SMARTBEAR
PactFlow

ABOUT ME

1. Developer Advocate @ SmartBear
2. Tester/Developer/Consultant since 2007
3. Once left IT as a career to restore old automobiles, but returned as I enjoyed conversing, learning and teaching

you54f

Agenda

& housekeeping

The 2 - 2.5 hour workshop covers:

1. Introduction to Pact (presentation)
2. Hands-on lab (step 1-5)
3. *5-10 minute break*
4. Hands-on lab cont. (step 6-12)
5. Q&A

🔴 Session is being recorded

SMARTBEAR PactFlow

The numbers

Four key indicators of high performing organisations[1]

Need < 1 day **lead time** for changes = **106x** faster time from commit -> deploy

Are able to **deploy** on demand = **208x** more deployments

Have **change failures** rates < 15% = **7x** lower change failure rates

Can **restore services** within 1 hour = **2604x** faster MTTR

[1] Data from the DORA 2019 State of DevOps report

SMARTBEAR PactFlow

The numbers

Challenges facing the market

Only **20%** of companies are “elite” performers[1]

81% of teams spend a third of their time or more on **fixing environments**[2]

36% of teams are impacted by wait times and cost of **test environments**[2]

76% spent one third of their time or more managing **test data**[2]

[1] Data from the DORA 2019 State of DevOps report
[2] Data from a Capgemini report on continuous testing in March 2019

SMARTBEAR PactFlow

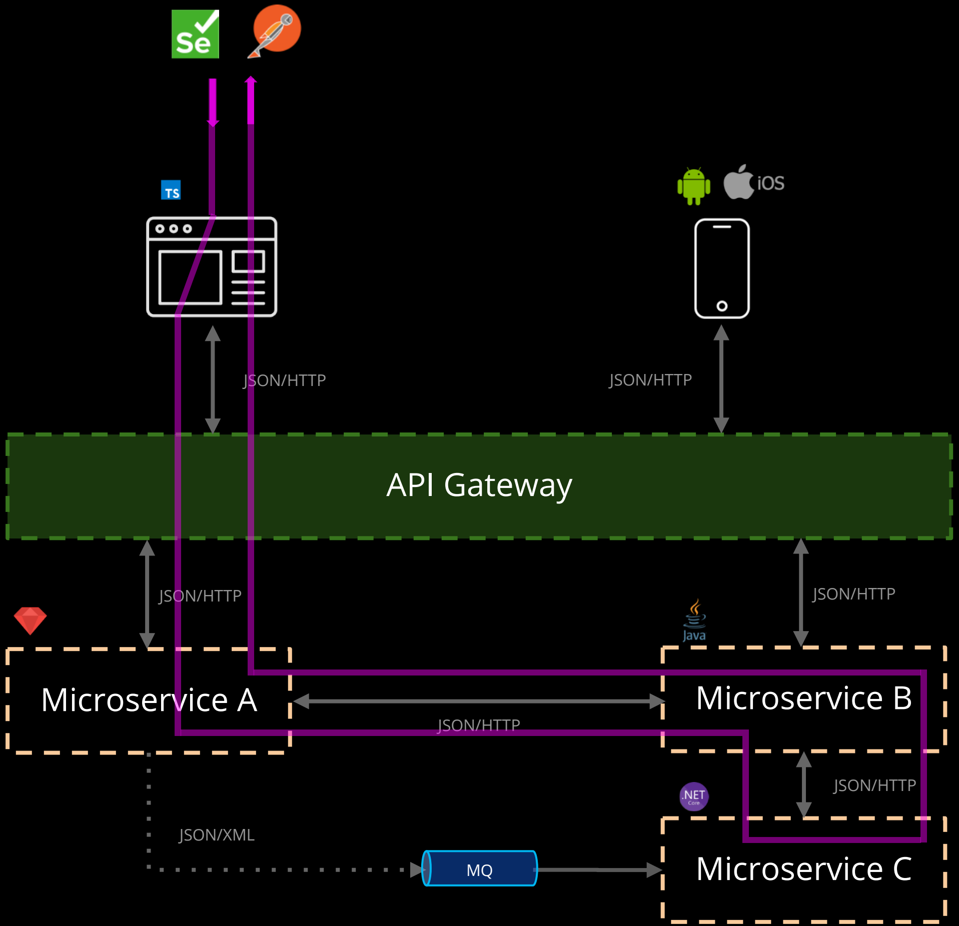
In **2013** we created **Pact**, an Open Source tool to solve this problem. In 2019, we launched **PactFlow** to enable organisations to do this *at scale.* In 2022, we were acquired by **SmartBear** allowing us to fit our contract-testing story, alongside a suite of tools designed to address the challenges of API and Product development, affecting the teams of today, and generations beyond.

SMARTBEAR PactFlow

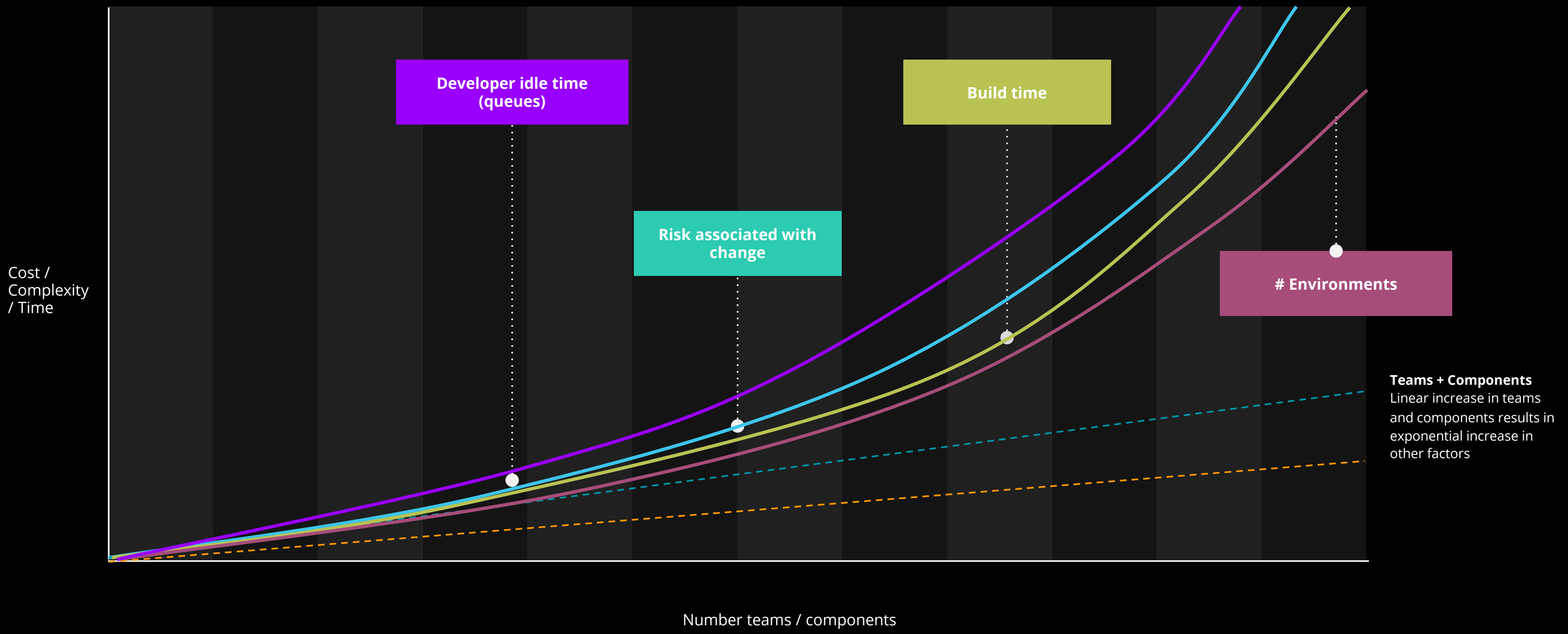
The old way...

Why this is hard

- Slow
- Fragile
- Hard to debug
- Test data management + environment management
- Coverage?
- All-at-once painful deployments
- Teams wait on build queues



Scaling



Teams + Components
Linear increase in teams and components results in exponential increase in other factors

“Integration tests are a **scam**”

- JB Rainsberger

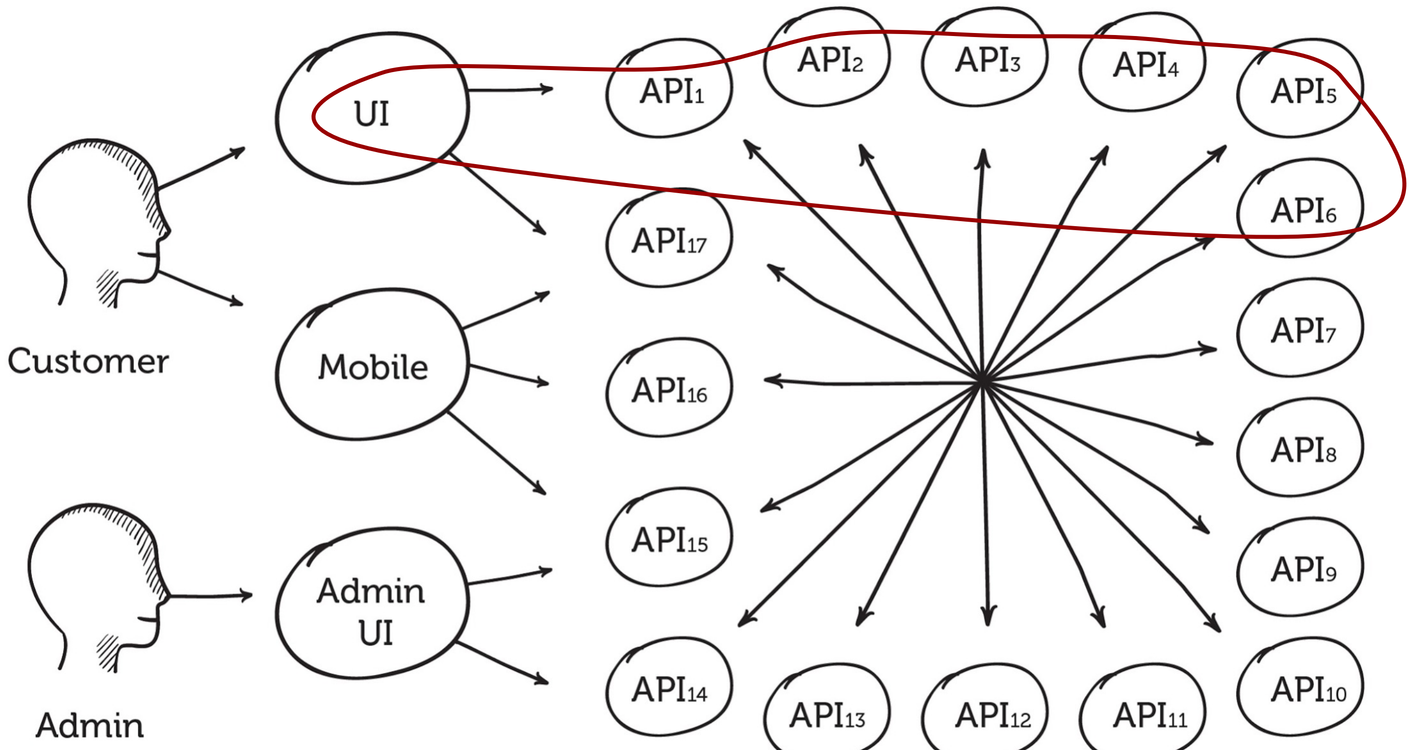
Scam, you say? Justify!

Integrated tests are:

- Slow
- Fragile
- Hard to manage

When they fail, you can’t point to the problem!

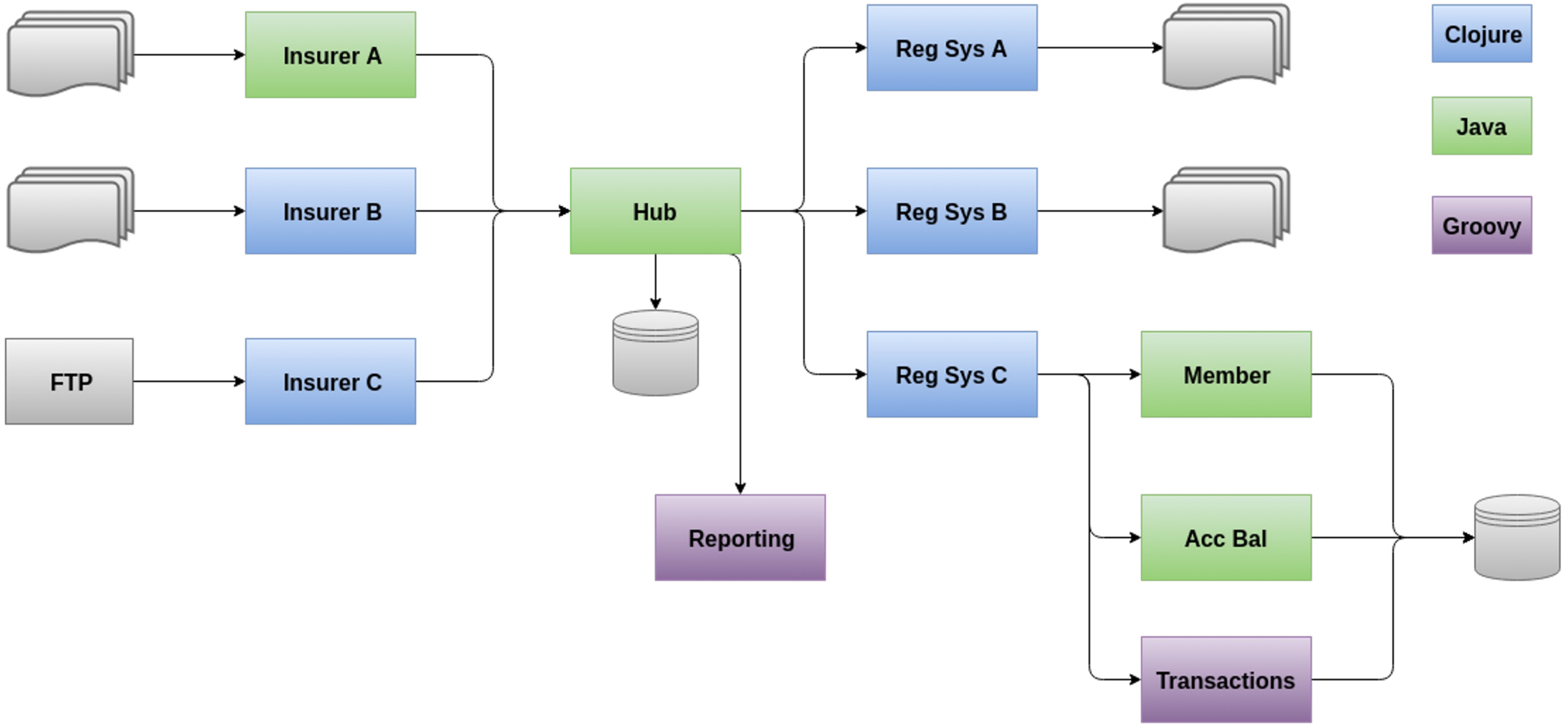
HOW WE TEST MICROSERVICES NOW



Branches per box vs **test cases required**

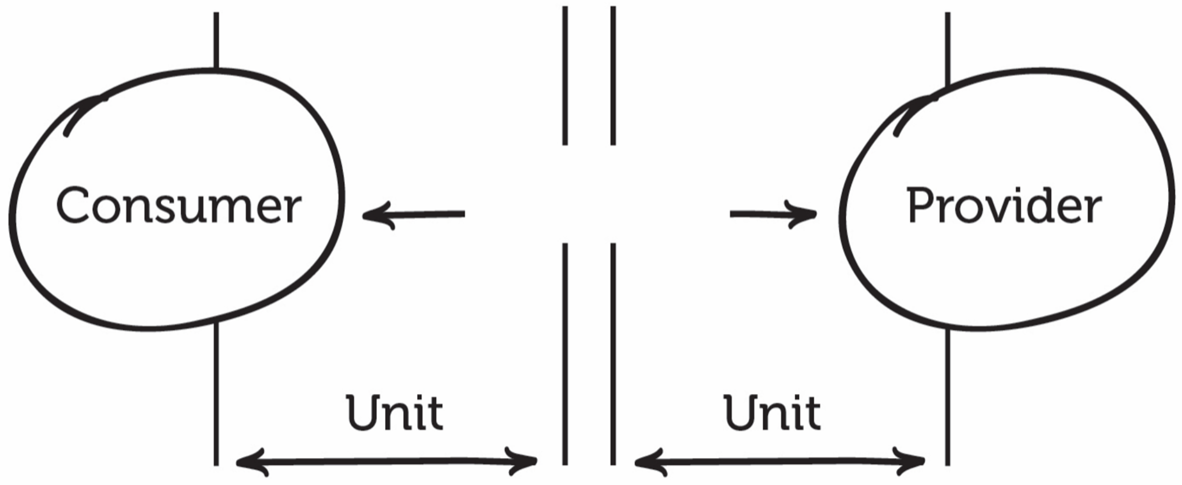
2 code branches = 128 tests
5 code branches = 78,125 tests
10 code branches = 10M tests

