用 Pact / Swagger 做契约测试

DEC 2018

API 测试

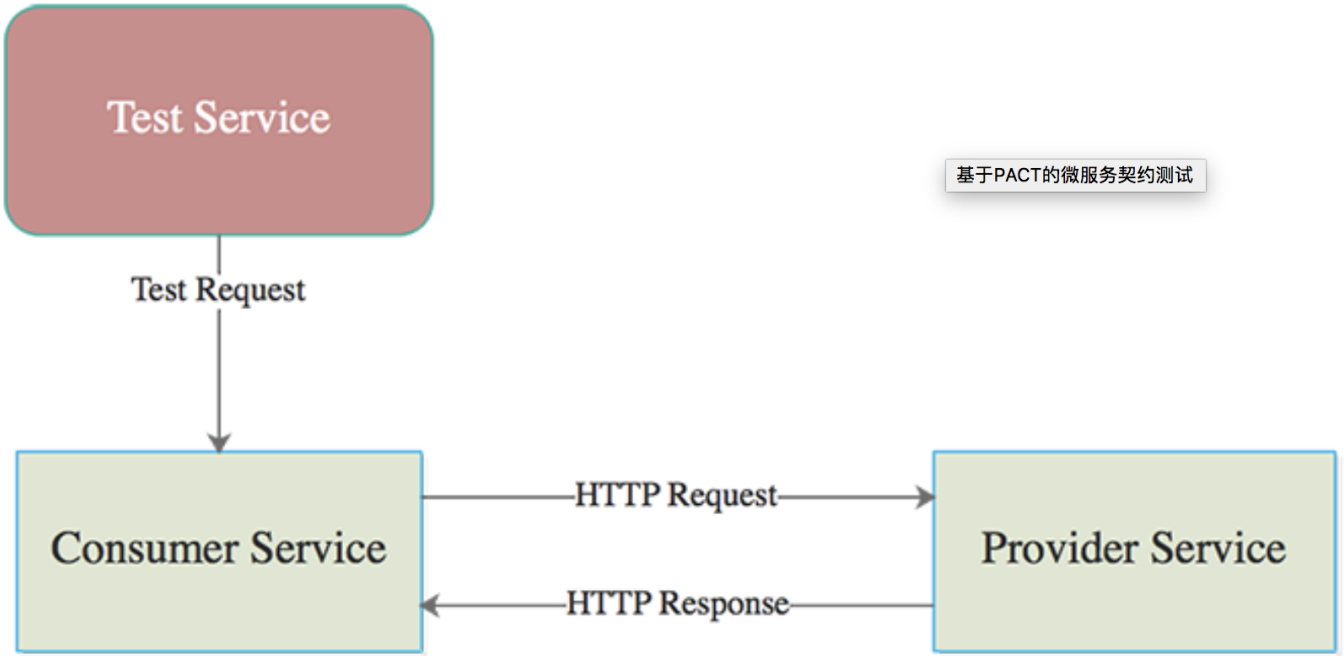
ThoughtWorks®

微服务架构 – More API will be included

- Martin Fowler: Microservices
- 单一应用被划分成一组小的服务
- 运行在单独的进程中
- 轻量级通信机制
- 松耦合，可独立部署、独立运行，可以并行开发

传统 API 测试方法

- 通常我们这样做集成测试



基于PACT的微服务契约测试

传统 API 测试方法在微服务框架中碰到的问题

- 环境搭建成本高，需启动多个服务
- 测试脆弱，外部依赖多，需要协调和统一多个组件的版本
- 学习成本高，用例编写难
- 运行慢

契约测试是如何解决这些问题

- 传统意义上需要同时运行多个服务的集成流程可被分解为多个集成点，每个集成点可被单独测试。
- Mock 服务的够降低测试的不稳定性。
- 更易定位测试失败的原因，因为一次只测试单个组件。
- 更快的执行速度。

