding of what an "environment" is
- The Pact Broker needs to know which versions of each application are in each environment so it can return the correct pacts for verification and determine whether a particular application version is safe to deploy.
- The environments should mirror your organisations environments in which you will target for deployments
- You can set these up in the Pact Broker/Pactflow

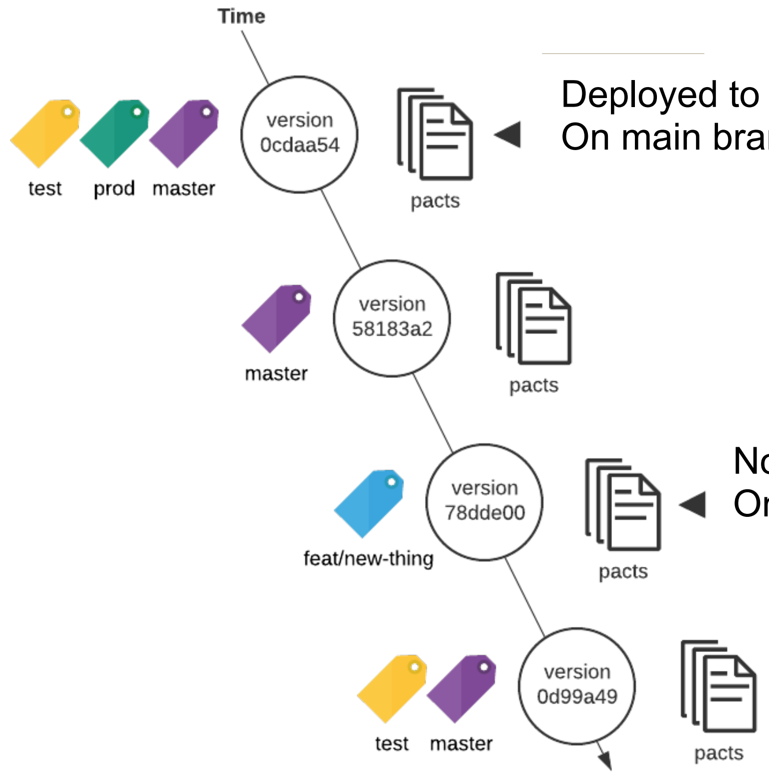
When are branches created?

- 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

When are deployments recorded?

- After deployment - this tells us which environment the application version is deployed to
  - record-deployment automatically marks the previously deployed version as undeployed, and is used for APIs and consumer applications that are deployed to known instances.
  - record-release does NOT change the status of any previously released version, and is used for mobile applications and libraries that are made publicly available via an application store or repository.

Environments and Branches over time



Deployed to test & prod envs.
On main branch

Not deployed.
On feature branch

Deployed to test.
On main branch
Undeploys version 0cdaa54
if used with record-deployment

What are branches used for?

- Branches are used to identify which pacts a provider should verify using consumer version selectors.
- Typically, the provider should be configured to verify the pacts belonging to the main branch of each consumer (amongst others - read more here).
- Branches are also used to calculate the pending status of a pact and identify work in progress pacts.

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?
- Set the branch name when you publish pacts
- Configure the provider version selectors to verify the main branch of the consumer
  - {mainBranch: true}
  - Set the Pacticipant main branch property to "master” (or whatever the name of your main line of development is)

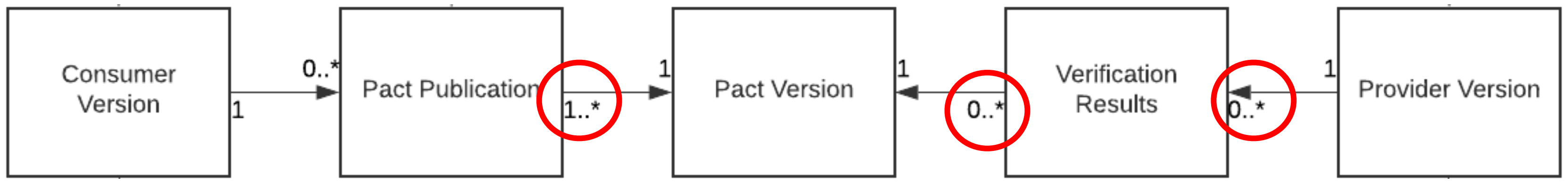
2. Ensure backwards compatibility

- 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
- Record-deployment / record-release with the stage name when you deploy application
- Configure the provider to verify the latest deployed or released pacts as well as the latest main branch.
- {mainBranch: true}, { deployedOrReleased: true }

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 publish with branch name OR do changes with feature toggle and publish with toggle name
- Collaborate with provider team!
- Once feature pact is successfully verified, merge to main branch/ 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

SMARTBEAR

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 - 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-environment 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