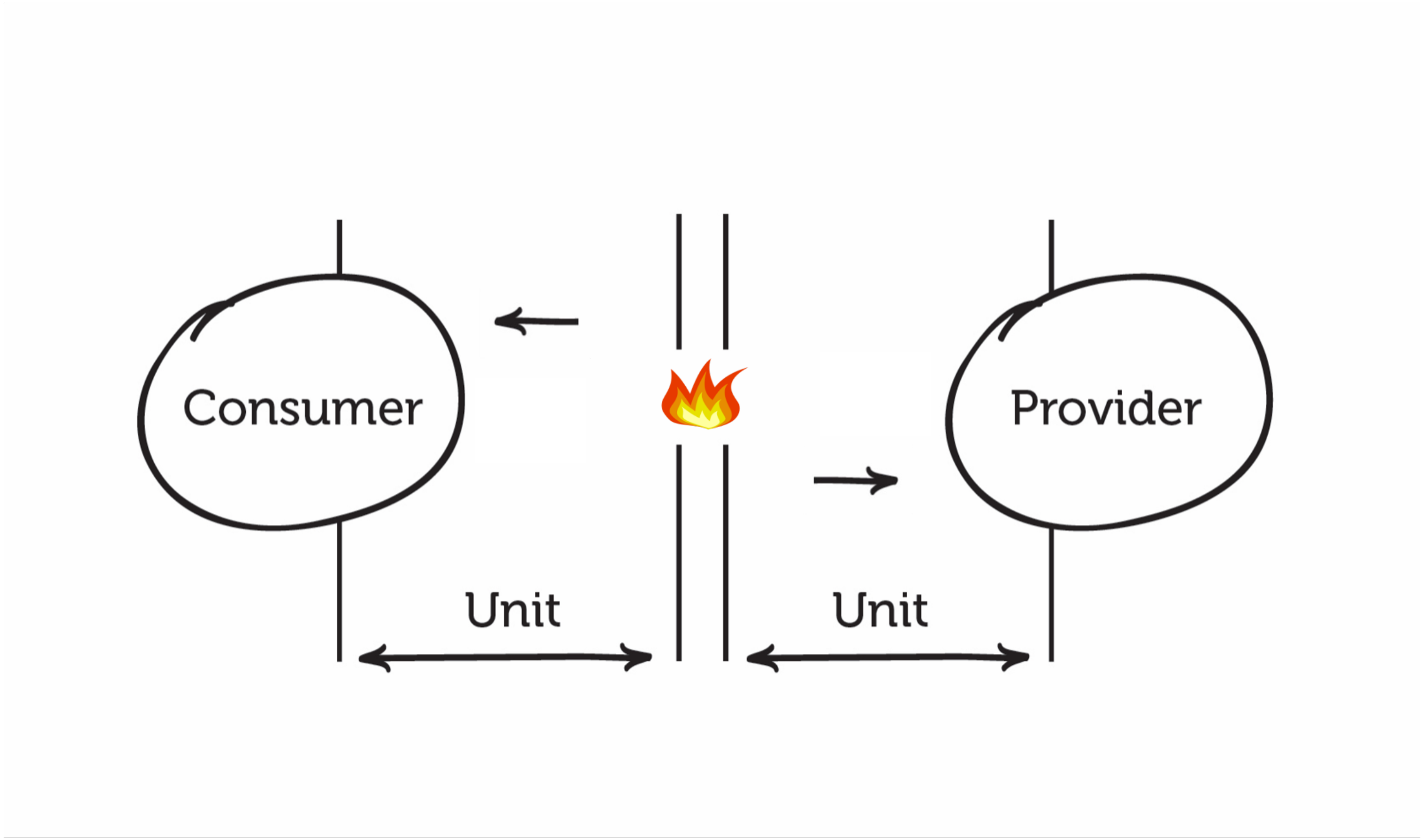
HOW WE TEST MICROSERVICES NOW



Good tests have the **exact opposite** properties

Mocks to the rescue?


Consumer
Provider
Unit
Unit


Consumer
Provider
Unit
Unit

Mocks

Solved problems

- Fast feedback
- Few dependencies
- No dedicated environment
- Reliable
- Easy to debug

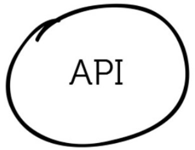
New problems

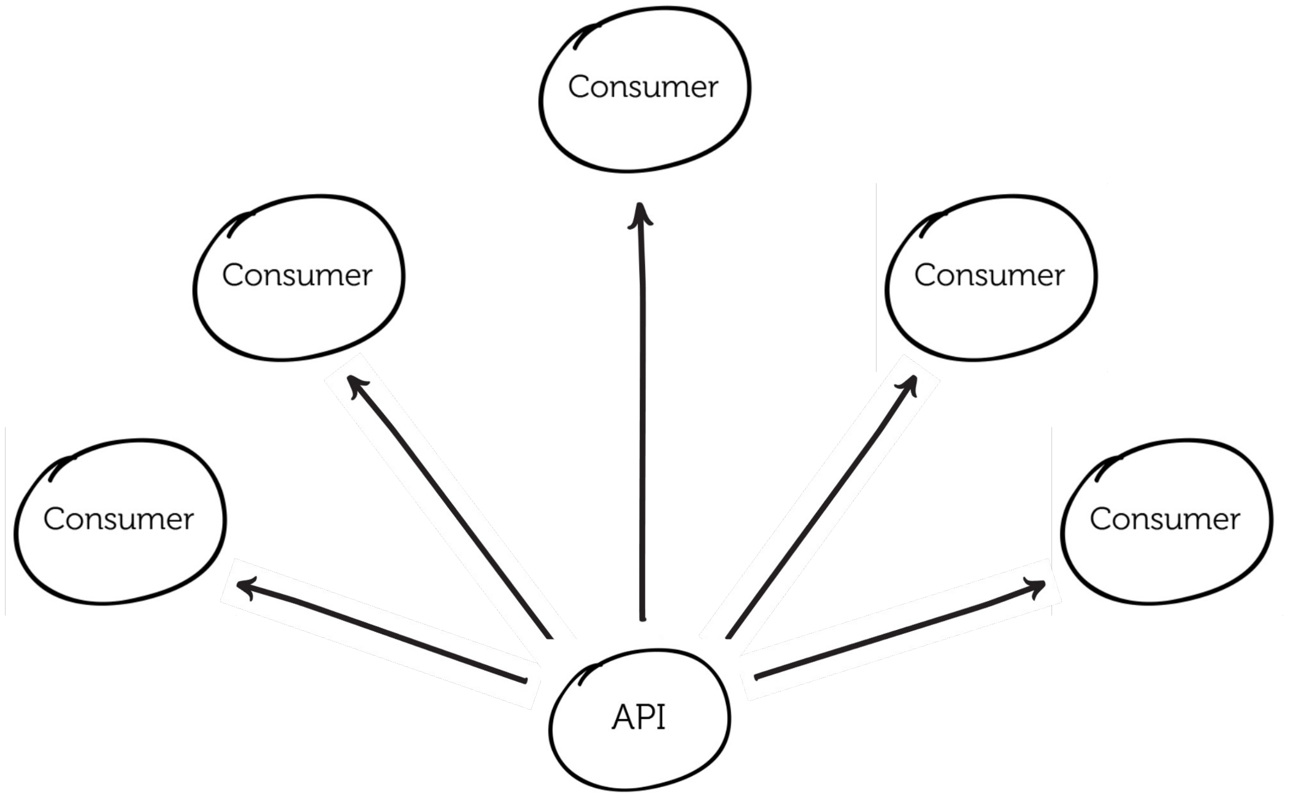
- Hard to keep both sides in sync

How about API Specs?

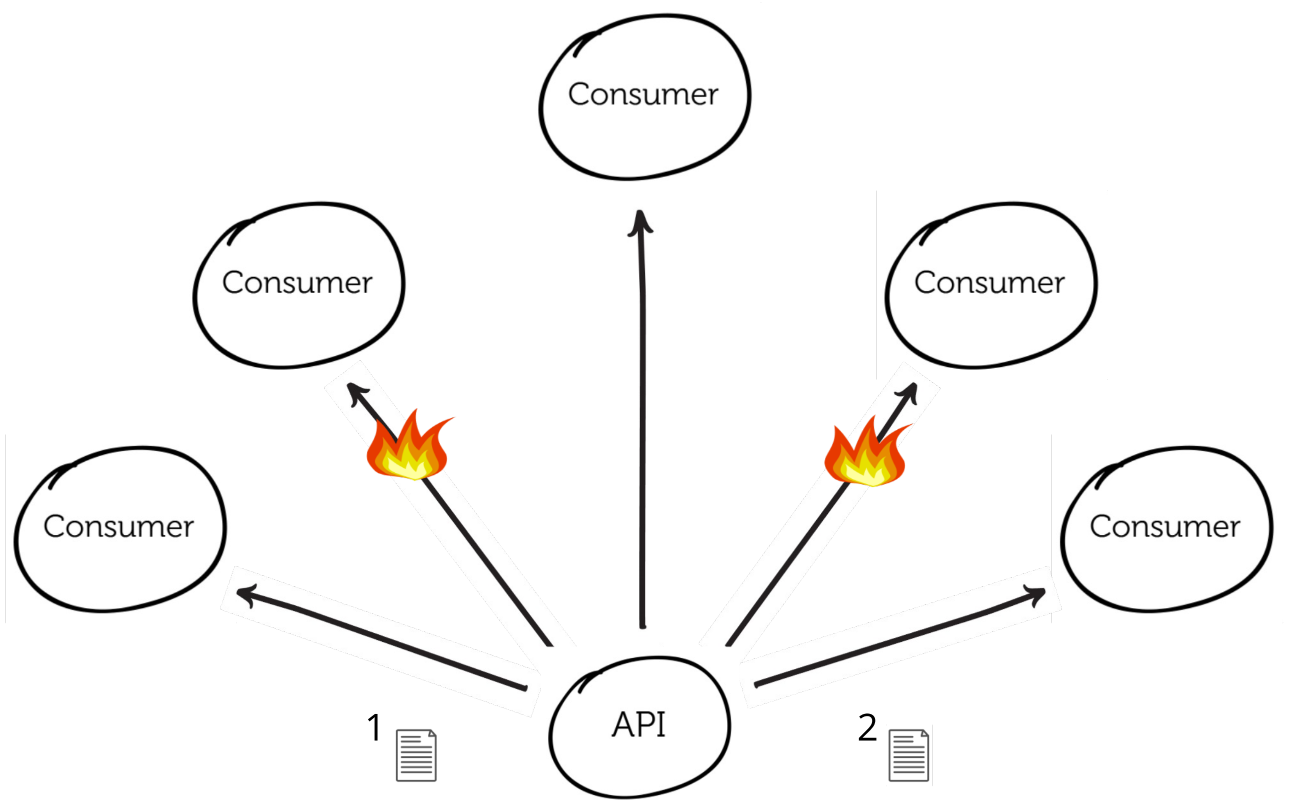
How to: Spec first development

1. Architect independent of teams postulate API requirements
2. Document perfect API (Swagger/OAS etc.)
3. Create said API
4. Publish said document to consumers
5. Repeat steps 1-4


API


Consumer
Consumer
Consumer
Consumer
Consumer
Consumer
API

Specification first design



Specification first design

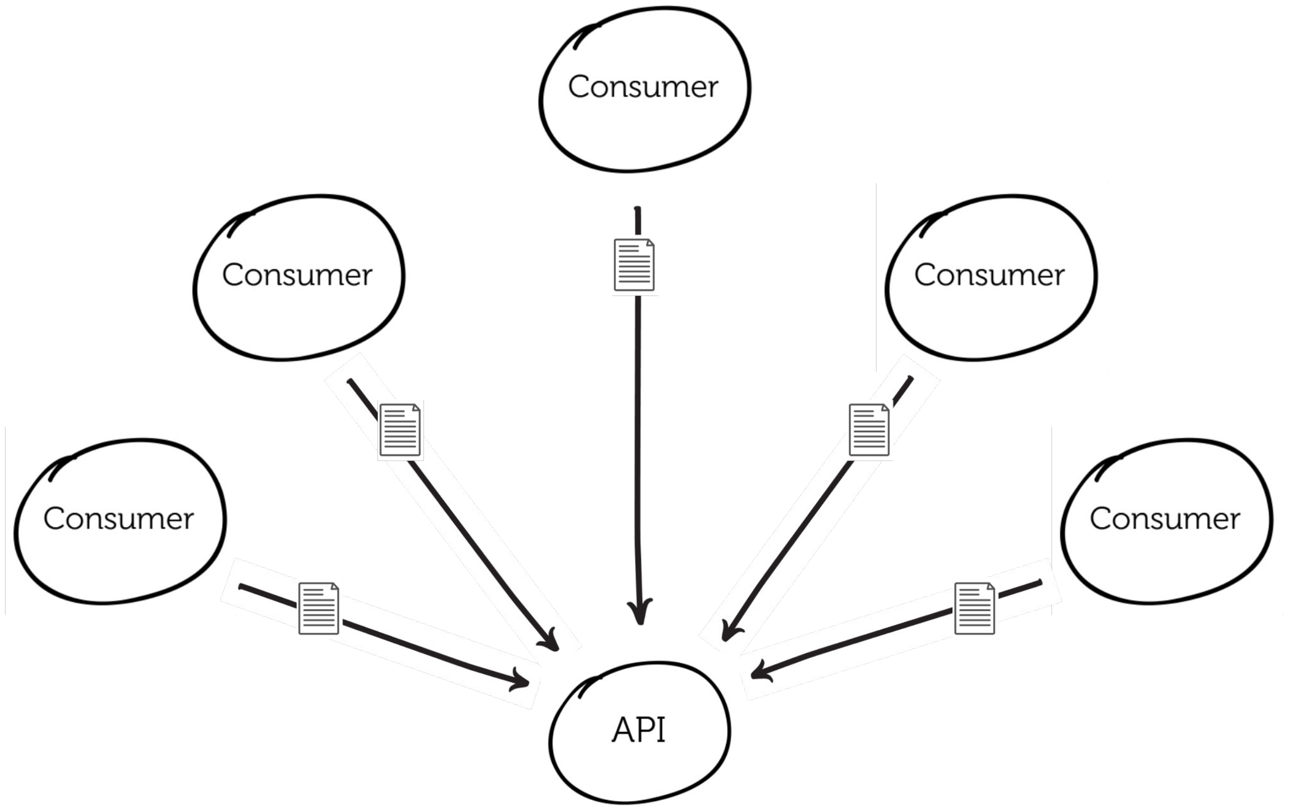
Solved problems

- Good documentation
- Aides discoverability and communication between teams/organisations
- Clearer expectations on API

New problems

- Who is using my API?
- Requires diligence to ensure backwards compatibility
- Developers hate maintaining versioning
- Limited by expressiveness of specification (vague)
- = Hard to get 100% coverage (can only say “not incompatible”)

Enter Consumer Driven Contracts


Consumer
Consumer
Consumer
Consumer
Consumer
API

Consumer Driven Contracts

Benefits

You know when you **break a consumer**

You get a form of **documentation**

You can test things **independently**

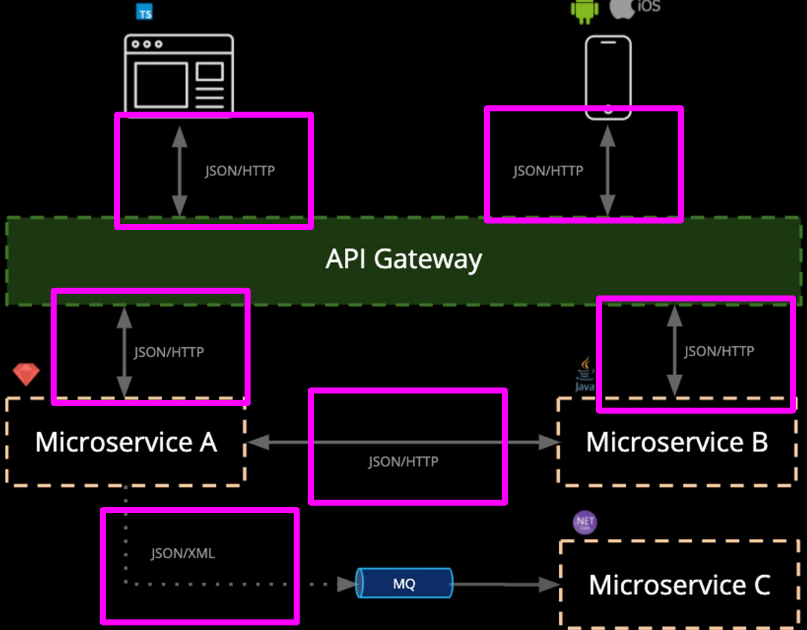
What is Contract Testing?

An alternative approach

Benefits:

- **Simpler** - test a single integration at a time - without having to deploy
- No **dedicated test environments** - run on a dev machine
- Get **fast**, reliable feedback
- Tests that scale **linearly**
- **Deploy** services independently

Pact **removes** the need for complicated release coordination: we have static knowledge about system compatibility.



What is Pact?