契约测试介绍

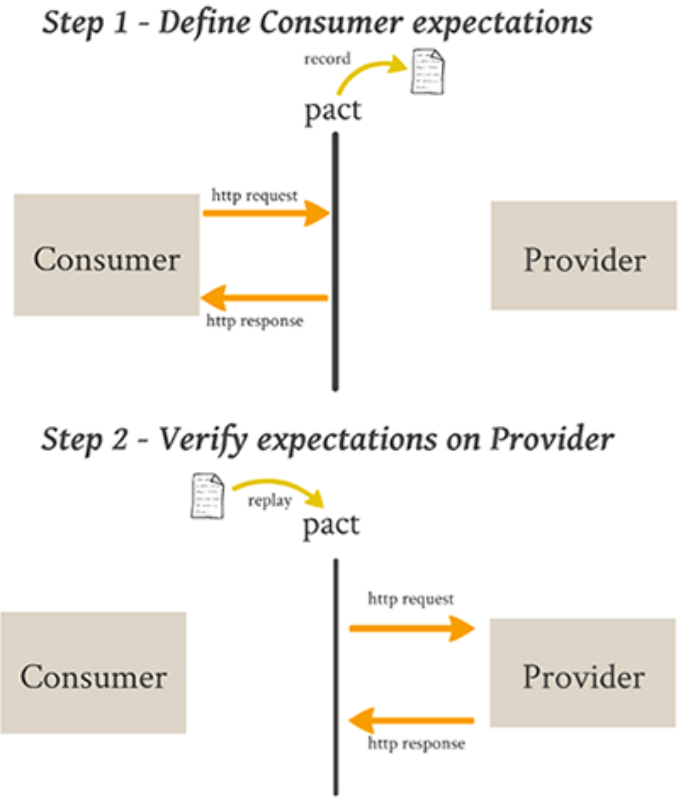
ThoughtWorks®

消费者驱动的契约测试—概念

- 契约
  - 以结构化文档（json 或 yaml 等）形式对服务接口的约定
- 契约测试
  - 以契约作为唯一耦合点，验证提供者端的实现是否满足消费者端需求的过程
- 消费者驱动的契约测试
  - 从消费者端的期望出发定义契约，并以契约驱动提供者端测试的过程

消费者驱动的契约测试—步骤

- 步骤 1： 消费者端
  - 编写并运行单元测试（包括对接口的请求参数和预期响应）
  - MockService 代替实际服务提供者（自动）
  - 生成契约文件（自动）
- 步骤 2： 提供者端
  - 启动服务提供者
  - 重放契约文件中的请求，验证真实响应是否满足预期（自动）