Potential feature toggle hack

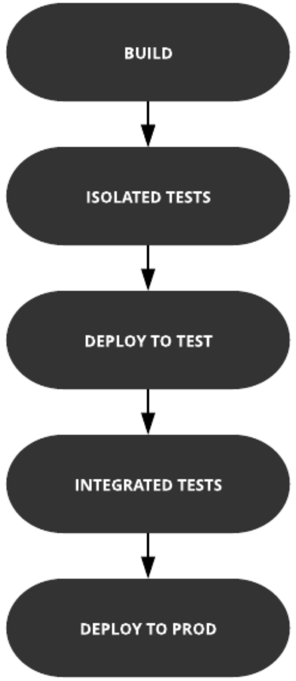
- Consumer
  - Create pact with toggles off
    - Consumer version - sha eg. `effe8a07`
    - Tag with ‘base’
  - Create pact with toggle A on
    - Consumer version sha+toggle_name eg. `effe8a07+feat_a`
    - Tag with toggle name
- Provider
  - Verify pacts with toggles off
    - Provider version -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

CI/CD

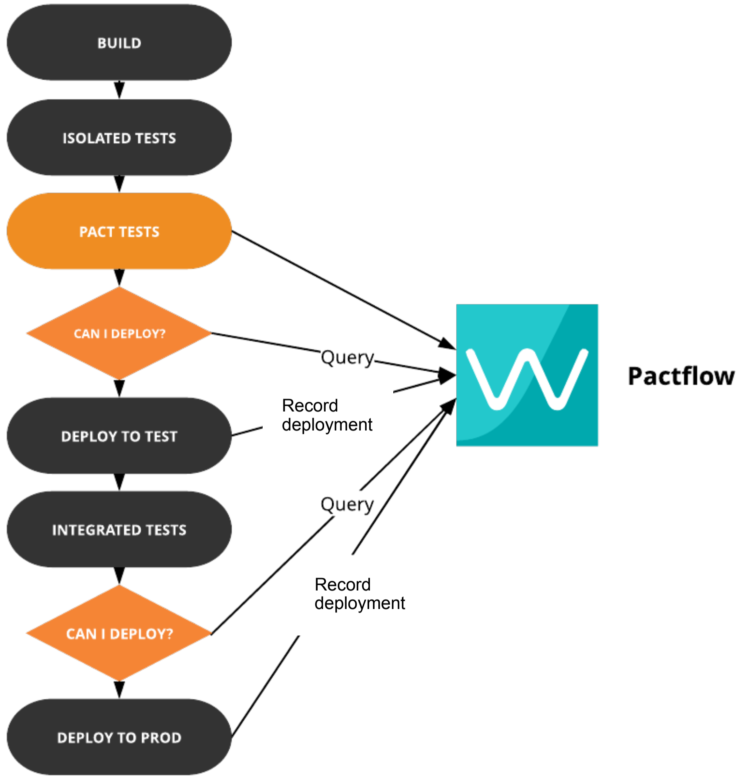
The CI/CD/Pact Broker touchpoints

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

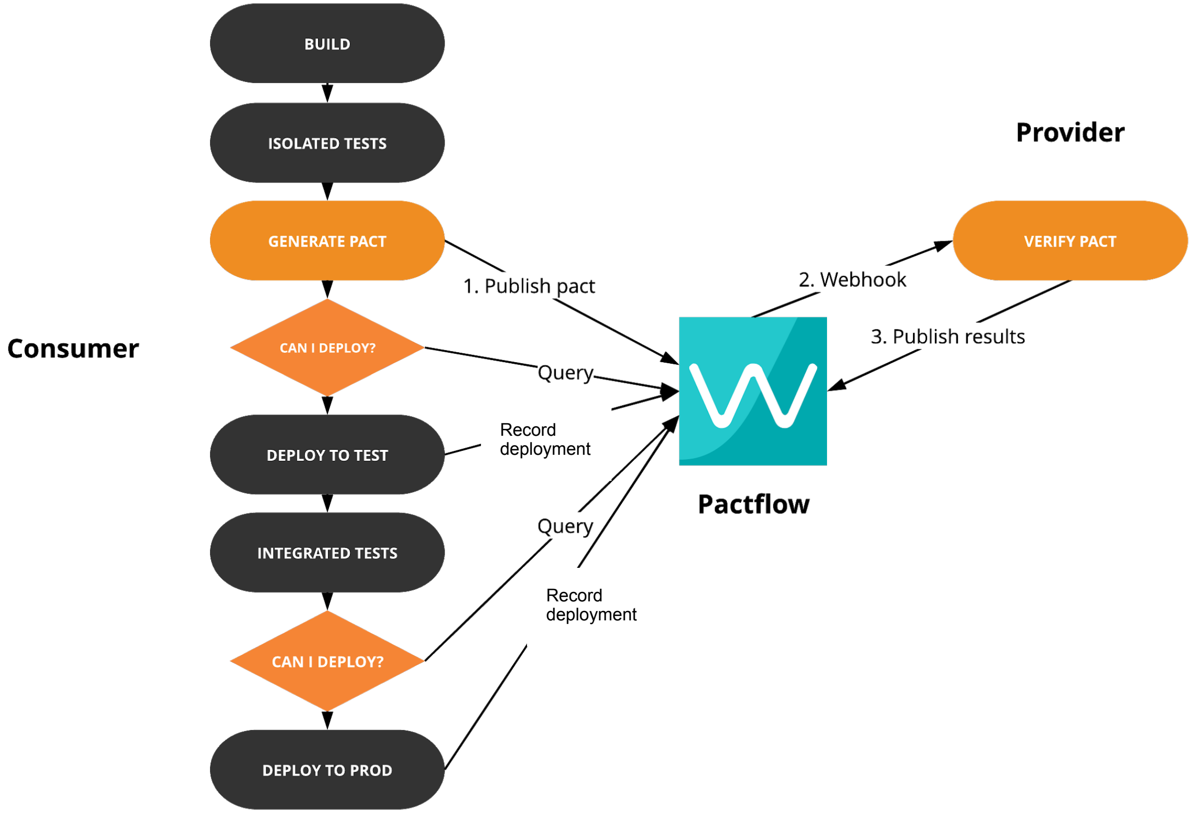
Build pipeline without Pact



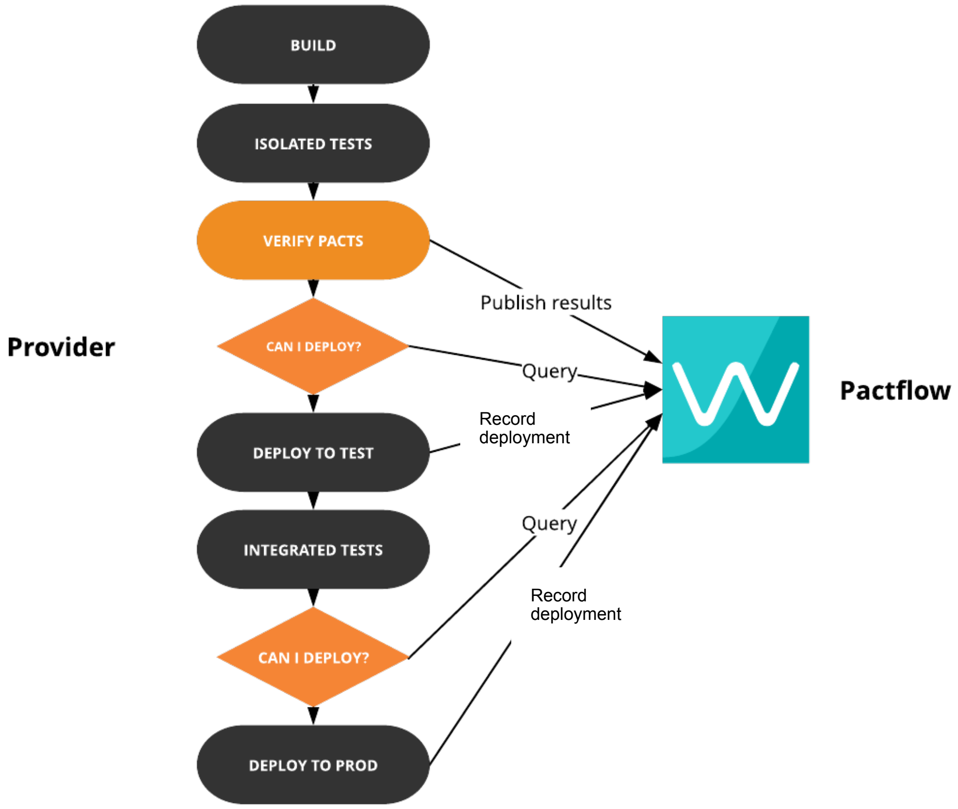
Build pipeline with Pact



Build pipeline with Pact - Consumer



Build pipeline with Pact - Provider



For extra brownie points

- Git statuses
- Slack updates

CI/CD - Question Time

SMARTBEAR

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

SMARTBEAR

Pending pacts - something to note

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

WIP Pacts - the problem

1. Changed pact published with branch ‘feat/foo’
2. ‘contract_requiring_verification_published’ webhook triggers verification - failure
3. Provider implements required changes
4. Provider runs verification for main consumer branch ‘master’ and all deployed and released versions

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

WIP Pacts - the solution

- Changed pact published with branch ‘feat/foo’
- ‘contract_requiring_verification_published’ webhook triggers verification - failure
- Provider implements required changes
- Provider runs verification for main consumer branch ‘master’ and all deployed and released versions - it also automatically verifies any “work in progress” pacts.