Microservice testing made easy

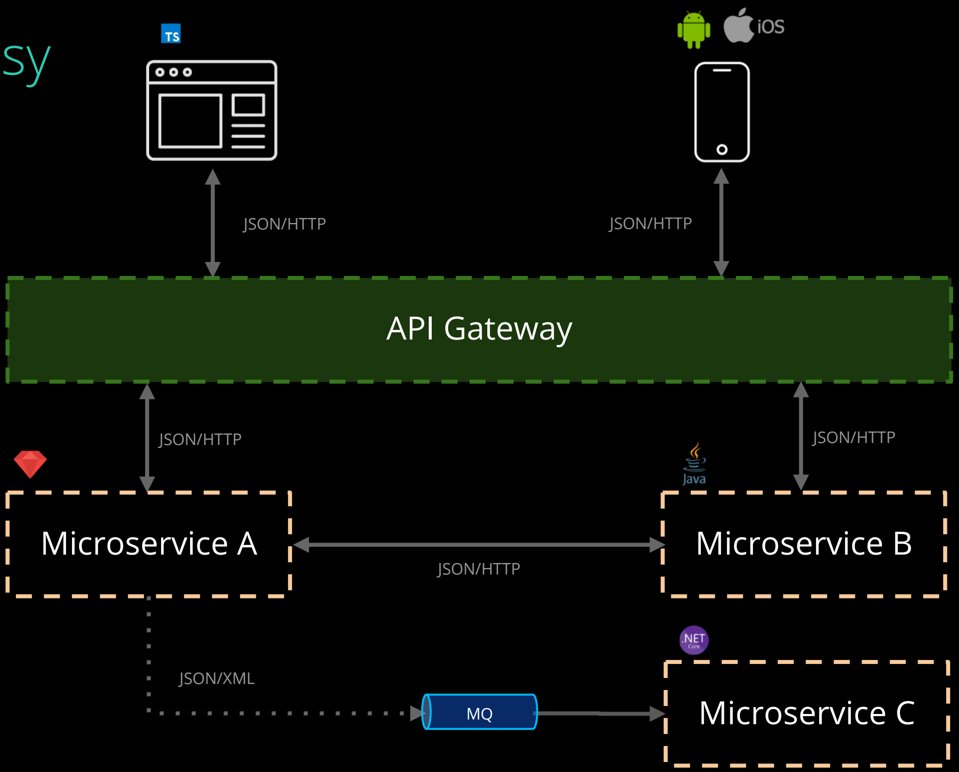
Pact is an Open Source, *consumer driven contract testing* tool that makes it easy to test microservices quickly, independently and release safely.

Use cases:

- Javascript web applications (e.g. React)
- Native mobile applications
- RESTful microservices with JSON and XML
- Asynchronous messaging (e.g. MQ)

Goals:

- Removing end-to-end integrated tests
- Reducing reliance on complex test environments



Open Source

…and in your preferred language

TS .NET Core Java GO JS

php R

SMARTBEAR
PactFlow

Concepts

Interaction Types

Types of interactions:

	Use Cases
• Synchronous/HTTP	REST (JSON/HTTP), SOAP (XML/HTTP), JSON-RPC, GraphQL
• Asynchronous/Messages	Kafka, Fire and Forget, Server Push
• Synchronous/Messages	gRPC/protobufs, Websockets, MQTT, Data Pipelines, AWS Lambda

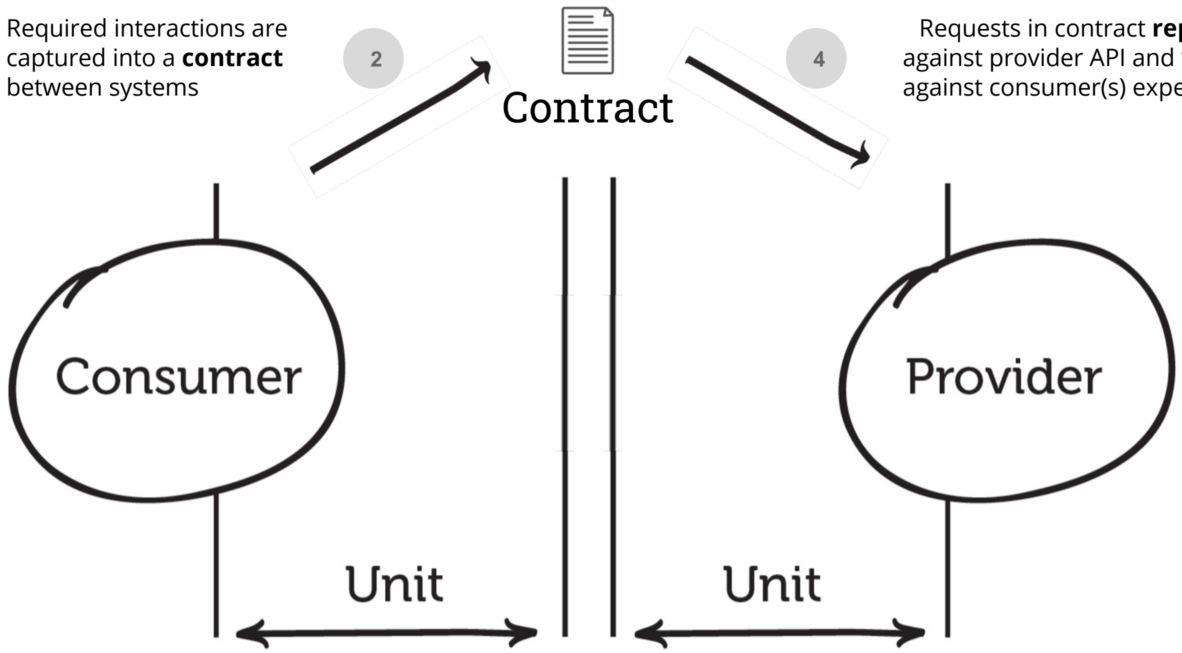
By combining interaction types with the various Plugin capabilities, rich support for various frameworks and protocols emerge.

https://github.com/pact-foundation/pact-specification/tree/version-4#interactions

HOW PACT WORKS

(HTTP)

SMARTBEAR PactFlow



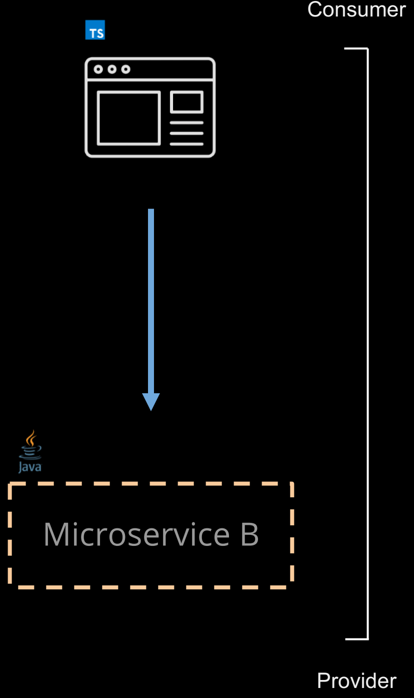
3 Contract is **shared** amongst teams to enable collaboration, using tools like Pactflow

2 Required interactions are captured into a **contract** between systems

4 Requests in contract **replayed** against provider API and **verified** against consumer(s) expectations

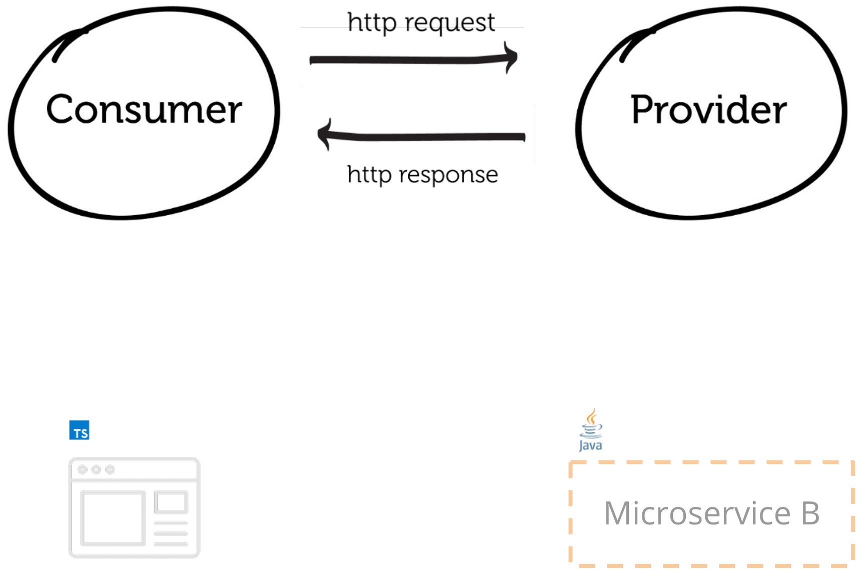
1 Consumer **unit tests** its behaviour against provider mock

5 Provider tests mock out any other systems, so it can be **tested in isolation**

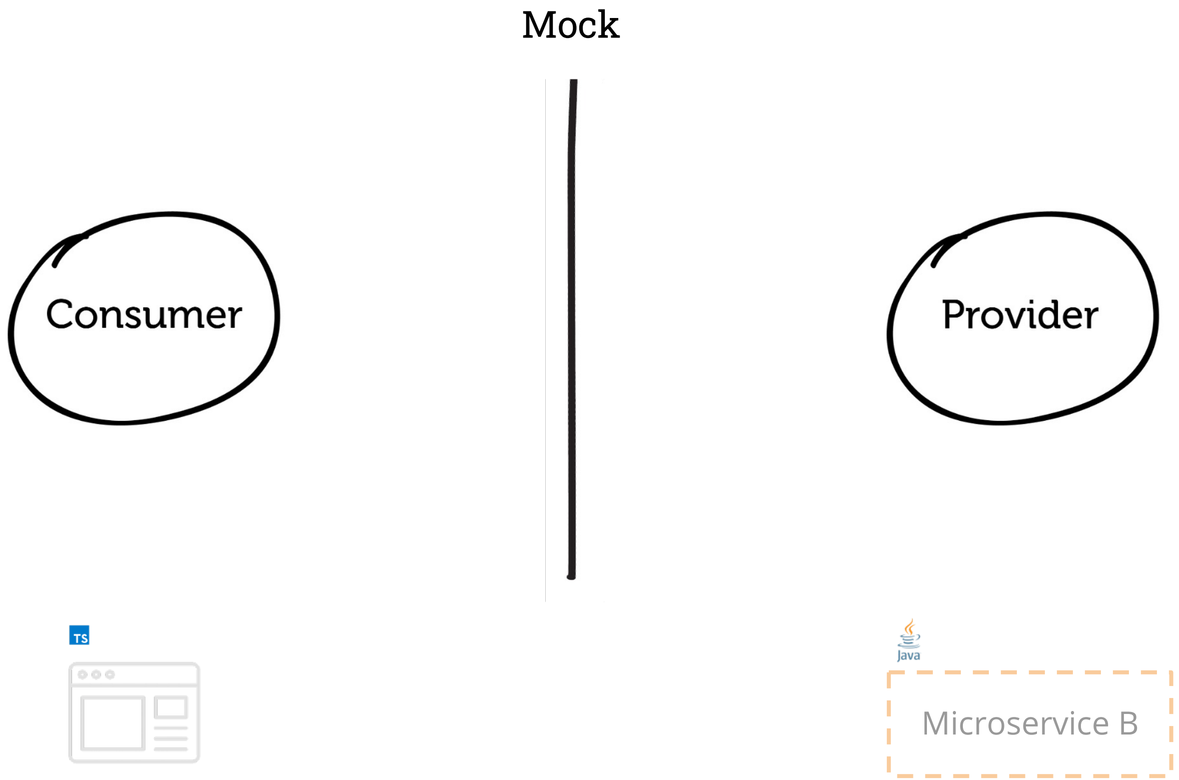


Contract

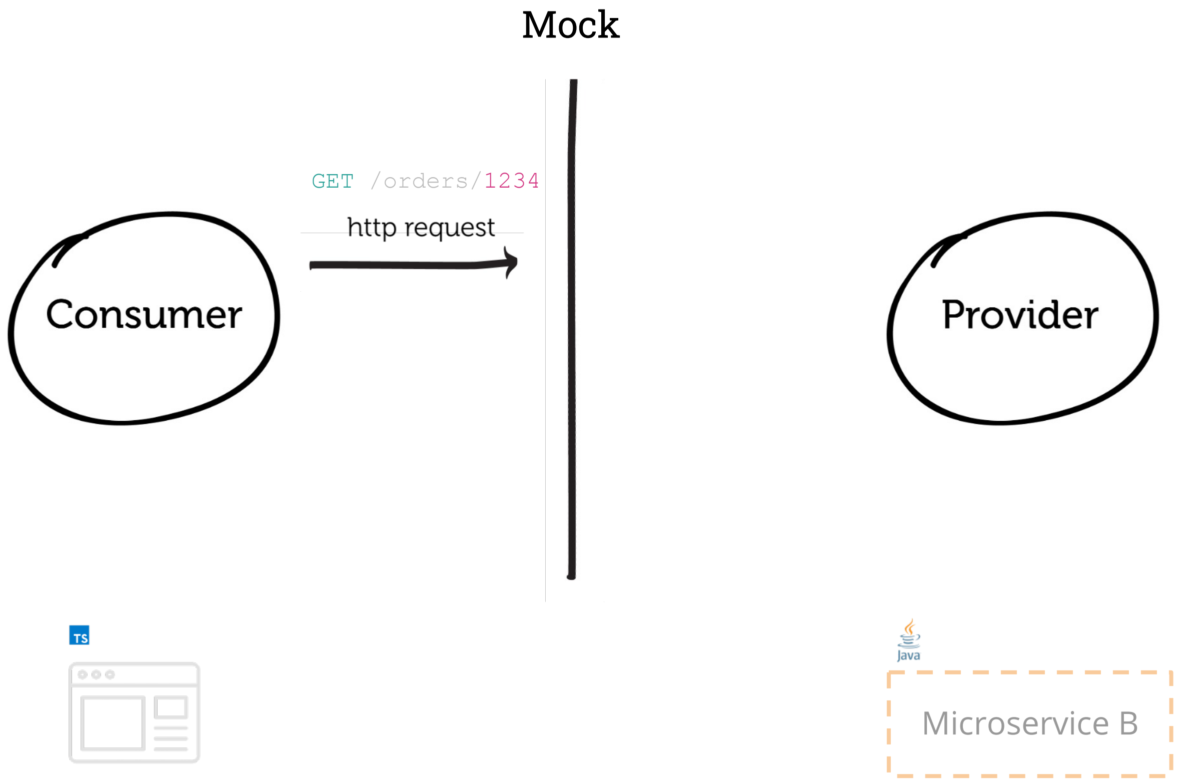
```
GET /orders/1234

{
 "id": 1234,
 "items":[
     ...
  ],
}
```


Consumer
http request
http response
Provider
TS
Java
Microservice B

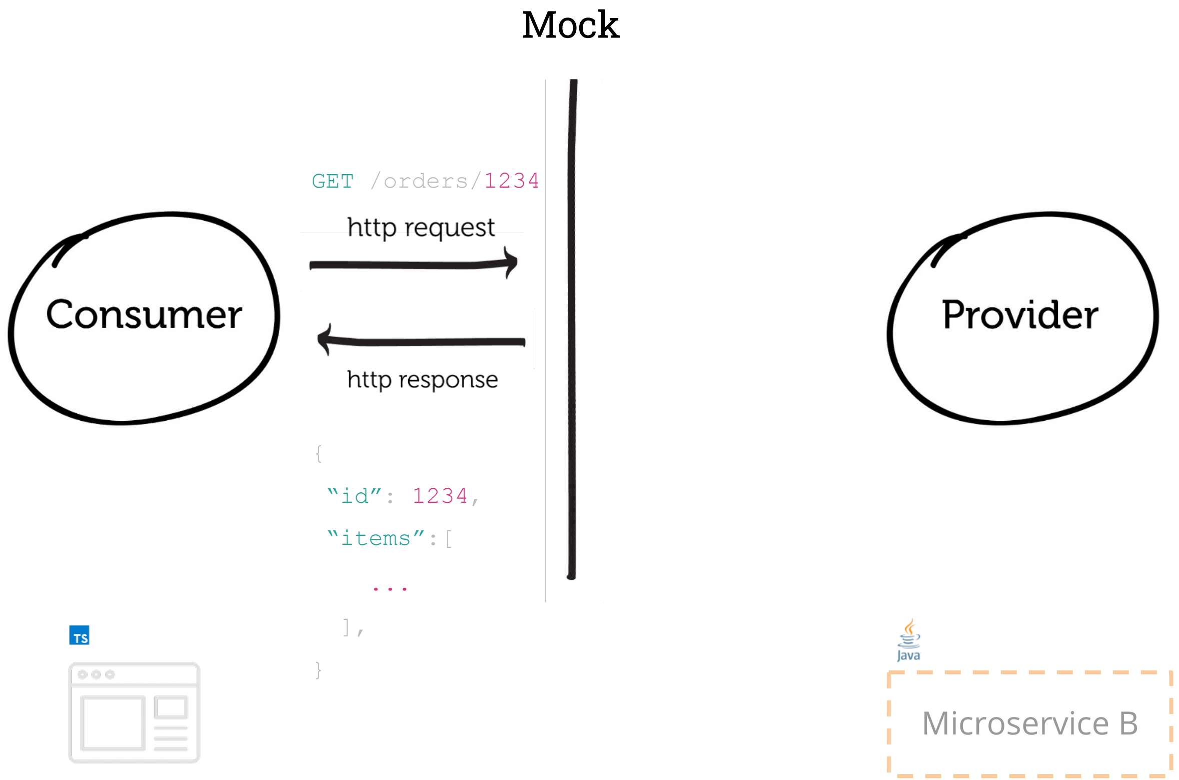
Step 1: test the consumer (contract capture)