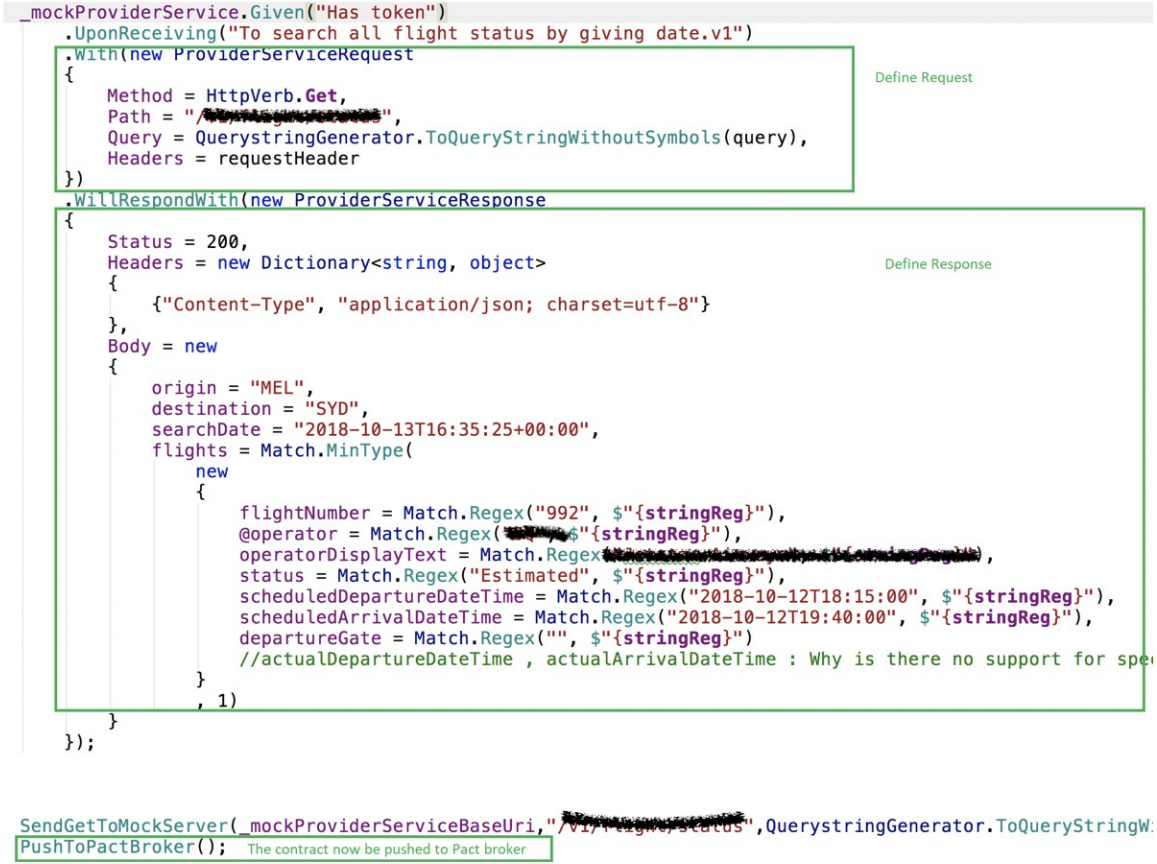
消费者驱动的契约测试—价值

- 将一个笨重的集成测试化为两个容易编写、容易运行的单元测试 / 接口测试
- 解耦消费者与提供者，甚至可以在没有提供者实现的情况下开展消费者端测试
- 通过测试保证契约和实现的一致性
- CI 的一部分，便于尽早发现问题
- 工具 / 框架：
  - PACT（推荐）
    - 轻量级、支持多种语言应用、支持与 Maven/Gradle 等集成
  - Spring Cloud Contract
    - 支持基于 JVM 的应用、支持与 Spring 其它组件集成

使用 PACT 实现契约测试——例子

ThoughtWorks®

Consumer side, define request and response, and after using mock server then generate a **contract json file**.



Pact broker:

