Beyond REST

Contract testing in the age of gRPC, Kafka and GraphQL.

Matt Fellows

YOW! Perth, Sep ‘22

Presented by

Matt Fellows - Principal Product Manager @ SmartBear

– Co-Founder, Pactflow

– Core Maintainer, Pact (Go, JS)

@matthewfellows

Consumer-Driven Contracts: A Service Evolution Pattern

This article discusses some of the challenges in evolving a community of service providers and consumers. It describes some of the coupling issues that arise when service providers change parts of their contract, particularly document schemas, and identifies two well-understood strategies - adding schema extension points and performing "just enough" validation of received messages - for mitigating such issues. Both strategies help protect consumers from changes to a provider contract, but neither of them gives the provider any insight into the ways it is being used and the obligations it must maintain as it evolves. Drawing on the assertion-based language of one of these mitigation strategies - the "just enough" validation strategy - the article then describes the "Consumer-Driven Contract" pattern, which imbues providers with insight into their consumer obligations, and focuses service evolution around the delivery of the key business functionality demanded by consumers.

12 June 2006

Ian Robinson

CONTENTS

https://martinfowler.com/articles/consumerDrivenContracts.html#InterludeBurdenedWithServices

PACTFLOW
A SMARTBEAR COMPANY

If I got a penny every time...

(a story)

“If we just used <*insert some new tech*>
then we wouldn’t need contract testing”

Modern architecture

The challenges facing today’s engineering leaders

PACTFLOW
A SMARTBEAR COMPANY

Customer quote

We have a very large program with many different scrum teams building a wide variety of components all operating in a microservices event based architecture.

Testing inside a highly volatile set of integrated environments is extremely challenging today.

Looking to get better confidence by doing better isolated contract testing...

Between direct calls to RESTful or GraphQL APIs, or messages using AWS event bridge or Kafka, and also 3rd party SaaS and partner integrations...it's difficult to manage.

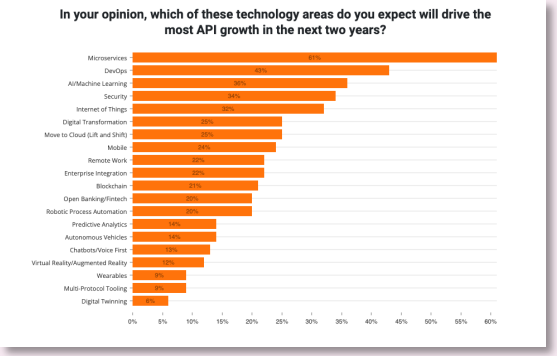
a large banking prospect

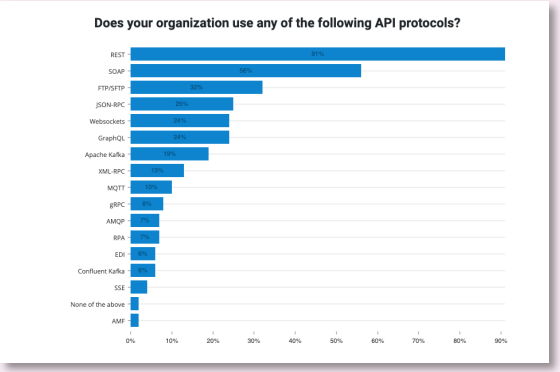
PACTFLOW
A SMARTBEAR COMPANY

Industry insights

More microservices, more protocols

1. 61% say most API growth from microservices
2. 81% of companies operate in a multi-protocol environment
3. 57% use 3 or more protocols





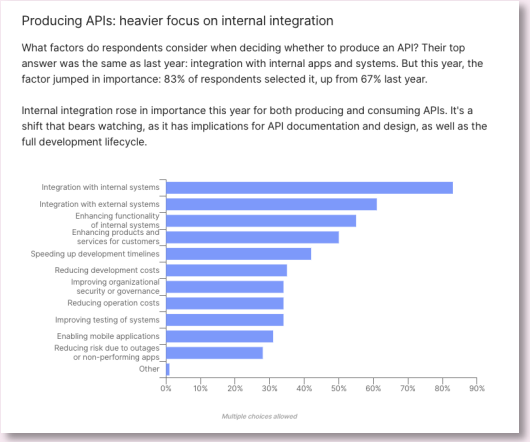
https://smartbear.com/state-of-software-quality/api/

PACTFLOW
A SMARTBEAR COMPANY

Industry insights

More microservices, more protocols

1. Internal integration is the new focus
2. = open up new use cases / data to the org



https://www.postman.com/state-of-api/

PACTFLOW
A SMARTBEAR COMPANY

Industry insights

The headwinds of *"Microservices sprawl"*

Barriers to implementing microservices:

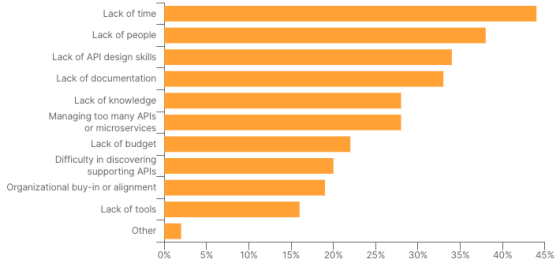
1. Experience or skills
2. Complexity of systems
3. Increasing demands on speed of delivery, and
4. Limited time due to workload

Mature organisations feeling the pain

Obstacles to producing APIs: lack of design skills

Lack of time was again organizations' biggest obstacle to producing APIs, followed by lack of people. But the third-biggest hindrance was new this year: lack of API design skills.

A gap in API design skills may be contributing to an overproliferation of microservices, which is a problem in itself. Managing too many APIs or microservices was respondents' sixth biggest obstacle to producing APIs. Among API-first leaders, it's an even bigger problem: too many microservices was their second-biggest obstacle.



Multiple choices allowed

"By 2025, less than 50% of enterprise APIs will be managed, as explosive growth in APIs surpasses the capabilities of API management tools."