Step 1: test the consumer (contract capture)

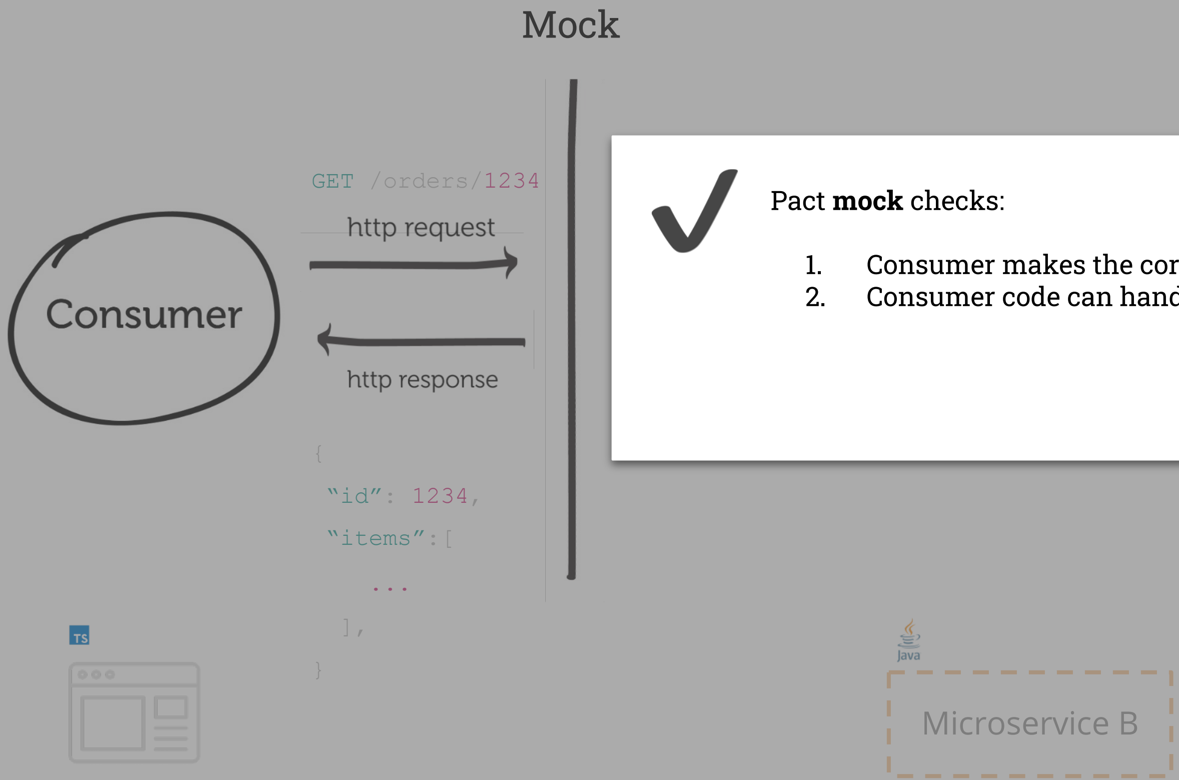


Step 1: test the consumer (contract capture)



```
{
 "id": 1234,
 "items":[
     ...
  ],
}
```

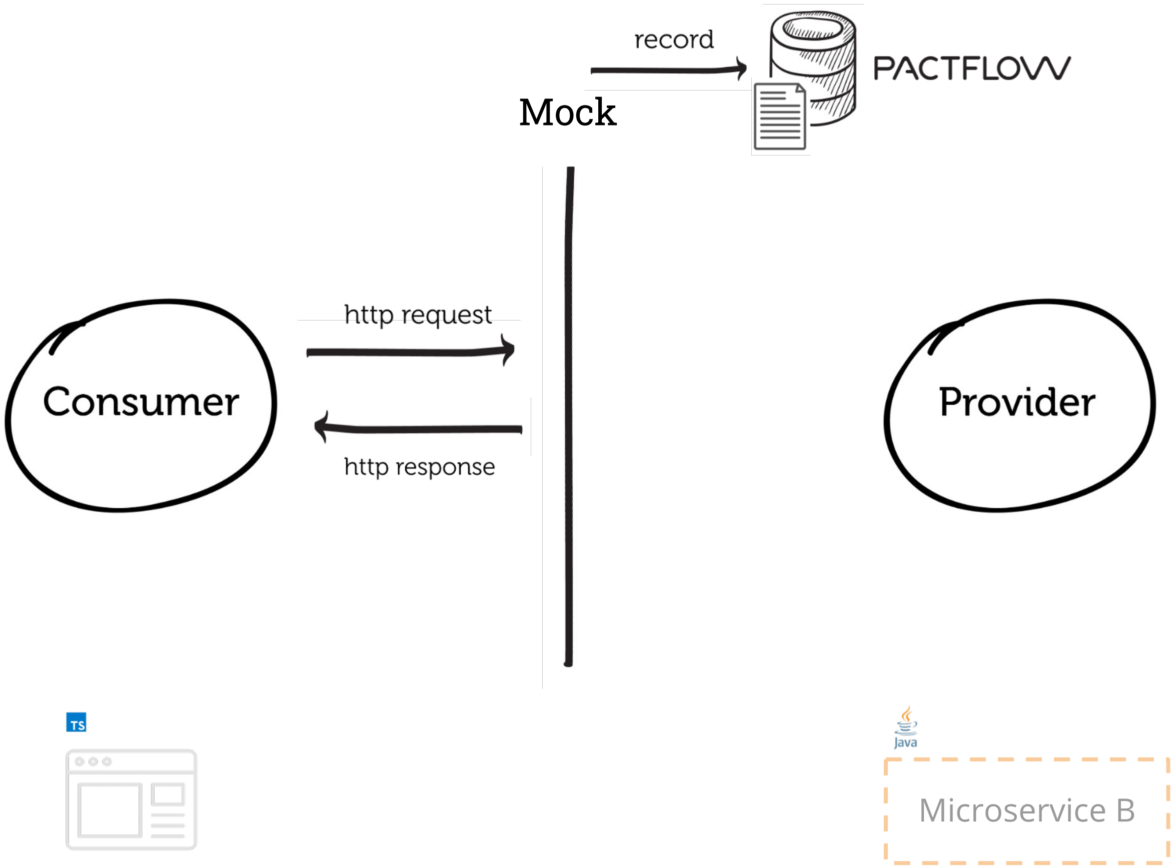
Step 1: test the consumer (contract capture)



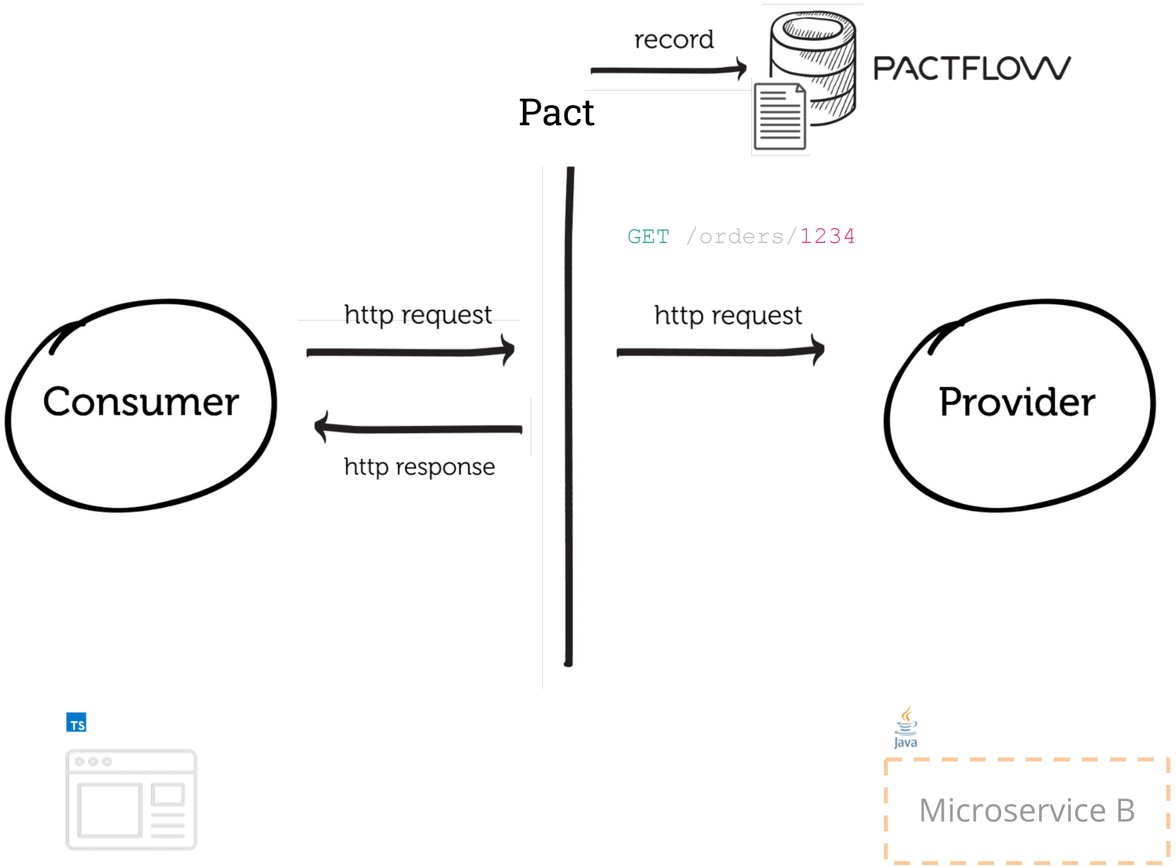
Pact **mock** checks:

1. Consumer makes the correct call to API
2. Consumer code can handle the response

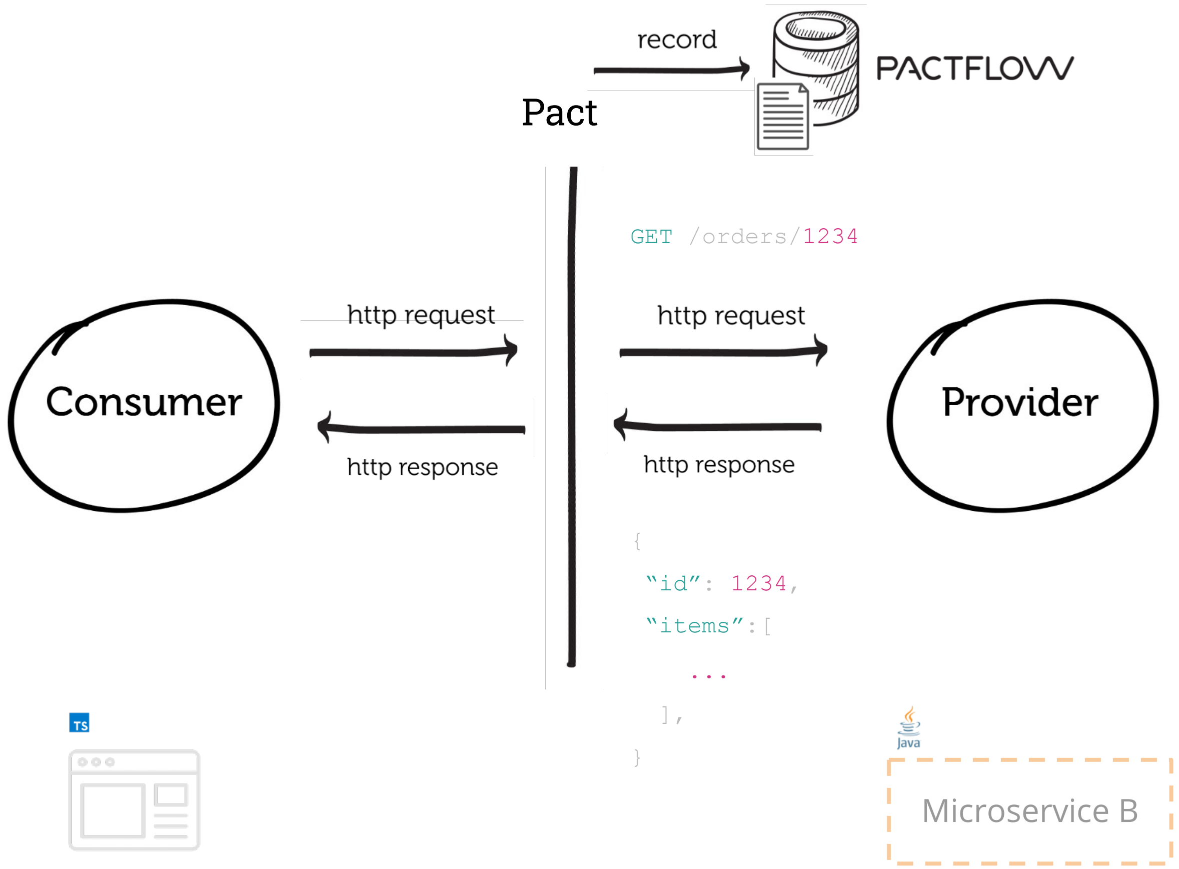
Step 2: share the contract with the Pactflow



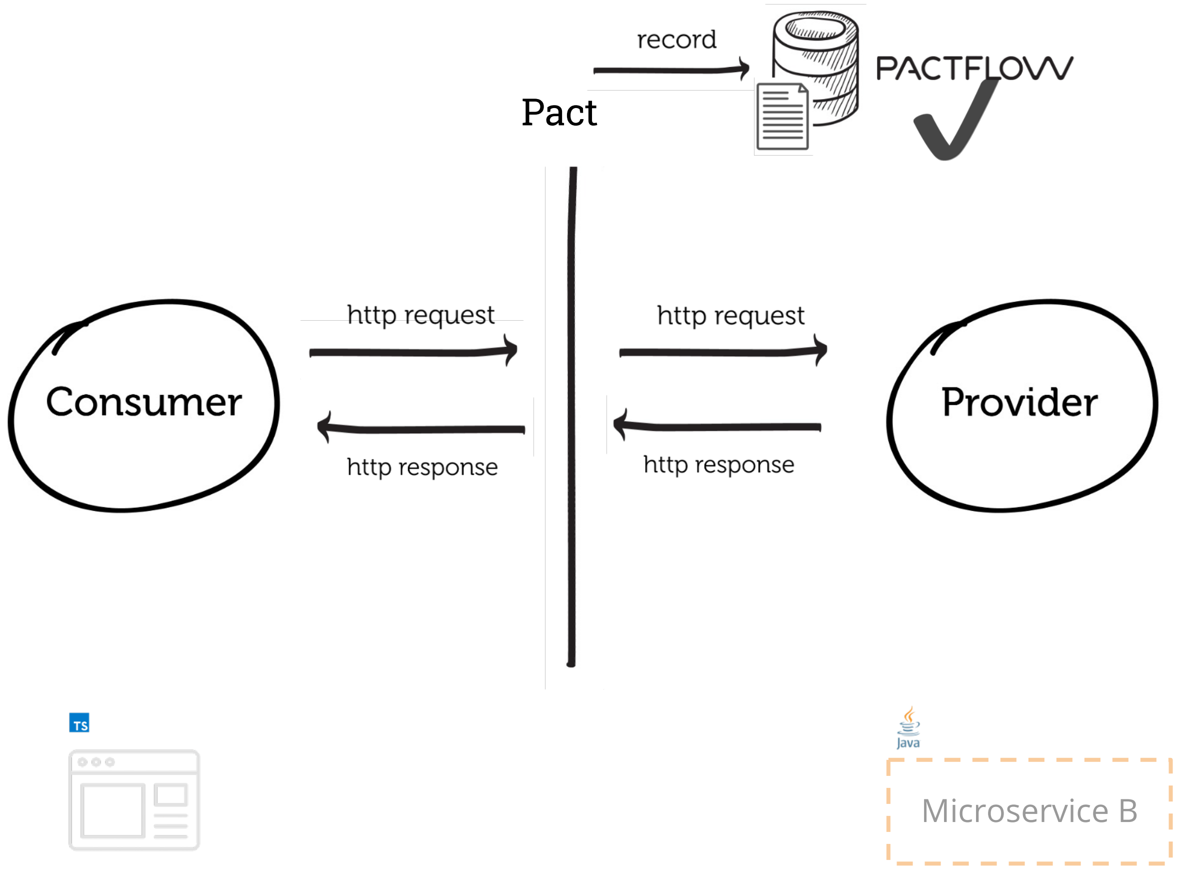
Step 3: test the provider (contract validation)