http://test-pact-broker-alb-12546

xxxxxx

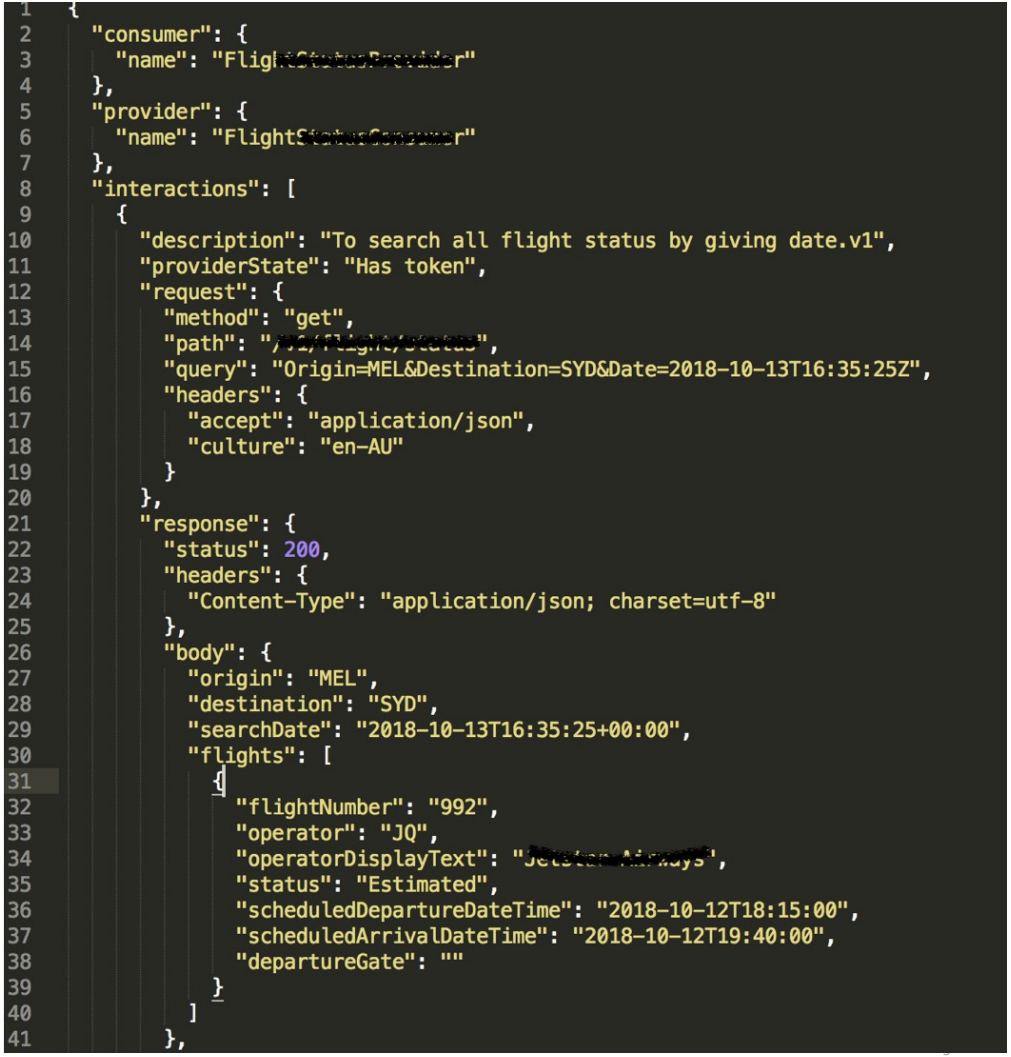
. xxxxxxxxxx-2.xxxxx.amazonaws.com/

("admin", "xxxxxxx")

C

Contract file (json file)

- 契约文件（由 PACT 自动生成）
- 以 json 形式对接口形式作出约定



```
{
  "consumer": {
    "name": "Flig[redacted]"
  },
  "provider": {
    "name": "Flight[redacted]r"
  },
  "interactions": [
    {
      "description": "To search all flight status by giving date.v1",
      "providerState": "Has token",
      "request": {
        "method": "get",
        "path": "[redacted]",
        "query": "Origin=MEL&Destination=SYD&Date=2018-10-13T16:35:25Z",
        "headers": {
          "accept": "application/json",
          "culture": "en-AU"
        }
      },
      "response": {
        "status": 200,
        "headers": {
          "Content-Type": "application/json; charset=utf-8"
        },
        "body": {
          "origin": "MEL",
          "destination": "SYD",
          "searchDate": "2018-10-13T16:35:25+00:00",
          "flights": [
            {
              "flightNumber": "992",
              "operator": "JQ",
              "operatorDisplayText": "[redacted]",
              "status": "Estimated",
              "scheduledDepartureDateTime": "2018-10-12T18:15:00",
              "scheduledArrivalDateTime": "2018-10-12T19:40:00",
              "departureGate": ""
            }
          ]
        },
```