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

Organisational Scale

3 key steps to ensure and scale enterprise wide adoption

- Stakeholder buy-in: Using Pact requires collaboration and commitment from each consumer and provider team. Getting buy-in from key stakeholders and aligning on clear goals, objectives and how we will measure success.
- Run a PoC: In our experience, it's best to start small with an initial MVP and a reduced working group on which to establish the appropriate working context, validate concepts and hypotheses. With these learnings it will be easier to scale and expand across the organisation.
- GoP: Once a PoC has demonstrated value in moving forward, the challenge becomes scaling contract testing throughout the organisation. There is value in creating an internal **group of practise** to accelerate adoption, training, eduction and scaling.

Group of Practise (GoP)

- Metrics: Establish metrics and KPI’s to track the speed of adoption across the organisation
- Processes: Develop a standard methodology for team adoption with step-by-step instructions including
  - Workflow of all contract testing related tasks within every spring
  - Recommended development model that follows the release cycle using a branching model
  - Adaptation of the SDLC to include contract testing
  - Definition of the onboarding and monitoring processes for all relevant teams.
- Communication: Facilitate real-time collaboration by enabling the rapid collection of data and ideas
  - Organise regular “open sessions” to address/resolve any questions or issues
  - Regular meetings to track progress with stakeholders (including Dev, QA, SRE, PO)
  - Create an open commenting/annotation system to loop in others and improve coverage
- Tracking: Reporting of KPI progress to all key stakeholders
  - Support and follow up for all teams
  - Feedback to revise and refine all processes and practise

How to onboard teams?

- Buy in: Establish the current problems and how the new model will help solve those challenges
- Training: Step by step training and workshop, based on company domain
- Alignment: Define the main objects and agree on metrics to measure adoption
- Commitment: Reserve time to commit to adoption of new practise
- Adoption: Set a group of Contract testing champions to help build spikes
    - Champions pair programming
    - Continuous support via champions and GoP
- Progress tracking: Follow up and KPI’s monitoring
- Refinement: Open talks and retros with teams to gather feedback

Quantitative Metrics

- Engagement
  - No of pipelines with can-i-deploy tool enabled
  - No of teams/projects with contracts in Pactflow
  - No of endpoints covered by contracts (contract testing coverage)

- Impact
  - Time spent on manual testing (we want to see a reduction)
  - Deployment frequency - how often do you release?
  - Lead time for change - how long does it take to get a release from commit to production
  - Change failure rates - how frequently does a change result in a failure as a percentage
  - Mean time to recovery (MTTR) - how long does it take to recover from a failure

Qualitative Metrics

- How is the team feeling with the new tools?
- What are the plans and gains?
- How active is the tool?
- How active and supportive is the community?

SMARTBEAR

Example results from a customer after 7 sprints

- Governance
  - 2 teams training in CT techniques, 4 CT champions
  - 8 KPI’s defined
  - GoP setup to engage teams and continue CT using Pact gaining adoption
  - Implementation of RBAC in Pactflow
- Technical Aspects
  - 6 endpoints covered with CT
  - 2 Jenkins pipelines integrated with Pactflow
  - 2 bugs detected which were present in production environments
- Times
  - 8 hours to develop first CT test
  - 3 weeks to integrate first pipeline
  - 4 weeks from the project start date to discover first bug

Learnings: Contract testing champions as a key role

- Main role to accelerate the process
    - Evangelises the rest of the teams
    - Supports the practise also over the organisation
    - Ownership of the practise
    - Autonomy and trust of the organisation to make decisions

Learnings: A Group of Practise is a must to expand adoption

- Group of Practise should always be established to guide, support & continuously track the degree of adoption of the new practise within an organisation.
  Key responsibilities include
  - Evangelise about the use and benefits of contract testing
  - Train teams in the technique
  - Establish metrics for monitoring and checking the degree of success
  - Collect feedback from the teams
  - Resolve dependencies and impediments
  - Provider teams with adequate resources to implement contract testing efficiently
  - Prepare status reports
  - Organise meetings, seminars, workshops, lectures on contract testing within the organisation

Learnings: Monitoring metrics are key to track success

- For any process of implementation of new practises in an organisation, it is important to establish monitoring metrics that allow us to know the degree of evolution of the same.
  This data will give us information on:
  - Degree of implementation in teams
  - Effectiveness of contract testing within the organisation
  - Alignment with business objectives
  - Fulfilment of expectations
  - Detection of deviations or inefficiencies

Thank you - Question Time