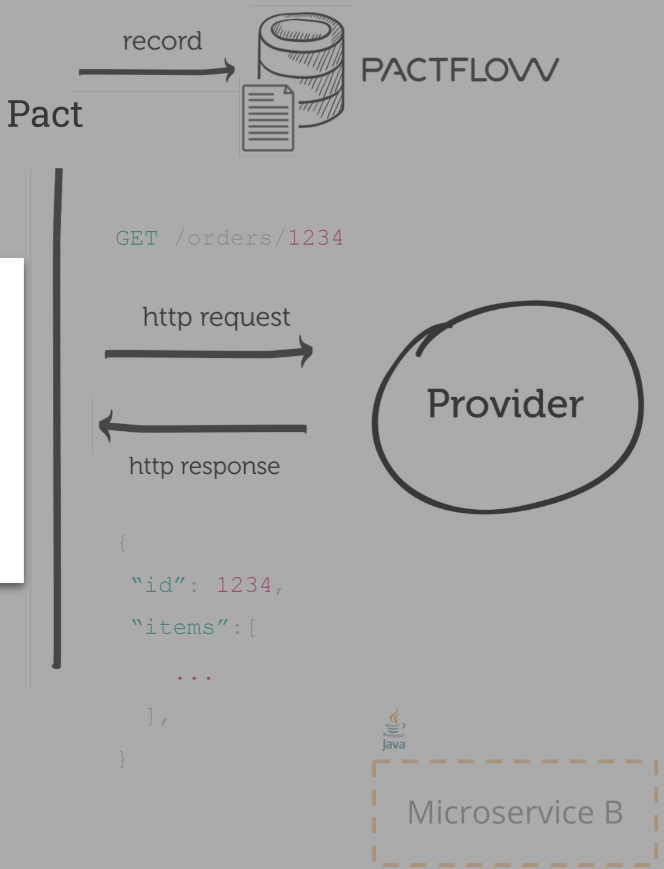
Step 3: test the provider (contract validation)



Step 3: test the provider (contract validation)



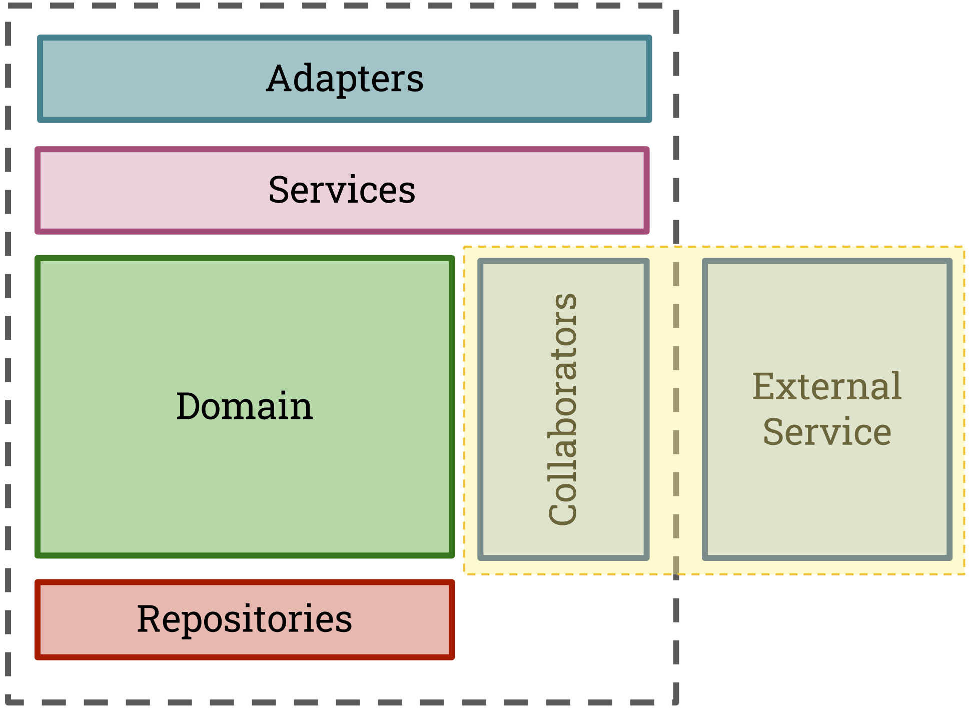
Step 3: test the provider (contract validation)



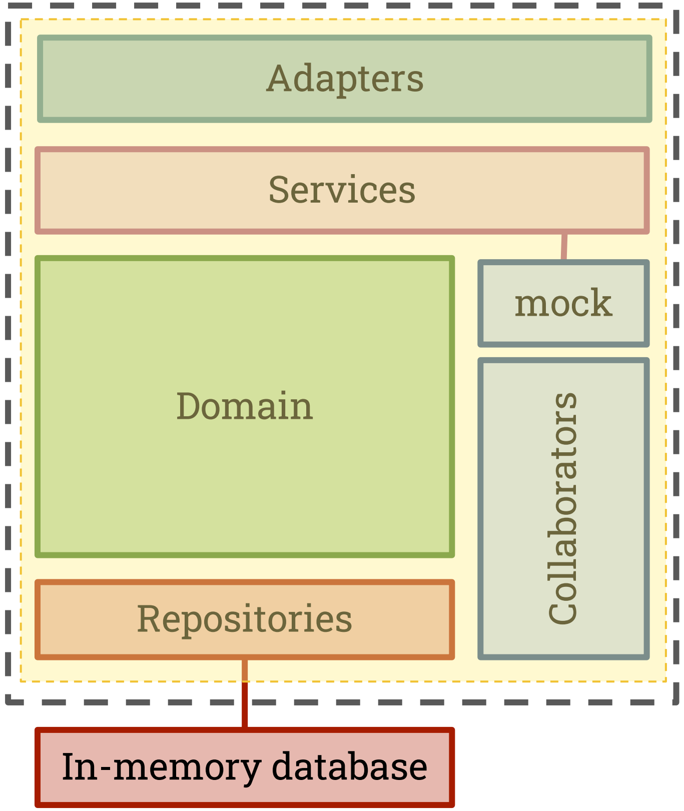
Pact **verifier** checks:

1. All known consumers of the provider
2. Provider can respond to all requests for each consumer
3. For each request, the response (headers, status, body etc.) matches rules in the contract

Scope of consumer test



Scope of Provider Test



HOW IT WORKS

(bi-directional contracts - **PactFlow only feature**)

what are bi-directional contracts?

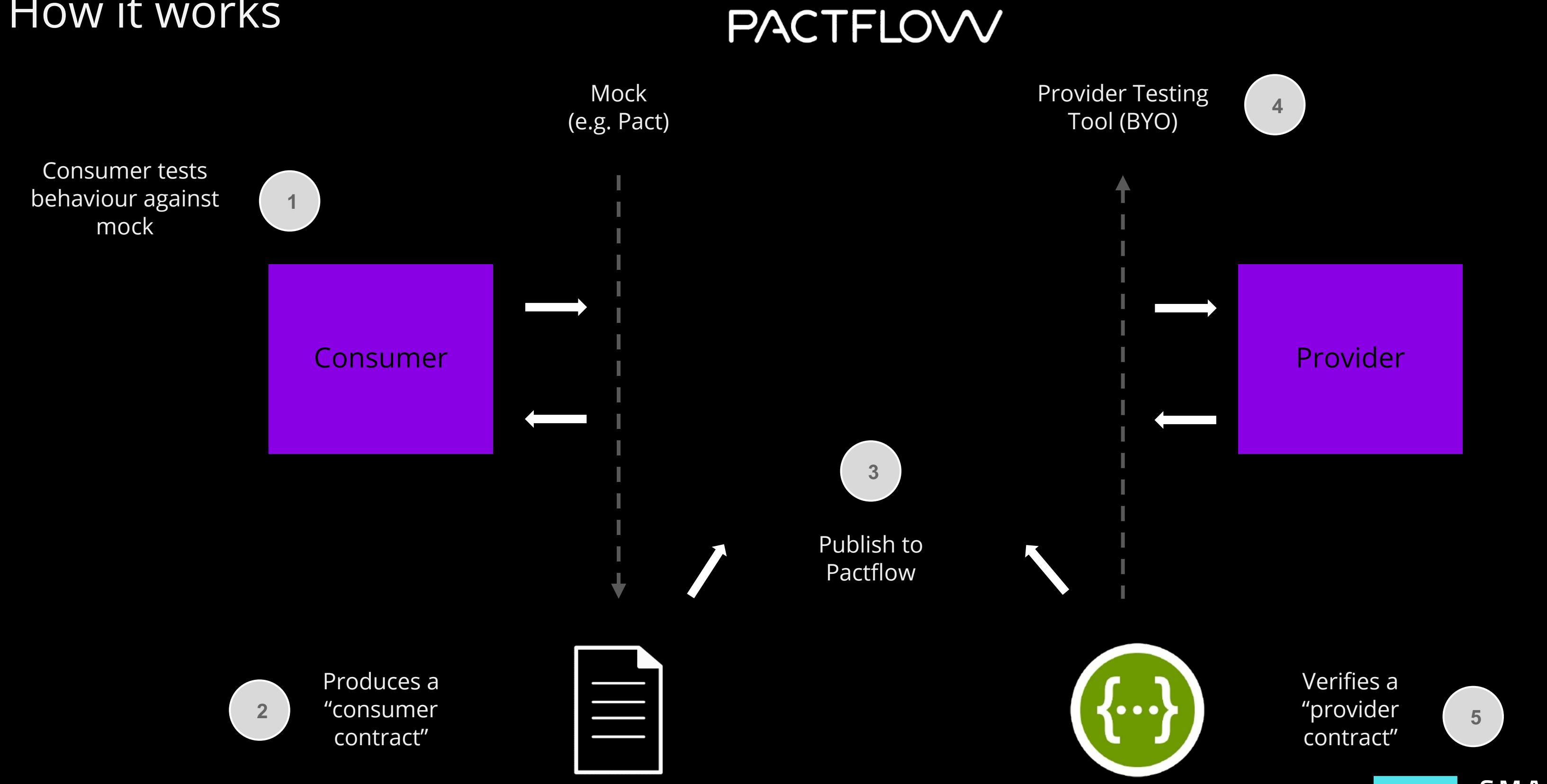
When contract-testing with Pact, you need to write and maintain a separate set of tests that are responsible for ensuring systems are compatible.

Unlike Pact, Bi-directional contracts allows teams to generate a contract from existing mocks (such as Wiremock), and to verify API providers using the functional API testing tools they are already using (such as Postman). Teams can use our plug-and-play adapters for popular tools or write their own.

All of the usual PactFlow collaboration tools and benefits apply, including the use of tools such as `can-i-deploy`.

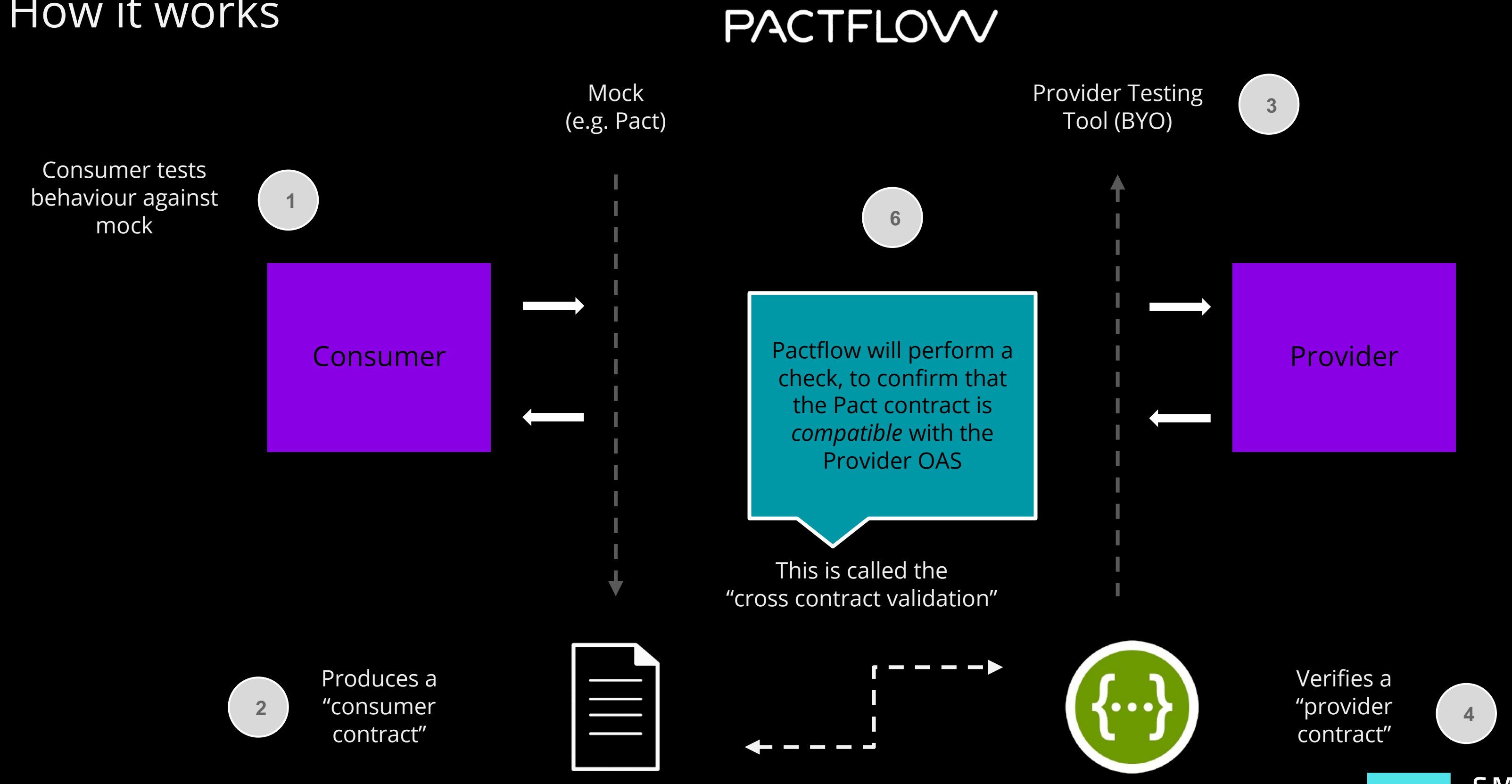
With bi-directional contracts, you can "upgrade" your existing tools into a powerful contract-testing solution, simplifying adoption and rapidly improving time-to-value and ROI.

How it works



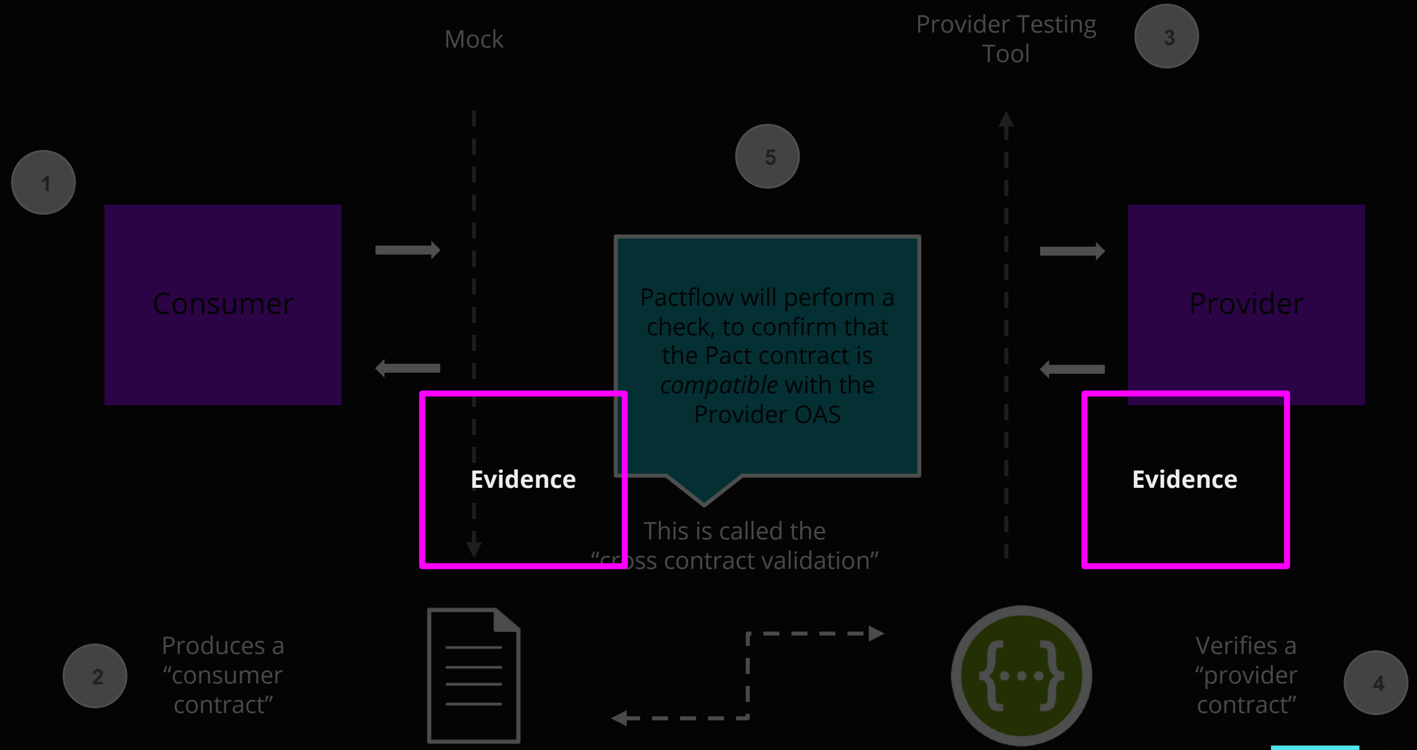
SMARTBEAR PactFlow

How it works



SMARTBEAR PactFlow

How it works


Mock
Provider Testing Tool
3
5
1
Consumer
Pactflow will perform a check, to confirm that the Pact contract is *compatible* with the Provider OAS
Provider
Evidence
Evidence
This is called the "cross contract validation"
2
Produces a "consumer contract"
Verifies a "provider contract"
4

SMARTBEAR PactFlow

PROBLEM STATEMENT

(why bi-directional contracts?)