Provider side, replay the **contract json file** on real server.

```
[Fact]
public void EnsureProviderApiHonoursPactWithConsumer()
{
    IPactVerifier pactVerifier = new PactVerifier(PactVerifierConfig);
    pactVerifier.ServiceProvider("FlightStatusProvider", ProviderUri)
        .HonoursPactWith("FlightStatusConsumer")
        .PactUri(
            @"https://jed[redacted],
            new PactUriOptions("admin", "[redacted]")
        .Verify();
}
```

Read contract JSON file from server

Replay request and response by using real server

Pact 特性

ThoughtWorks®

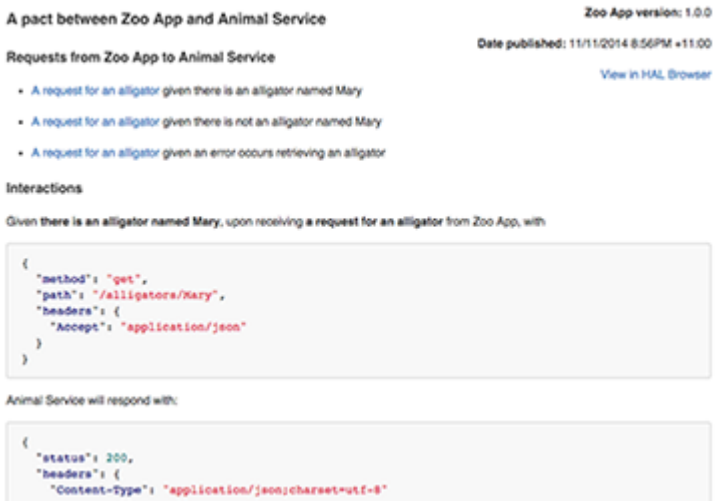
预置提供者状态—— Provider State

- Provider State： 提供者的某种状态，如预置了测试数据、模拟了异常等
- 作用：赋予对同一接口的不同用例 / 场景的测试能力
- 消费者端指定 state：
  builder.given("some state")
- 契约文件中：
  "providerState": "some state"
- 提供者端通过 state 入口（对应的用例执行前）对提供者状态进行修改：

```
@State("some state")
public void toSomeState() {
// Prepare service before interaction that require "some" state
// ...
System.out.println("Now service in some state");
}
```

契约文件的共享方法

- 怎样在消费者和提供者之间共享契约文件?
  - A. 手动 Copy 或公共目录（简单，适于本地运行）
  - B. 使用 Pact Broker作为中间服务（推荐用于 CI ）： 可与 CI 集成完全自动化、可管理版本、 API 可视化、可自动生成服务调用关系图



A pact between Zoo App and Animal Service

Zoo App version: 1.0.0

Date published: 11/11/2014 8:56PM +11:00

View in HAL Browser

Requests from Zoo App to Animal Service

- A request for an alligator given there is an alligator named Mary
- A request for an alligator given there is not an alligator named Mary
- A request for an alligator given an error occurs retrieving an alligator

Interactions

Given **there is an alligator named Mary**, upon receiving **a request for an alligator** from Zoo App, with