Gartner

https://smartbear.com/state-of-software-quality/api/

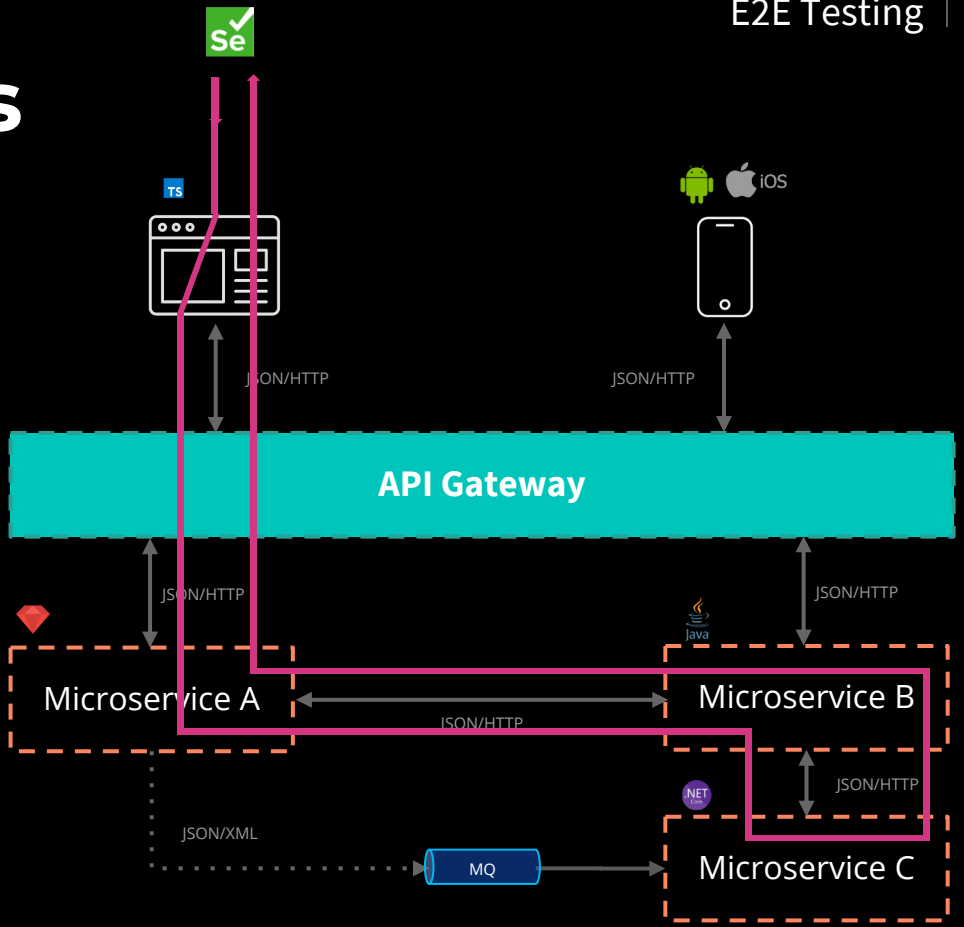
How we test microservices now

And why it doesn’t scale

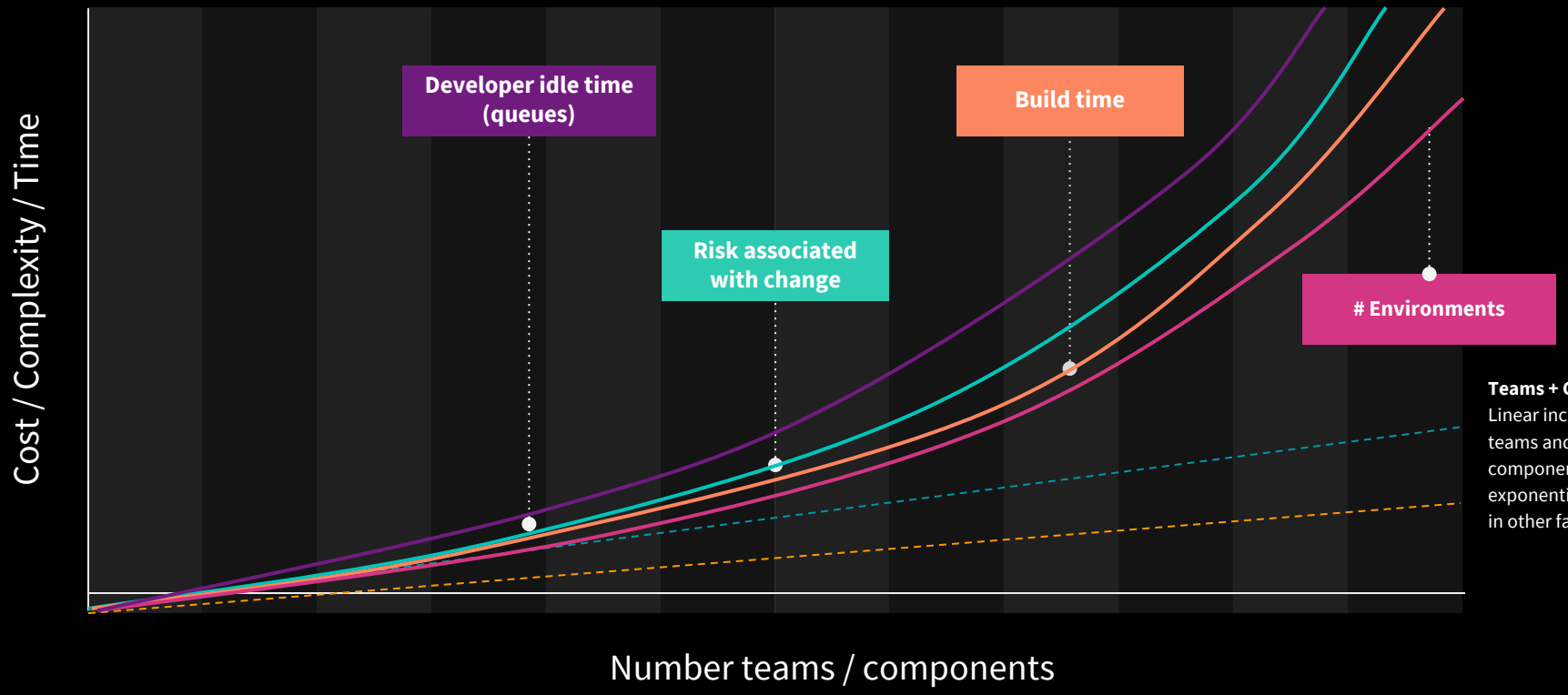
End-to-end tests

Why this is hard

- Slow
- Fragile
- Hard to debug
- All-at-once painful deployments
- Teams wait on build queues