SMARTBEAR PactFlow

why try a different approach?

Non-technical reasons:

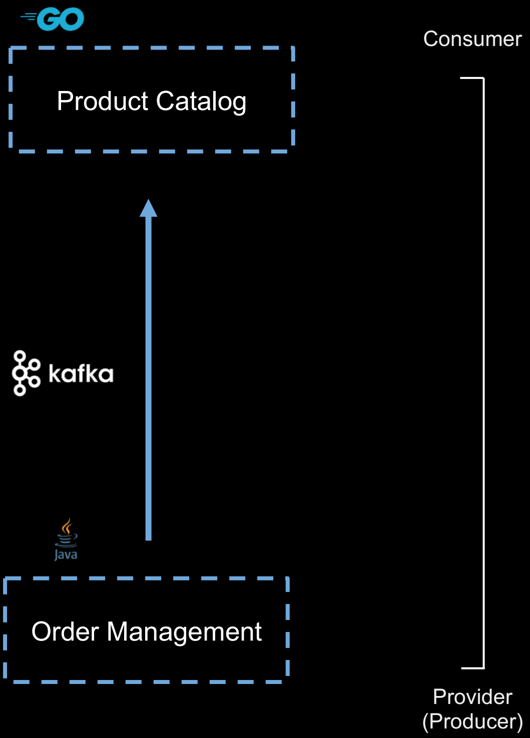
1. **Steep learning curve -** Education often required to get the most of Pact
2. **Technical investment required** - Pact requires both parties of an integration point to write and maintain tests
3. **Developer only** - Pact requires access to the source code, excluding some roles from participating
4. **Suitability for API first design workflows** - many organisations have a provider-first workflow
5. **Convincing people -** there are a number of excuses!

Technical reasons:

1. **Applicability to certain architectures / classes of problems** - Pact is not ideally suited to working with API gateways, 3rd party APIs or APIs with large numbers of consumers.
2. **UI testing** - Creating pacts from UI tests can lead to pain if not done carefully

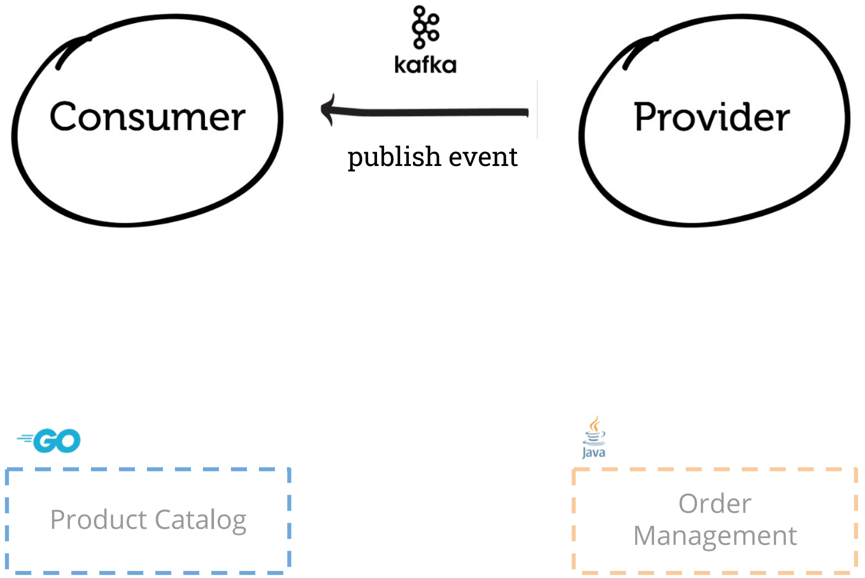
HOW PACT WORKS

(Async/Messages)

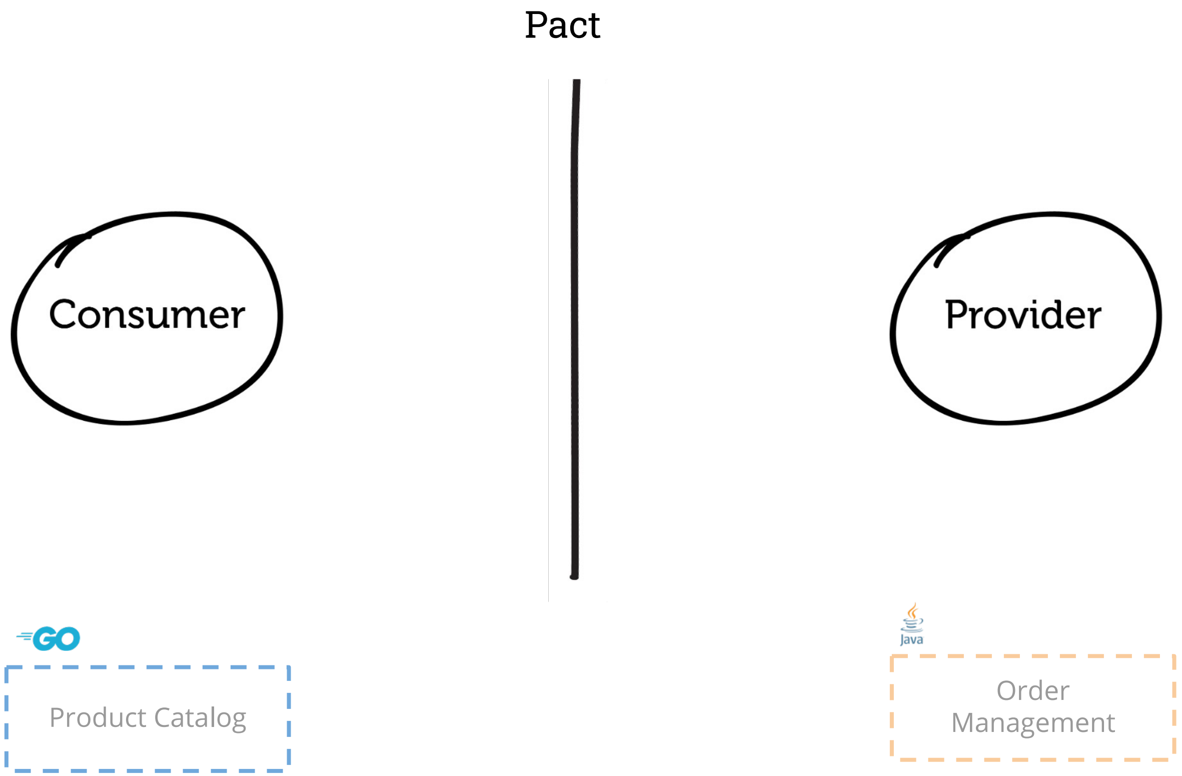


Contract

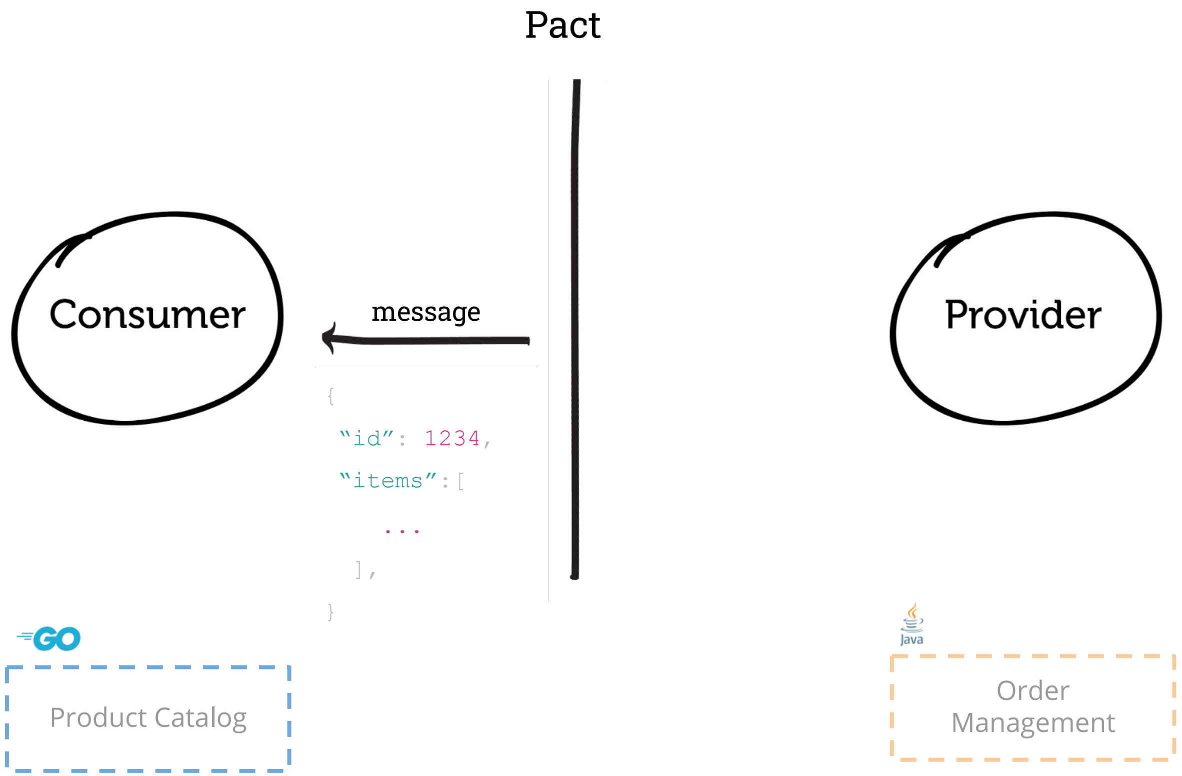
```
topic: products
content-type: application/json

{
 "id": 1234,
 "items":[
     ...
   ],
}
```


kafka
Consumer
publish event
Provider
GO
Product Catalog
Java
Order Management

Step 1: test the consumer (expects to receive…)

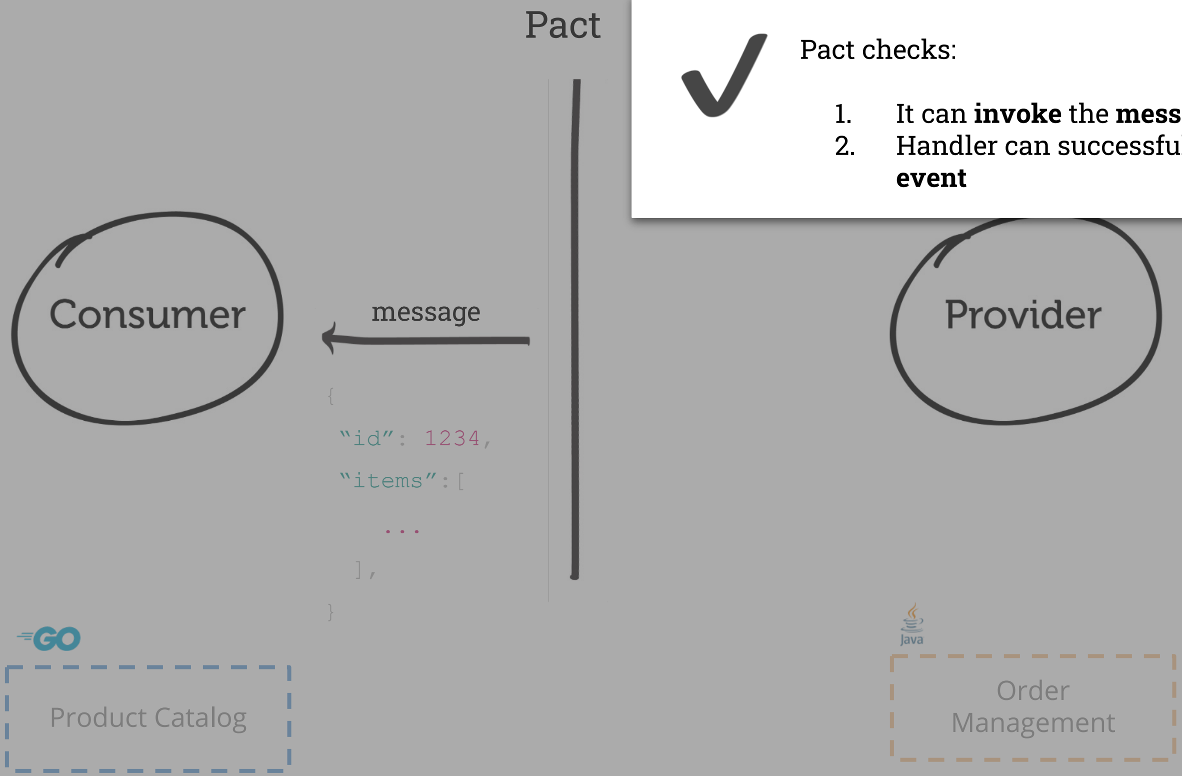


Step 1: test the consumer (expects to receive…)



```
{
 "id": 1234,
 "items":[
     ...
   ],
}
```

Step 1: test the consumer (expects to receive...)

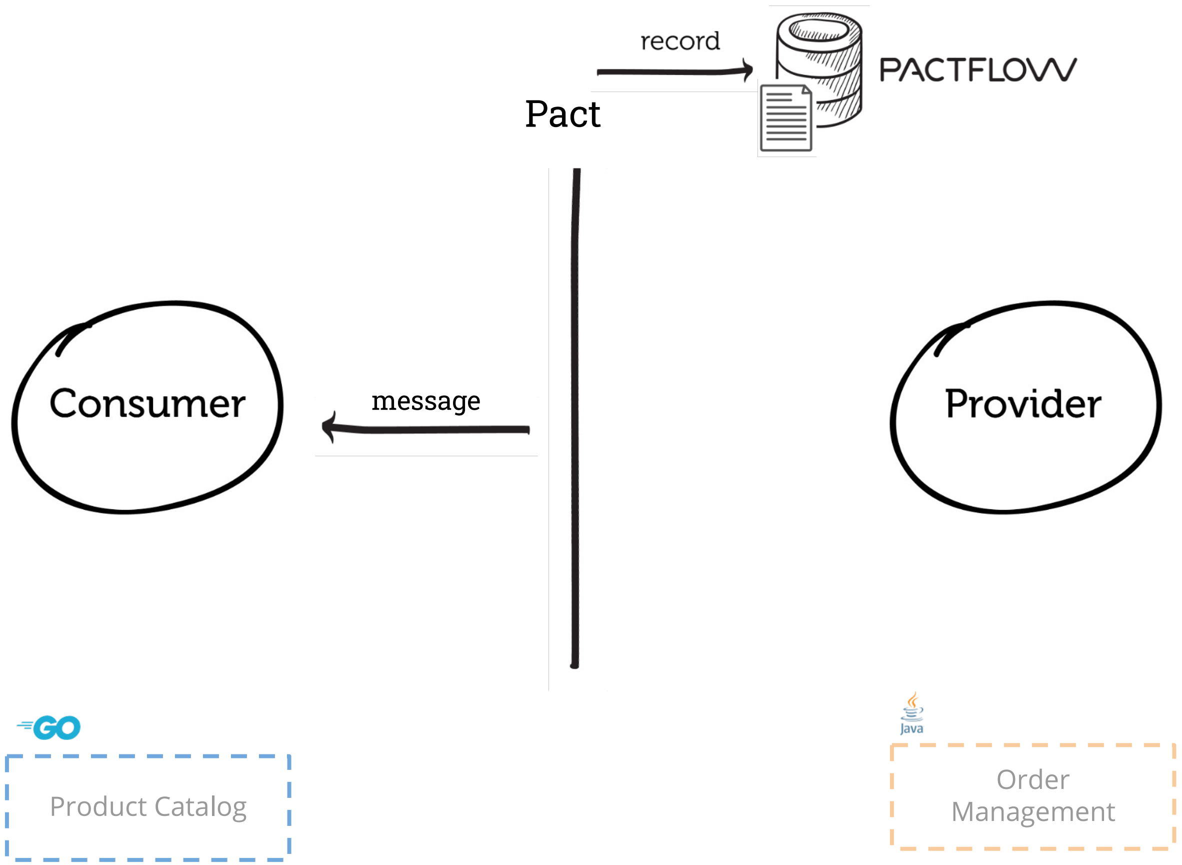


```
{
 "id": 1234,
 "items": [
    ...
  ],
}
```