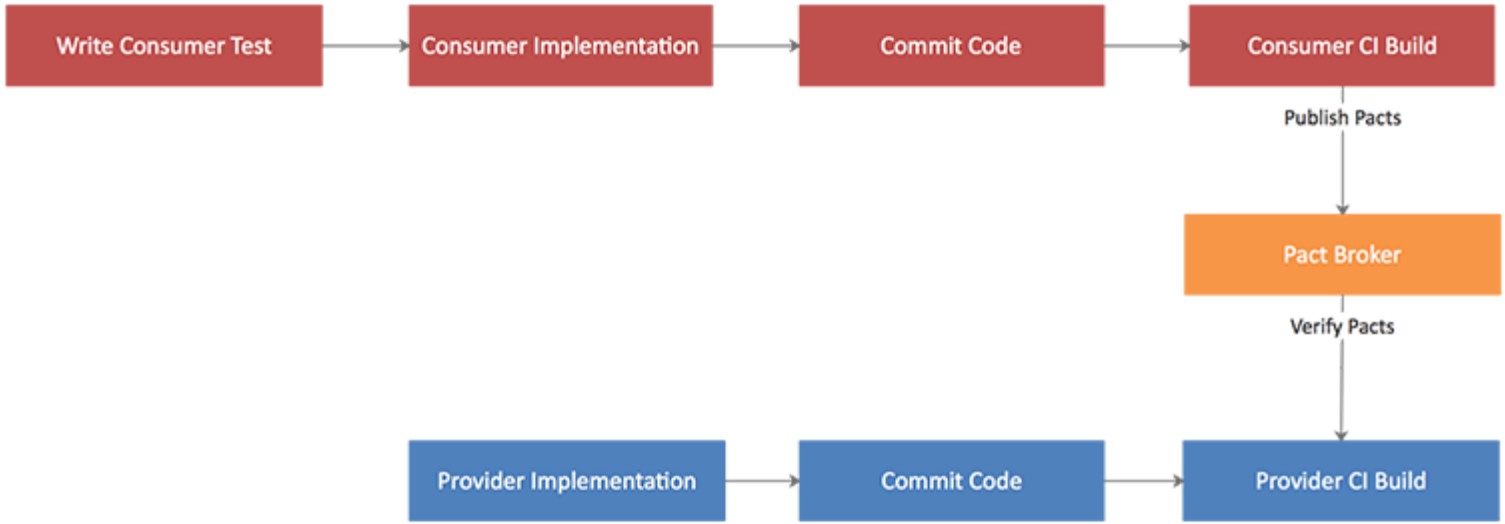
```
{
  "method": "get",
  "path": "/alligators/Mary",
  "headers": {
    "Accept": "application/json"
  }
}
```

Animal Service will respond with:

```
{
  "status": 200,
  "headers": {
    "Content-Type": "application/json;charset=utf-8"
```

使用 Pact Broker 实现契约管理

- 搭建 Pact Broker：
  1. 推荐使用 Dockerised Pact Broker + Docker Compose实现一条命令式安装
- 使用 Pact Broker：
  1. 消费者端的 CI build 通过后，将 pact 文件 push 到 pact broker 上
  2. 提供者端的 CI build 时，从 pact broker 上取 pact 文件进行测试



C# 版本契约测试碰到的坑

ThoughtWorks®

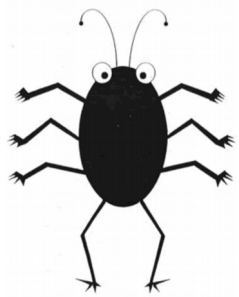
1. C# 版本的 Pact, 运行 consumer 端的单元测试，发现契约文件没有被上传到 broker 中，需要重新运行一遍才可以达到目的。 (change pact JSON version could do it)

2. 按照格式编写 response body 比较繁琐。在 C# 的 Pact 版本中，不能把这些 json 信息直接 copy 到代码中使用，例如 body 中信息类似右侧 Json 格式。如果 body 中的信息比较多，这个手工修改量让人汗颜。

```
{
  "origin":"Xian",
  "passengers":[
    {
      "type":"ADT",
      "discountCode":""
    }
  ]
}
```

在 C#code 中需要改成：

```
{
   origin = "MEL",
   passengers = new List<Object>(){
      new {
         type = "ADT",
         discountCode = ""
      }
   },
}
```



契约测试和传统 API 测试的区别

ThoughtWorks®

契约测试和传统的 API 测试（例如使用 Jmeter 做的传统接口 / 集成测试）有相似点，都包含发送 request 和验证 response 的步骤。两者区别是什么?
契约测试解决了几个传统 API 测试不好解决的问题。
例如设想两个场景：

场景 1 ： 前后端开发小组希望可以异步开发，互不影响

传统使用 Jmeter 的 API 测试，碰到这种情况，直觉是想在前端用一个 mock server 来做 ，但接着又会碰到一系列问题， mock server 中的 request 和 response 应该是 consumer 去定义，还是 provider ，接着，定义好的文件放在那里？可以被 consumer 和 provider 容易的获取 .