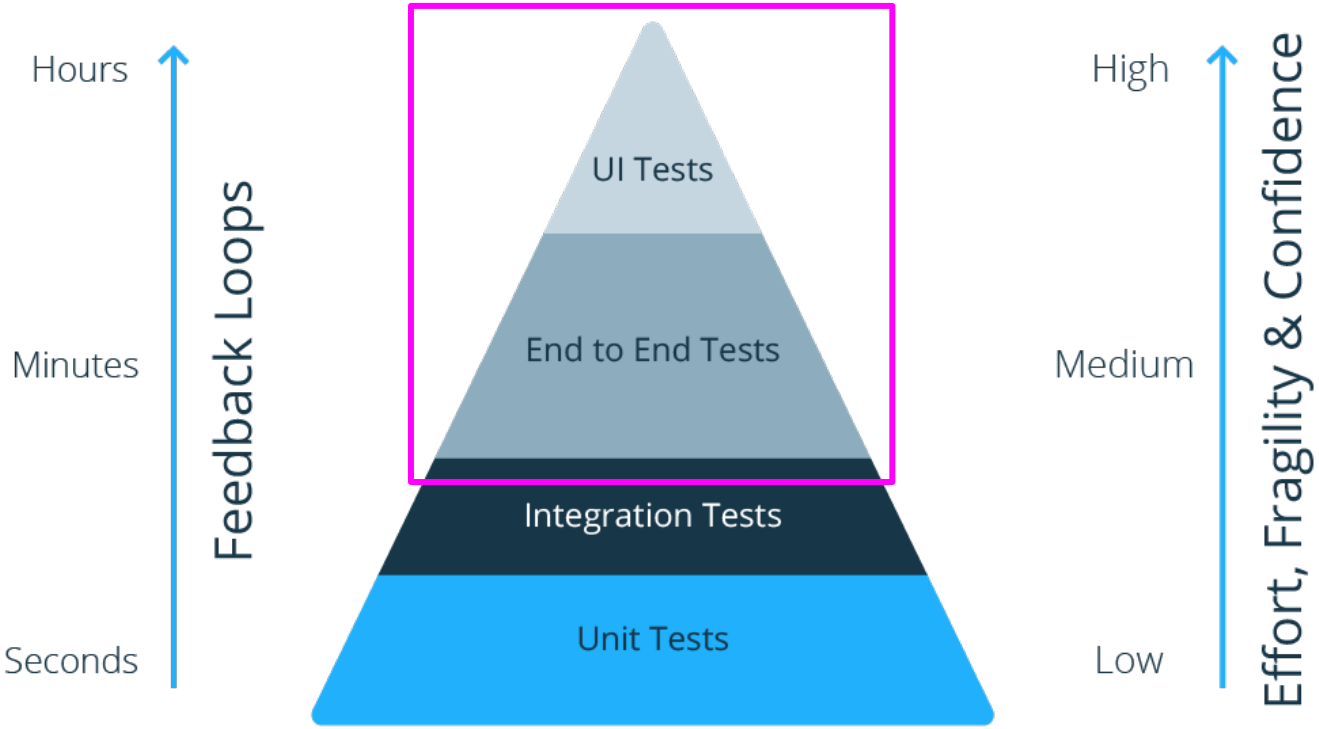


Scaling Challenges



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

Unbalanced test pyramid



Read the blog

The solution?

Consumer Driven Contract testing

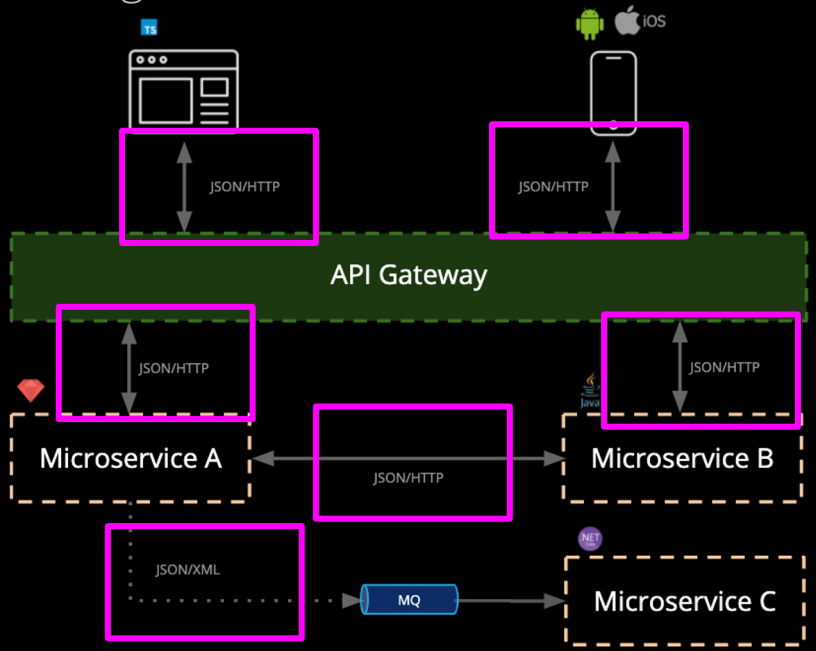
What is Contract Testing?