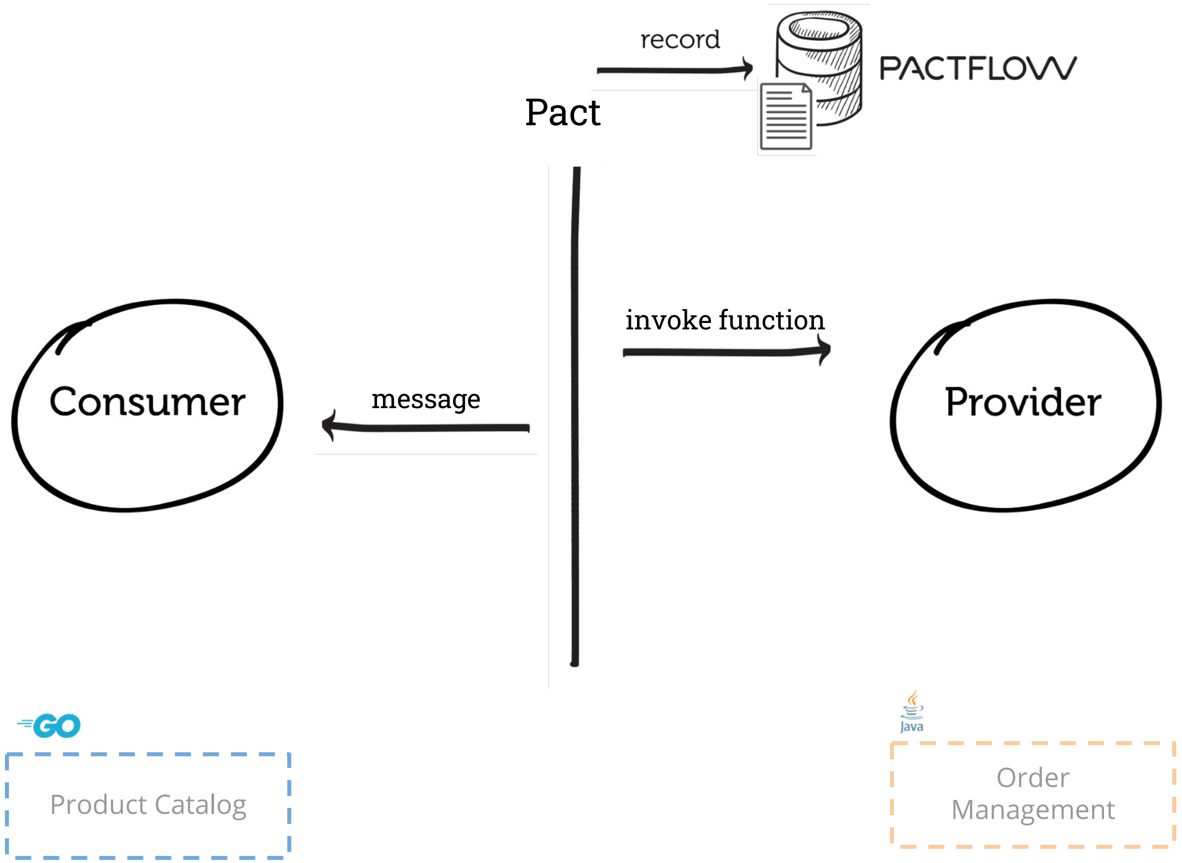
✓ Pact checks:

1. It can **invoke** the **message handler**
2. Handler can successfully **process event**

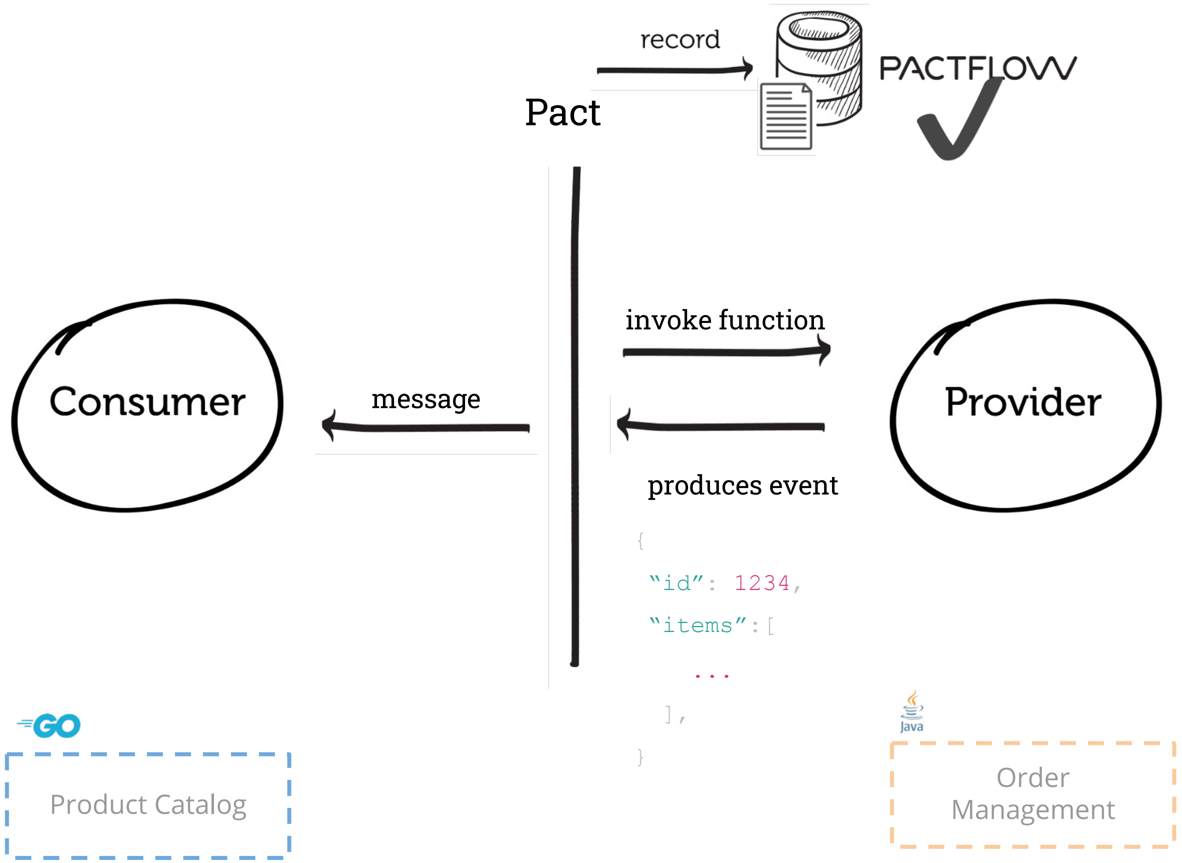
Step 2: share the contract with the Pactflow



Step 3: test the provider



Step 3: test the provider

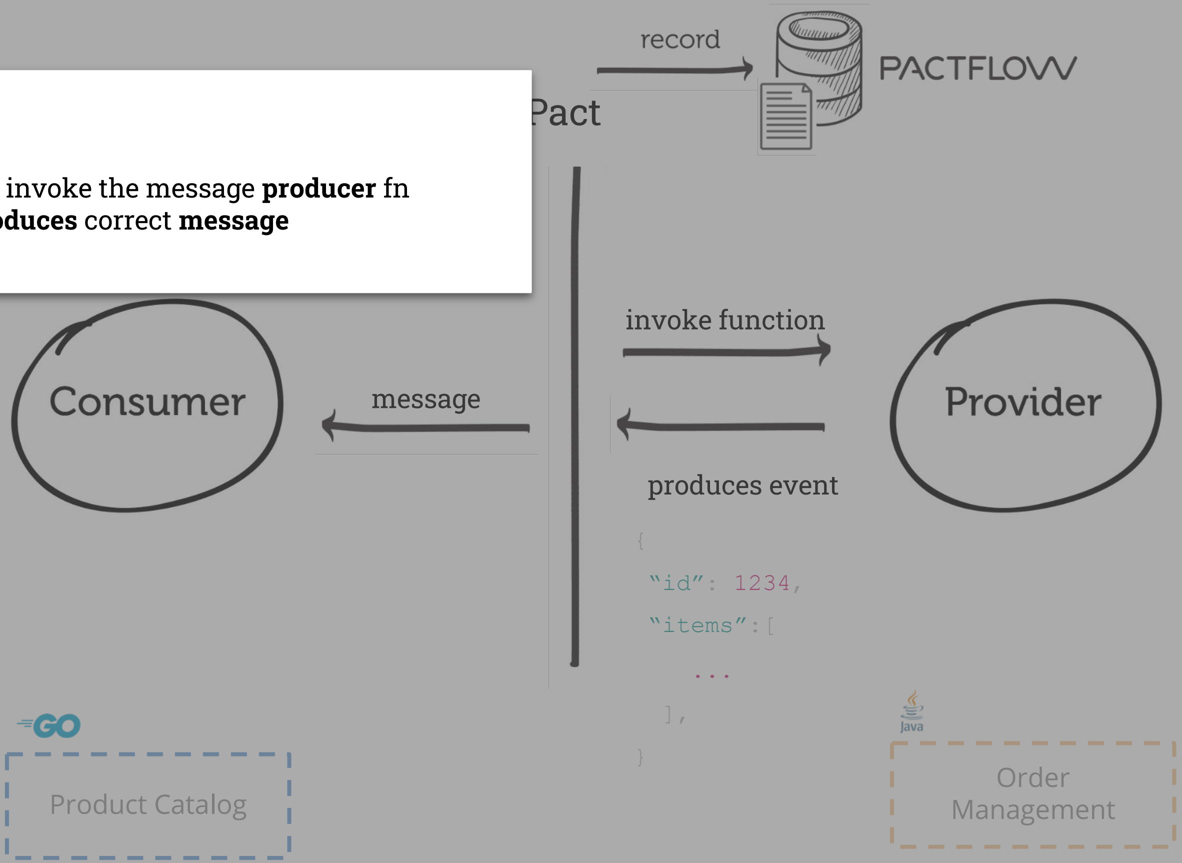


```
{
 "id": 1234,
 "items":[
     ...
   ],
}
```

Step 3: test the provider

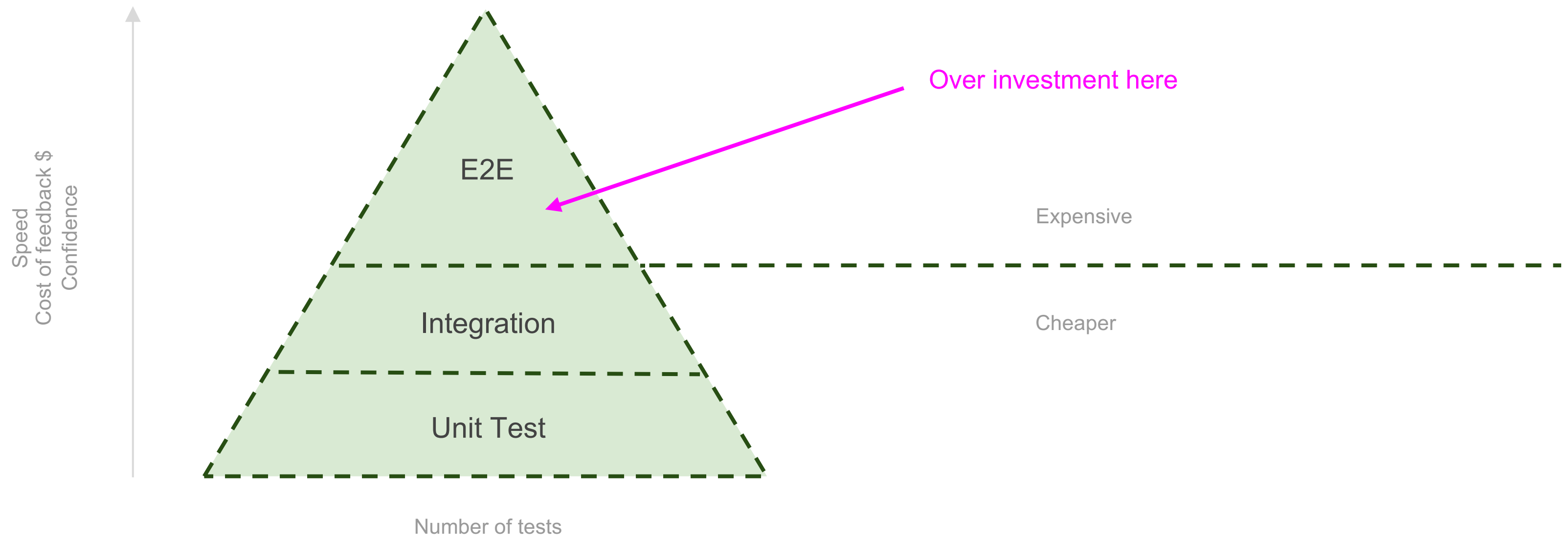
✓ Pact checks:

1. It can invoke the message **producer** fn
2. fn **produces** correct **message**

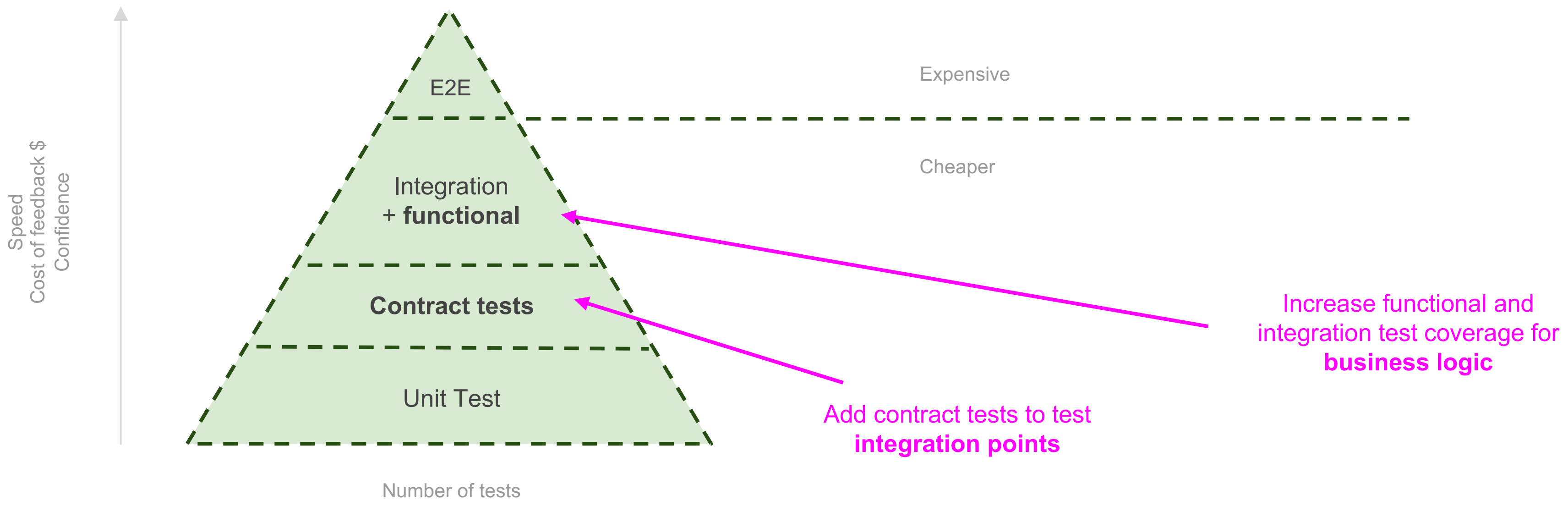


```
{
 "id": 1234,
 "items":[
    ...
  ],
}
```

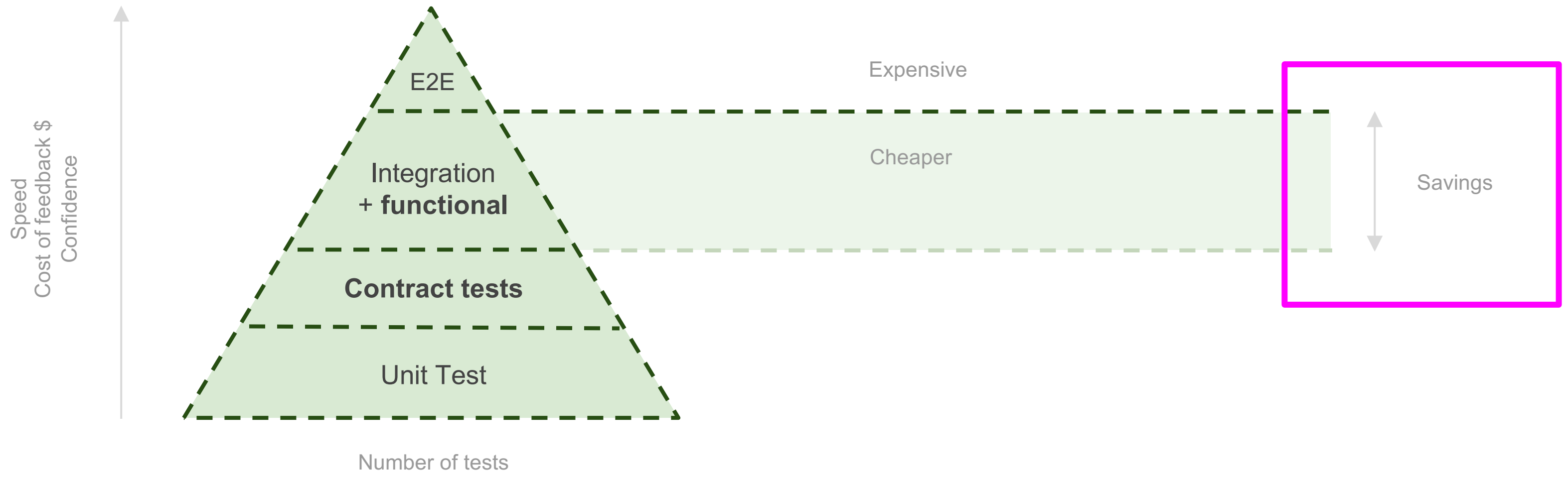
step 1: review test pyramid



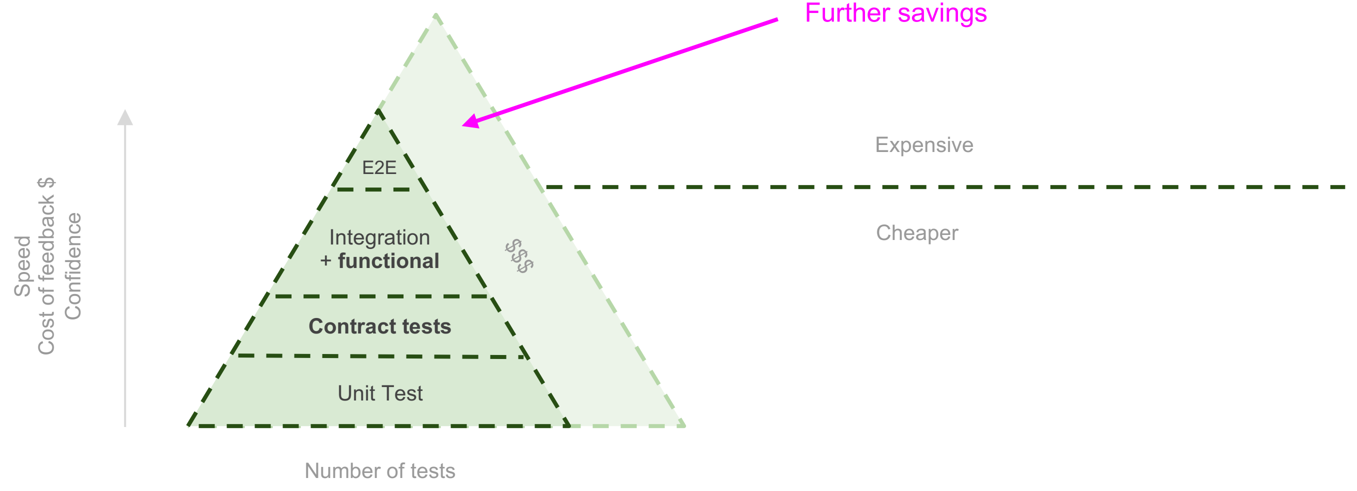
step 2: rebalance test pyramid


Speed
Cost of feedback $
Confidence
E2E
Integration
+ **functional**
Contract tests
Unit Test
Number of tests
Expensive
Cheaper
Increase functional and integration test coverage for **business logic**
Add contract tests to test **integration points**

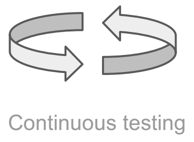
step 2: rebalance test pyramid



step 3: shrink the pyramid



step 4: continuous testing and monitoring

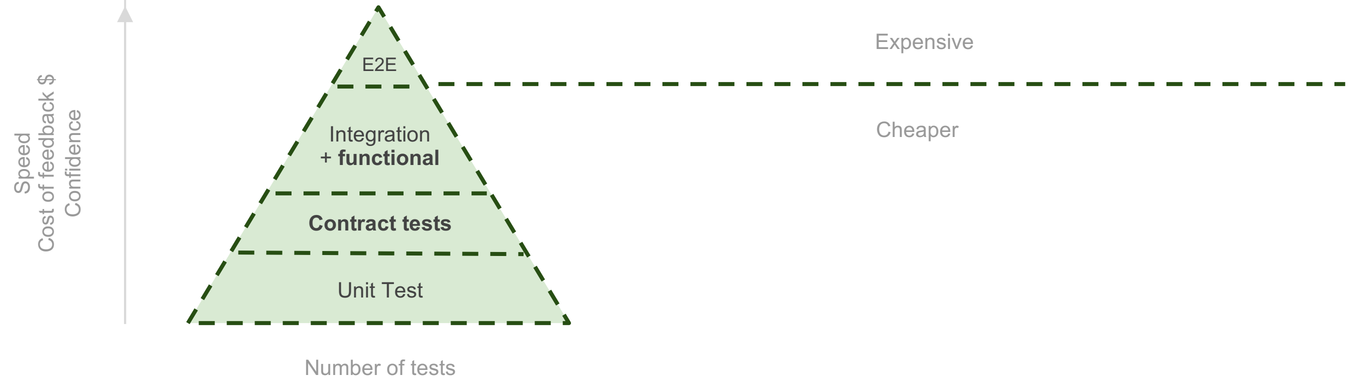


- Canaries, smoke tests and automated rollbacks
- Semantic monitoring / synthetic transactions
- Improved telemetry and observability
- Aggregated logging and access for the team
- Tune monitoring & alerting

Investment here

Post deployment

Before deployment



WORKSHOP

(https://github.com/pact-foundation/pact-workshop-js)

REMINDER

WORKSHOP

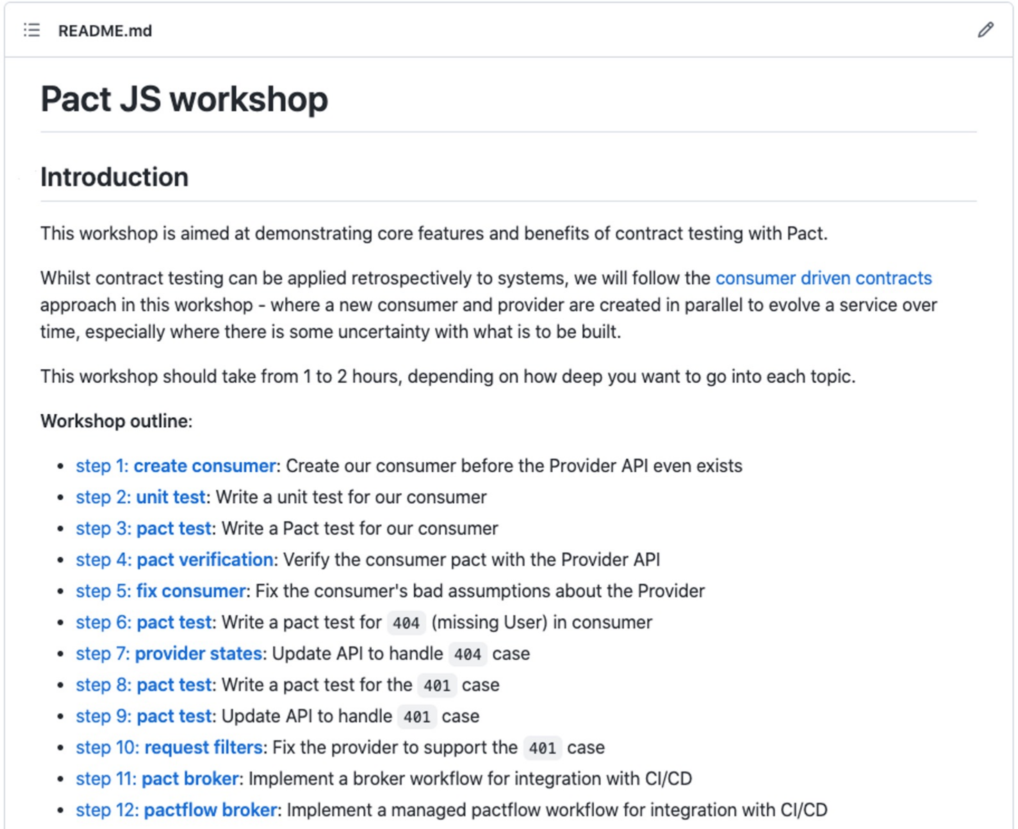
(https://github.com/pact-foundation/pact-workshop-js)

Participating

Getting the most out of the workshop

- Workshop arranged as a series of steps, each in a separate branch
- We will progress each step as a group, but you are encouraged to explore as we go
- Q&A will be available at the end of each step
- Each step has specific learning objectives

✨ **Follow the README!**



README.md

Pact JS workshop

Introduction

This workshop is aimed at demonstrating core features and benefits of contract testing with Pact.

Whilst contract testing can be applied retrospectively to systems, we will follow the consumer driven contracts approach in this workshop - where a new consumer and provider are created in parallel to evolve a service over time, especially where there is some uncertainty with what is to be built.

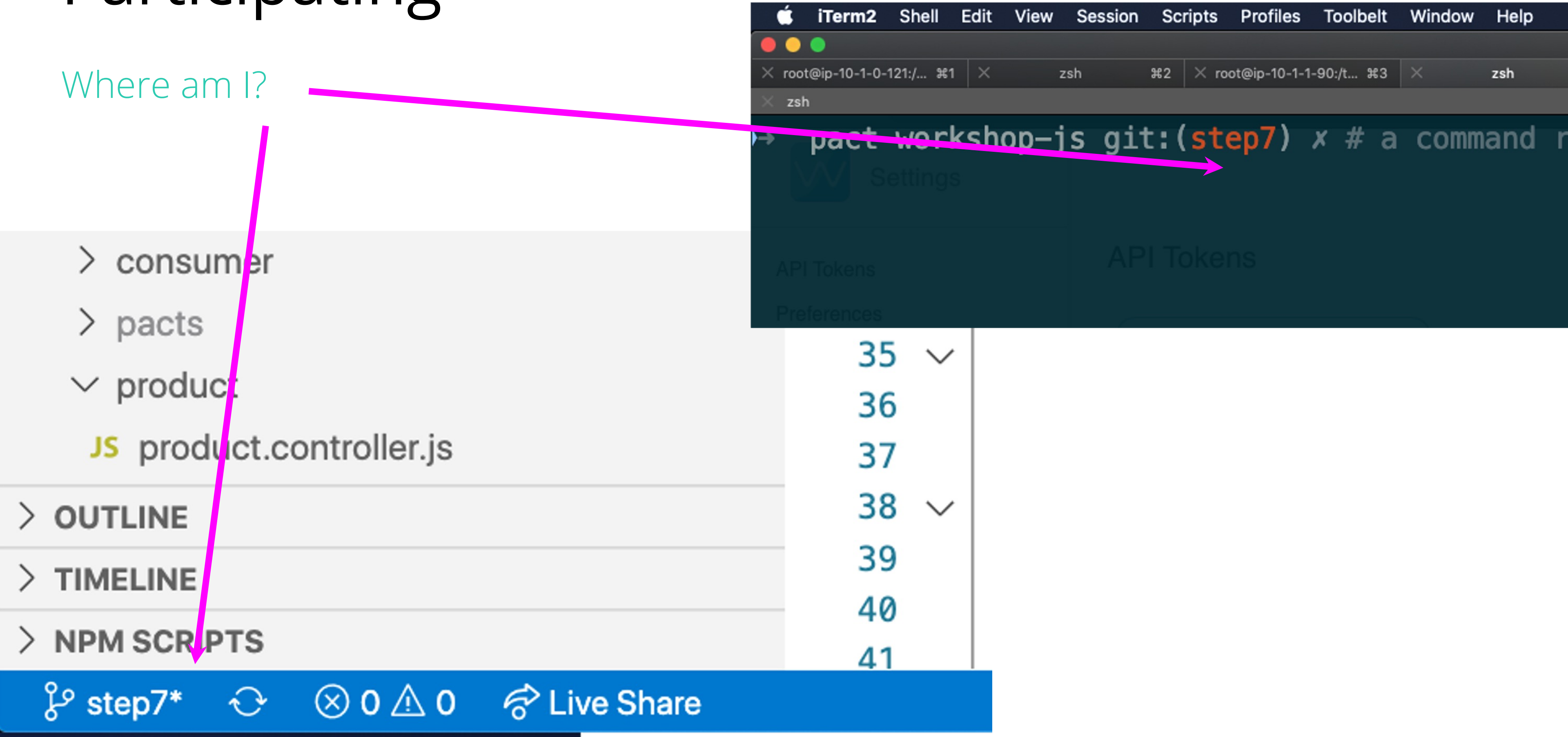
This workshop should take from 1 to 2 hours, depending on how deep you want to go into each topic.

Workshop outline:

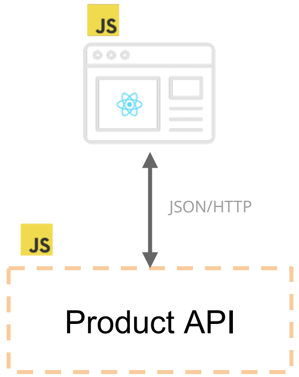
- step 1: **create consumer**: Create our consumer before the Provider API even exists
- step 2: **unit test**: Write a unit test for our consumer
- step 3: **pact test**: Write a Pact test for our consumer
- step 4: **pact verification**: Verify the consumer pact with the Provider API
- step 5: **fix consumer**: Fix the consumer's bad assumptions about the Provider
- step 6: **pact test**: Write a pact test for `404` (missing User) in consumer
- step 7: **provider states**: Update API to handle `404` case
- step 8: **pact test**: Write a pact test for the `401` case
- step 9: **pact test**: Update API to handle `401` case
- step 10: **request filters**: Fix the provider to support the `401` case
- step 11: **pact broker**: Implement a broker workflow for integration with CI/CD
- step 12: **pactflow broker**: Implement a managed pactflow workflow for integration with CI/CD

Participating

Where am I?

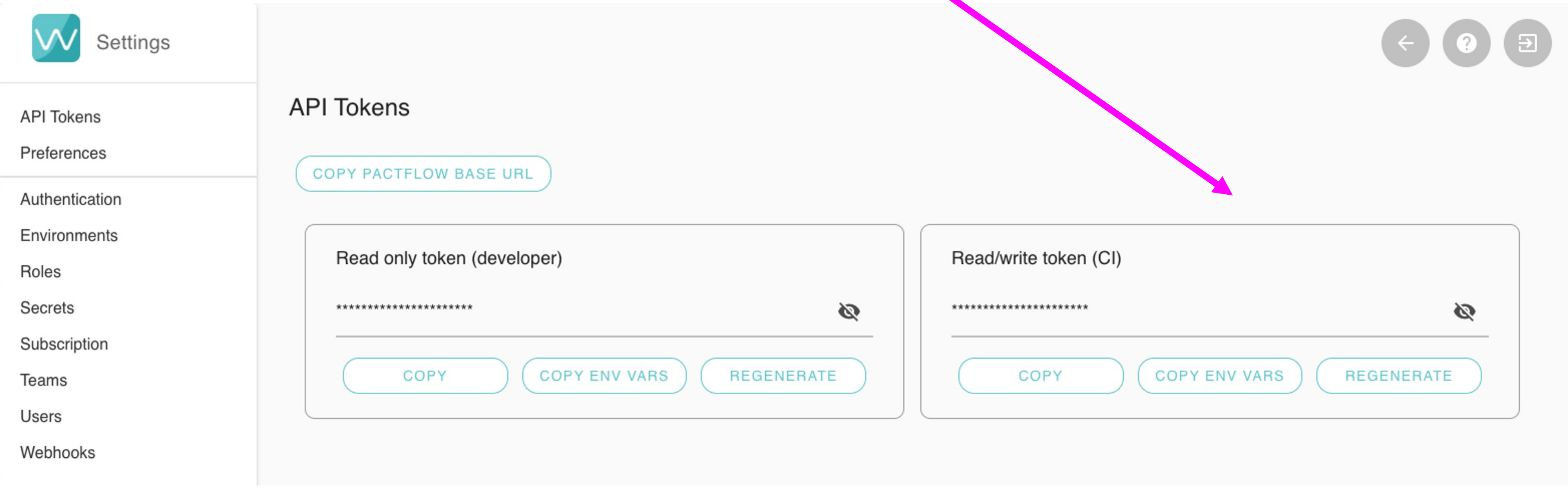


```
→ pact workshop-js git:(step7) ✗ # a command r
```


JS
JSON/HTTP
JS
Product API

Fetching your API Token (step 12)

Settings > API Tokens > read/write token



Settings

API Tokens
Preferences
Authentication
Environments
Roles
Secrets
Subscription
Teams
Users
Webhooks

API Tokens

COPY PACTFLOW BASE URL

Read only token (developer)

****

COPY | COPY ENV VARS | REGENERATE

Read/write token (CI)

****

COPY | COPY ENV VARS | REGENERATE

https://docs.pactflow.io/#configuring-your-api-token