An alternative approach to API communication testing

Benefits:

- **Simpler** - test a single integration at a time
- No **dedicated test environments** - run on a dev machine
- Get **fast**, reliable feedback
- Tests that scale **linearly**
- **Deploy** services independently

It tracks these over time, enabling **evolution**



PACTFLOW
A SMARTBEAR COMPANY

How it works

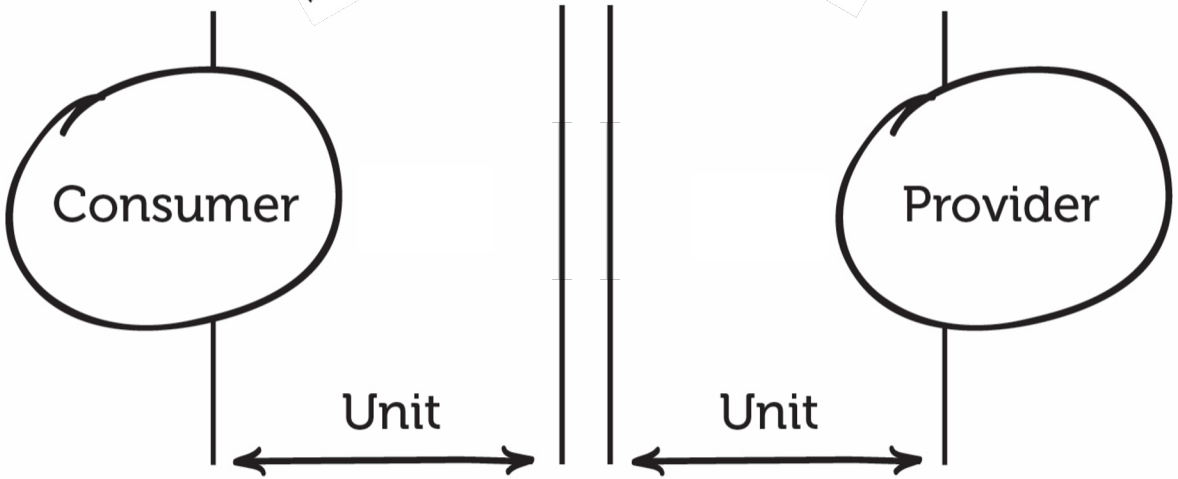
3 The contract is **shared** amongst teams to enable collaboration

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

2

4

Interactions in the 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**

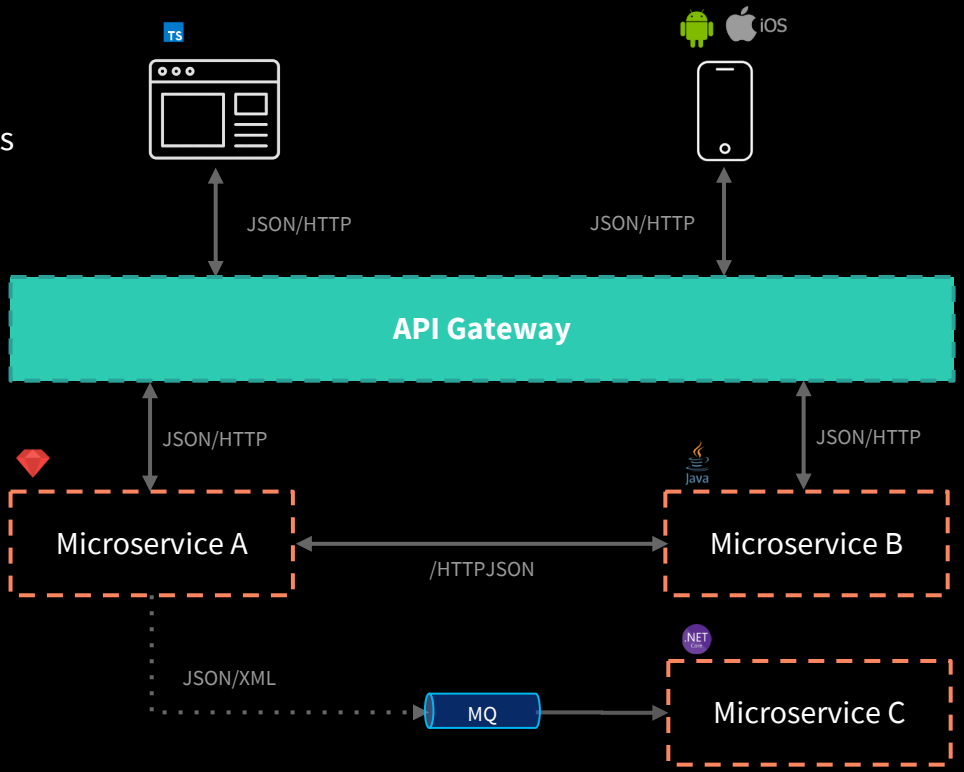
Pact

Microservice contract testing

Contract testing 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)
- And much more!



What is Contract Testing?

Ingredients of a consumer Pact test

```
describe('GET /orders', () => {
  before(() => {
    const interaction = new Interaction()
      given('there are orders to be fulfilled')
      upon_receiving('a request to get all open orders')
      .withRequest({
        method: 'GET',
        path: '/orders',
      })
      .willRespondWith({
        status: 200,
        headers: {
          'Content-Type': 'application/json',
        },
        body: [{
          id: 1234,
          total: 9900,
          items: [...]
        }],
      });

    return provider.addInteraction(interaction);
  });

  ...
});
```

```
describe('GET /orders', () => {
  ...
  it('returns all open orders', async () => {
    // Act
    const orders = await orderService.getOpenOrders();

    // Assert
    expect(orders[0].id).to.eq(1234)
  });
});
```

What is Contract Testing?

Outputs of a consumer Pact test

If the client doesn’t call the endpoint as expected, the test fails.

It’s a **mock** not a **stub.**

```
GET /orders
  × returns all open orders (2 ms)

● GET /orders › returns all open orders

  Test failed for the following reasons:

    Mock server failed with the following mismatches:

      0) The following request was expected but not received:
          Method: GET
          Path: /orders
          Headers:
            Accept: application/json

    at PactV3.<anonymous> (node_modules/@pact-foundation/src/v3/pact.ts:227:29)
    at step (node_modules/@pact-foundation/pact/src/v3/pact.js:33:23)
    at Object.next (node_modules/@pact-foundation/pact/src/v3/pact.js:14:53)
    at fulfilled (node_modules/@pact-foundation/pact/src/v3/pact.js:5:58)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        1.465 s, estimated 2 s
```

What is Contract Testing?

Outputs of a consumer Pact test

If the test passes, we get a contract containing the expectations from this consumer for a given API provider.

```
{
  "consumer": { "name": "Order Client" },
  "provider": { "name": "Order API" },
  "interactions": [
    {
      "description": "a request to get all open orders",
      "providerState": "there are orders to be fulfilled",
      "request": {
        "method": "GET",
        "path": "/orders"
      },
      "response": {
        "body": [...]
        "headers": {
          "Content-Type": "application/json"
        },
        "matchingRules": { ... },
        "status": 200
      }
    },
    {.. }
    }
  ],
  "metadata": {...}
}
```

PACTFLOW
A SMARTBEAR COMPANY

Collaborating
Pact Broker

What's New

Can I Deploy

API

OVERVIEW | NETWORK DIAGRAM | MATRIX | WEBHOOKS | CONTRACTS

A pact between Product Website and Product API

Consumer Details

CONSUMER VERSION
28669a49080d16cfb62291c1a10d55b431d61474

More consumer details

PUBLISHED AT
9 minutes ago

BRANCH
main

RELEASED ENVIRONMENTS
N/A

DEPLOYED ENVIRONMENTS
Production

TAGS
N/A

Provider Details

PROVIDER VERSION
ac1ca19d725736782d63f30f1041584f86cad037

More provider details

PUBLISHED AT
4 days ago

BRANCH
master

RELEASED ENVIRONMENTS
N/A

DEPLOYED ENVIRONMENTS
Production

TAGS
N/A

CONTRAC... | CONSUMER CONTRACT | PROVIDER...

Consumer Contract

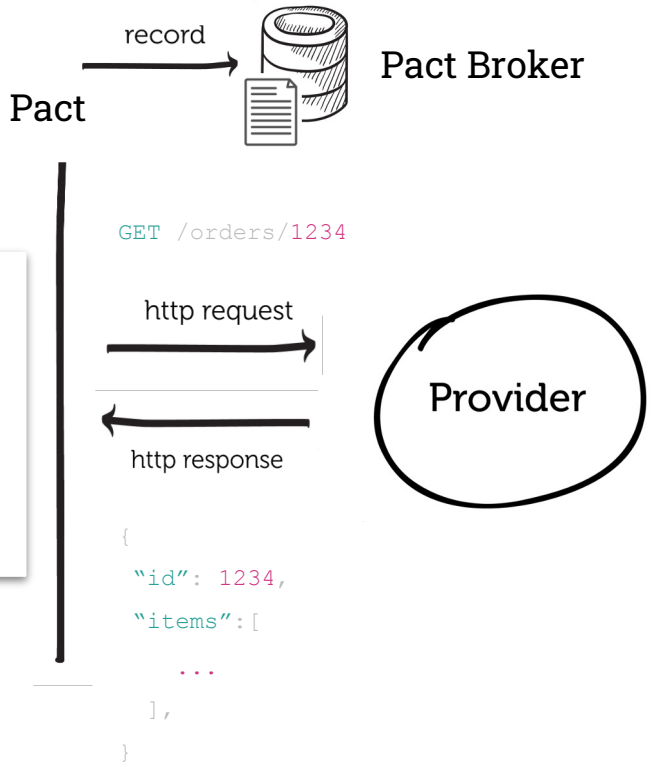
CONSUMER CONTRACT STATUS
Compatible

PACT SPEC VERSION
2.0.0

- Displays product item
- Displays product item by query
- Displays products

What is Contract Testing?

Provider contract test



✅ 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

PACTFLOW
A SMARTBEAR COMPANY

Pact Broker Evolution

What's New

Can I Deploy

API

OVERVIEW | NETWORK DIAGRAM | MATRIX | WEBHOOKS

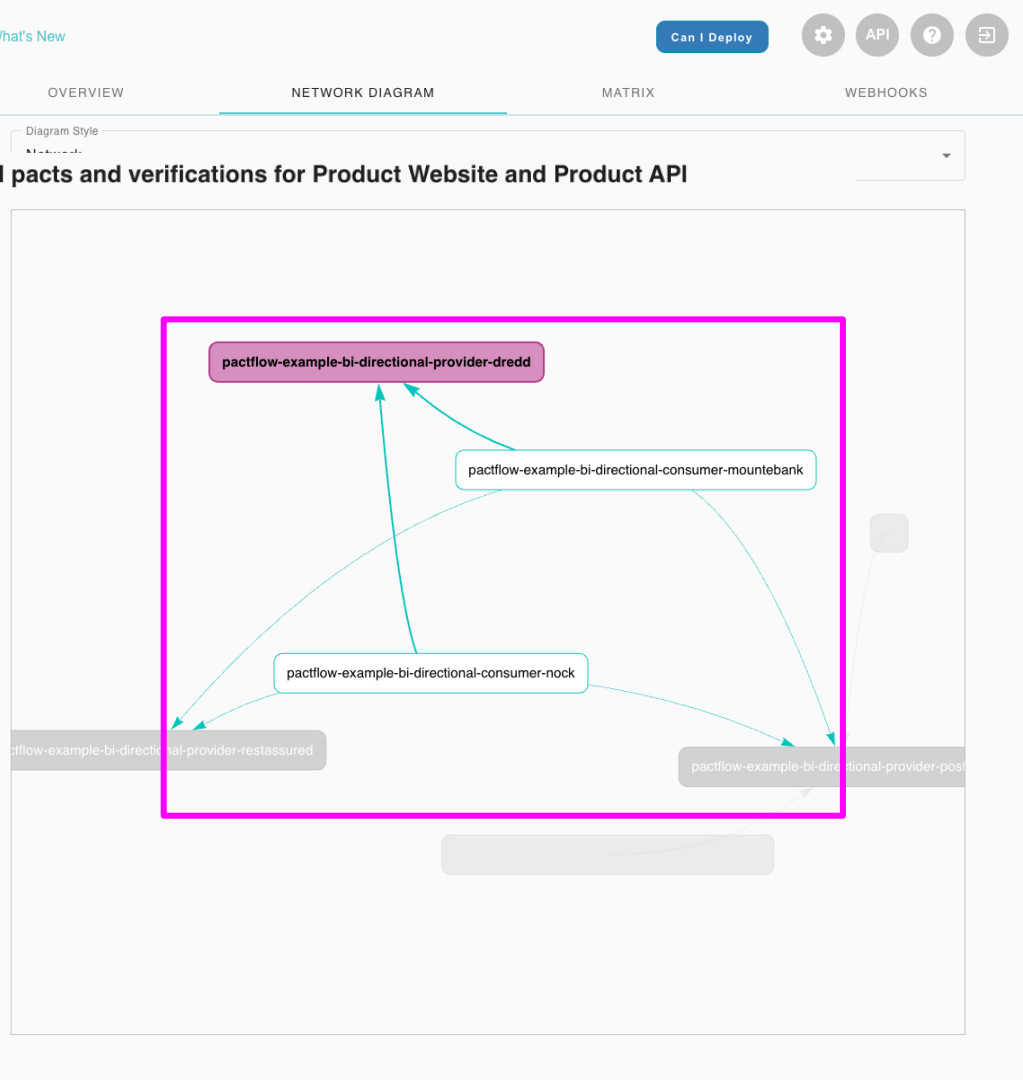
All pacts and verifications for Product Website and Product API

Count: 100 APPLY LIMIT

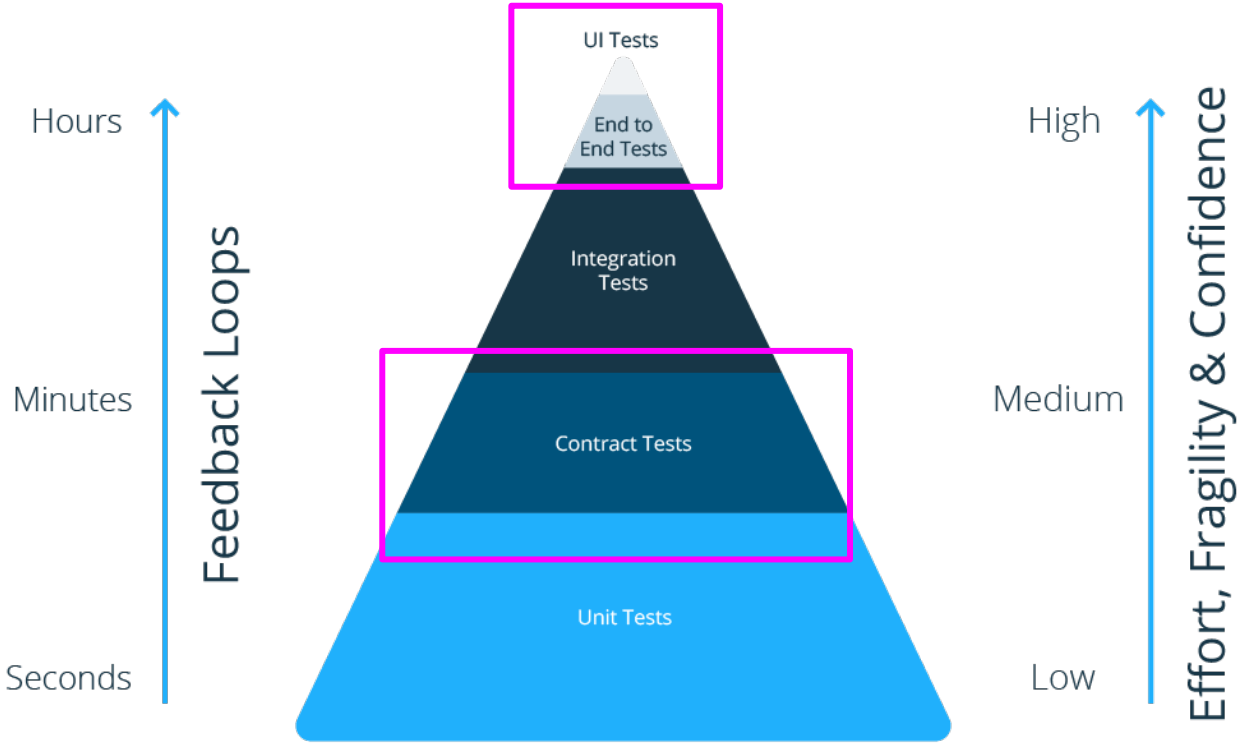
Consumer			Provider			Status
Version	Branch & Tags	Envs	Version	Branch & Tags	Envs	
24d838d Pact published 8 minutes ago	add_breaking_change	N/A	ac1ca19	master	Production	Verified 7 minutes ago
28669a4 Pact published 10 minutes ago	main	Production	ac1ca19	master	Production	Verified 10 minutes ago

PACTFLOW
A SMARTBEAR COMPANY

Pact Broker
Track dependencies



Rebalanced test pyramid



Read the blog

PACTFLOW
A SMARTBEAR COMPANY

New protocols to the rescue!

Can OpenAPI/AsyncAPI, gRPC, GraphQL or others dig us out of this hole?

If we just used

API Specifications

Then, we wouldn’t need contract testing

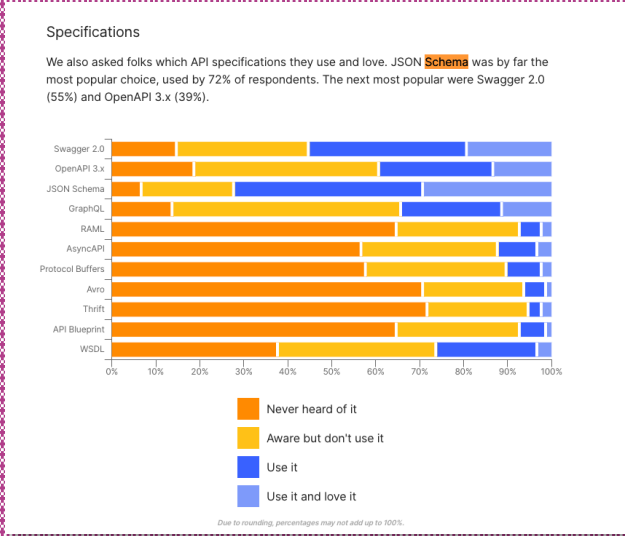
PACTFLOW
A SMARTBEAR COMPANY

OAS + JSON Schema

How it aims to solve the problem

1. Specifications contain all of the bits needed for humans and computers to communicate an API's functionality
2. It uses JSON Schema – we know what the shapes of the resources are allowed to be
3. We can generate API clients from OAS, so we know they won't have breaking changes in them

If we can generate client code from the OAS, aren't we guaranteed to have a working system?



PACTFLOW
A SMARTBEAR COMPANY

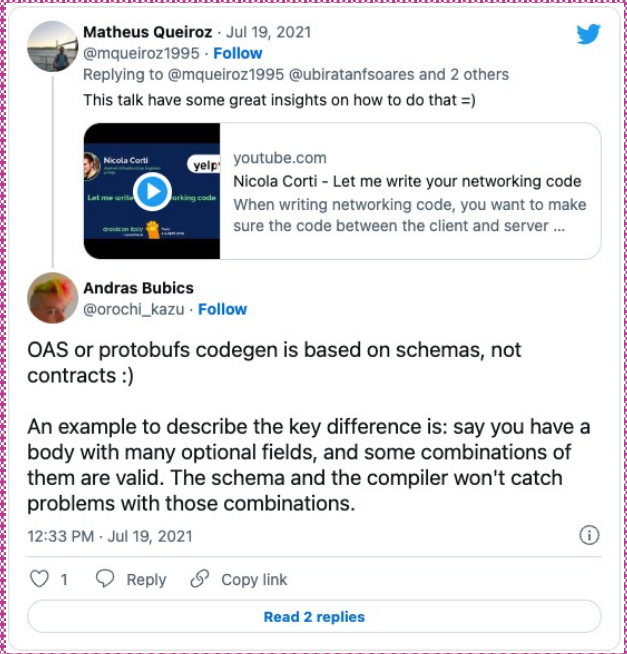
OAS + JSON Schema

Why it doesn't

1. Schemas are abstract – testing requires diligence to prove correctness
2. Loss of sight of API surface area required by consumers
3. A mechanism for evolution is needed
4. Client SDKs are often modified and can be used in unexpected ways in practice

> *Any "validation tool" for a sufficiently complex data format, therefore, will likely have two phases of validation: one at the schema (or structural) level, and one at the semantic level. The latter check will likely need to be implemented using a more general-purpose programming language*

- JSON Schema



PACTFLOW
A SMARTBEAR COMPANY

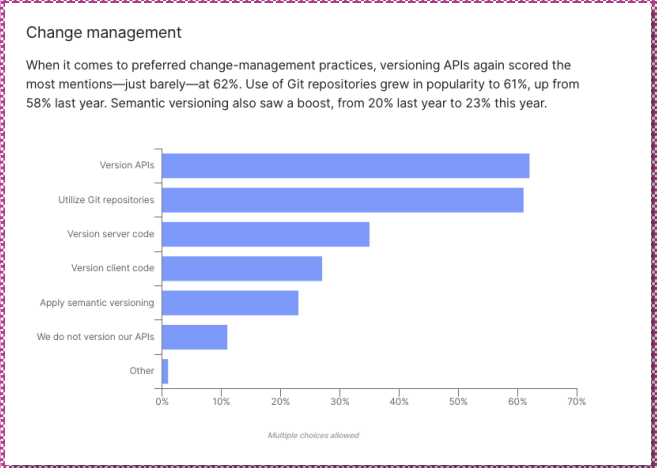
What about versioning?

API versioning is the most common practice

We can use API versioning if we believe there to be a breaking change. However:

1. Teams need to build and maintain more code
2. Without knowing what consumers are using, functionality must persist between API versions
3. Consumers need to update to later versions, and teams need to monitor and coordinate this migration
4. Managing the APIs across environments

This overhead and coordination is costly.



If we just used

Interface Definition Languages

Then, we wouldn’t need contract testing

Protobufs (+ Avro and Thrift)

How it aims to solve the problem

1. Designed with **schema evolution** in mind
2. In built forwards and backwards **compatibility**
3. Supports **codegen** to create server/client SDKs

> *"Protocol buffers provide a language-neutral, platform-neutral, extensible mechanism for serializing structured data in a forward-compatible and backward-compatible way. It's like JSON, except it's smaller and faster, and it generates native language bindings."*

Protobufs (+ Avro and Thrift)

Why it doesn’t

Colourless green ideas sleep furiously

Protobufs (+ Avro and Thrift)

Why it doesn't

The curious case of missing merchant payments

> *I lost count of how many bugs we had at <redacted> because people where unaware of the default value behaviour*
>
> *- Poor soul responsible for finding the bug*

Protobufs (+ Avro and Thrift)

Why it doesn't

1. Message semantics
2. Optionals and defaults: a race to incomprehensible APIs
3. Managing breaking changes (e.g. Field descriptors)
4. Providing transport layer safety
5. Narrow type safety (strict encodings)
6. Loss of visibility into real-world client usage
7. Coordinating changes (forwards compatibility)

> *Forwards and backwards compatibility is not enforced: while forwards and backwards compatibility is a promise of Protobuf, actually maintaining backwards-compatible Protobuf APIs isn't widely practiced, and is hard to enforce.*
>
> https://docs.buf.build

If we just used

GraphQL

Then, we wouldn’t need contract testing

GraphQL

How it aims to solve the problem

It shares many of the attributes of schemas, plus ...

1. It is a type system, therefore you get the benefits of types (such as type safety)
2. In built deprecation capabilities to avoid versioning

"GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data. GraphQL provides a complete and understandable description of the data in your API, gives clients the power to ask for exactly what they need and nothing more, makes it easier to evolve APIs over time, and enables powerful developer tools."

```
type Film {
  title: String
  episode: Int
  releaseDate: String
  openingCrawl: String
- director: String
  directedBy: Person
}

type Person {
  name: String
  directed: [Film]
  actedIn: [Film]
}
```

```
type Film {
  title: String
  episode: Int
  releaseDate: String
  openingCrawl: String
+ director: String @deprecated
  directedBy: Person
}

type Person {
  name: String
  directed: [Film]
  actedIn: [Film]

}
```

Evolve your API without versions

Add new fields and types to your GraphQL API without impacting existing queries. Aging fields can be deprecated and hidden from tools. By using a single evolving version, GraphQL APIs give apps continuous access to new features and encourage cleaner, more maintainable server code.

GraphQL

Why it doesn't

1. GraphQL is still likely to interface with non-GraphQL APIs e.g. REST, legacy APIs etc.
2. Deprecation is at runtime [1]
3. Versioning is still a thing / A mechanism for safe evolution is required
4. Loss of sight of API surface area required by consumers [1]
5. Default values

See also: reasons as to why Schemas don't fix it

[1] Apollo's "deprecation" feature is 👌

PACTFLOW
A SMARTBEAR COMPANY

Contract Testing

How it can help

Your Provider Contract

…is only one representation your API

> *With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody*
>
> *- Hyrum's Law*

https://xkcd.com/1172/

Contract Testing

A generalised approach to API communication testing

1. Record / replay → Tests the representative examples against the real provider
2. Specification by example → Reduces ambiguity, improves API comprehension
3. Service evolution → Time travel, by pairing application versions with known supported contracts
4. Transport concerns → Are encoded in the contract
5. Typed field matchers → Provide advanced narrow type system, including semantics (such as dates)
6. API surface area → Is made visible, by the sum of all of the consumer contracts

PACTFLOW
A SMARTBEAR COMPANY

Pact

Extend capabilities via Plugins

With plugins, you can create custom:

1. Transports (e.g. gRPC)
2. Protocols (e.g. protobufs)
3. Matching rules (e.g. semver strings)

Currently in beta (Q4 2022 delivery)

README.md

PACTFLOW

Pact Protobuf/gRPC Plugin

Pact-Protobuf-Plugin Build passing

> Pact plugin for testing messages and gRPC service calls encoded with as Protocol buffers using the Pact contract testing framework.

About this plugin

This plugin provides support for matching and verifying Protobuf messages and gRPC service calls. It fits into the Pact contract testing framework and extends Pact testing for Protocol buffer payloads and gRPC.

Table of Content

Demo

Scenario – Route Guide

```
syntax = "proto3";

package routeguide;

// Interface exported by the server.
service RouteGuide {
  // A simple RPC.
  //
  // Obtains the feature at a given position.
  //
  // A feature with an empty name is returned if there's no feature at the given
  // position.
  rpc GetFeature(Point) returns (Feature) {}

  // ...
}

// Points are represented as latitude-longitude pairs in the E7 representation
// (degrees multiplied by 10**7 and rounded to the nearest integer).
// Latitudes should be in the range +/- 90 degrees and longitude should be in
// the range +/- 180 degrees (inclusive).
message Point {
  int32 latitude = 1;
  int32 longitude = 2;
}

// A feature names something at a given point.
//
// If a feature could not be named, the name is empty.
message Feature {
  // The name of the feature.
  string name = 1;

  // The point where the feature is detected.
  Point location = 2;
}
```

Demo

gRPC example - Consumer

```
func TestRouteServiceGetFeature(t *testing.T) {
    p, _ := message.NewSynchronousPact({...})

    grpcInteraction := `{
        "request": {
            "latitude": "matching(number, 180)",
            "longitude": "matching(number, 200)"
        },
        "response": {
            "name": "notEmpty('Big Tree')",
            "location": {
                "latitude": "matching(number, 180)",
                "longitude": "matching(number, 200)"
            }
        }
    }`

    e[illegible] - GetFeature").
        Given("feature 'Big Tree' exists").
        WithContents(grpcInteraction, "application/protobuf").
        ExecuteTest(t, func(transport message.TransportConfig, m message.SynchronousMessage) error {
            feature := getFeature(transport.Port)
            assert.Equal(t, "Big Tree", feature.GetName())
            assert.Equal(t, int32(180), feature.GetLocation().GetLatitude())

            return nil
        })

    assert.NoError(t, err)
}
```

Demo

gRPC example - Provider

```go
func TestGrpcProvider(t *testing.T) {
    go startProvider()

    verifier := provider.PluginVerifier{}

    err := verifier.VerifyProvider(t, provider.VerifyPluginRequest{
        ProviderAddress: "http://localhost:8222",
        Provider:        "grpcprovider",
        PactFiles: []string{
            filepath.ToSlash(fmt.Sprintf("%s/../pacts/grpcconsumer-grpcprovider.json", dir)),
        },
    })

    assert.NoError(t, err)
}

func startProvider() {
    lis, err := net.Listen("tcp", fmt.Sprintf("localhost:%d", 8222))
    if err != nil {
        log.Fatalf("failed to listen: %v", err)
    }
    var opts []grpc.ServerOption
    grpcServer := grpc.NewServer(opts...)
    pb.RegisterRouteGuideServer(grpcServer, server.NewServer())
    grpcServer.Serve(lis)
}
```

Demo

gRPC example – Provider Output

```
Verifying a pact between grpcconsumer and grpcprovider

  Route guide - GetFeature

  Given a RouteGuide/GetFeature request
      with an input .routeguide.Point message
      will return an output .routeguide.Feature message [OK]
    generates a message which
      has a matching body (OK)
```

Demo

gRPC example – Bad Provider

```
Verifying a pact between grpcconsumer and grpcprovider

  Route guide - GetFeature

  Given a RouteGuide/GetFeature request
      with an input .routeguide.Point message
      will return an output .routeguide.Feature message [FAILED]
    generates a message which
      has a matching body (FAILED)

Failures:

1) Verifying a pact between grpcconsumer and grpcprovider - Route guide - GetFeature
    1.1) has a matching body
           $.name      -> Expected an non-empty string
           $.latitude  -> Expected 'number(180)' to be equal to '180'
           $.longitude -> Expected 'number(200)' to be equal to '200'

There were 1 pact failures

=== RUN   TestGrpcProvider/Provider_pact_verification
```

PACTFLOW
A SMARTBEAR COMPANY

Summary

Key takeaways

1. Multi-protocol internal microservice adoption is accelerating
2. Lack of standardization for design and test is contributing to the challenges of “microservices sprawl”
3. Hyrum’s law – need to reduce ambiguity
4. Contract testing is an approach that can reduce the complexity of API testing and the ambiguity inherent in all API specifications
5. Pact is a contract testing tool that can be used to standardise the API communication testing across languages, transports and protocols

Read the blog

THANK YOU

Visit the Pact docs

Get in touch

@matthewfellows

pactflow.io