provider 端开发了一部分功能，想做验证，怎么做？这时候需要在 provider 端回放 consumer 端定义的 request 和 response 。

最后，当有人想从 high level 去查看微服务之间的调用关系时候， Pact 提供了自动生成服务调用关系图的功能。

Pact 作为测试框架，比较优雅的解决了这一系列的问题。

场景 2 加快错位排查速度

当 consumer 端定义了 request 和 response 后，运行 consumer 端的契约测试会先生成契约文件，实际上我们会在接受 mock server 返回的数据后，接着测试一些其他业务功能 [因为这些功能依赖于 response 的内容]，假设相关功能是一个 1000 行的复杂业务逻辑代码。

单元测试通过了。

过了两天后，发现这个单元测试失败了。

按理，如果这两天内定义的契约文件没有发生变化，那么导致测试失败的原因肯定是在哪个 1000 行代码的业务逻辑里，也就是说，这个错误是由 consumer 端的开发人员造成的。我们不需要去排查是不是其他组的 provider 提供的接口发生了变化。

而以前，用 jmeter 、 postman 等做 api 测试，由于 consumer 端和 provider 端的返回的真实结果紧密耦合，无法立刻判断出出错的点。这时错误的原因可能会有两个，一种是 consumer 端的 1000 行的业务逻辑代码被改变了，也可能是 provider 端修改了 response 的内容才导致测试失败。接着要花费相对更多的时间去排查错误。

怎样做契约测试才不会陷入令人抓狂的误区

ThoughtWorks®

一个简单的消费者与提供者交互的常规场景如下所示：

```
Given "there is no user called Mary"
When "creating a user with username Mary"
  POST /users { "username": "mary", email: "...", ... }
Then
  Expected Response is 200 OK
```

如果只验证常规场景，则会遗漏不同响应的其他场景，并可能使消费者误解提供者的行为。所以，我们还需要验证失败的场景：

```
Given "there is already a user called Mary"
When "creating a user with username Mary"
  POST /users { "username": "mary", email: "...", ... }
Then
  Expected Response is 409 Conflict
```

可以看出，通过使用不同的响应代码，我们覆盖了一个新的错误场景验证。

接下来，*用户服务*的团队告诉我们，用户名只允许包含字母，其最大长度不能超过20个字符，空白的用户名是无效的。那我们是不是应该在契约测试中加点东西呢?

这会成为连锁反应的开始，很可能会向契约测试中添加3个场景，类似于这样：

```
When "creating a user with a blank username"
  POST /users { "username": "", email: "...", ... }
Then
  Expected Response is 400 Bad Request
  Expected Response body is { "error": "username cannot be blank" }
```

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

```
When "creating a user with a username containing numbers"
  POST /users { "username": "us3rn4me", email: "...", ... }
Then
  Expected Response is 400 Bad Request
  Expected Response body is { "error": "username can only contain letters" }
```

到这一步为止我们其实已经滥用了契约测试，我们实际上是在测试*用户服务*是否正确实现了校验规则：而这实际上是功能测试，应该在*用户服务*自己的代码库中所覆盖。

简而言之，契约场景不应该深入到提供者的业务逻辑细节，而应该专注于验证消费者和提供者是否对请求和响应达成共识。

如何用 Swagger 做契约测试

ThoughtWorks®

用 Swagger 做契约测试 - 1

步骤：

1. 首先 swagger 的 yaml 或者 json 格式的 api specfication ， 和 Pact 中的 Json 格式契约文件起一样的作用。
2. 接着假设 provider 端还没有开发完成，我们在 consumer 端做测试，这时候就需要第一个工具 Prism 来支撑，实际上是个 mockserver ， 这个 mockserver 去解析 swagger 中的 api sepcfication ， 就可以根据定义的 request 去返回 response 。
3. 最后，假设 provider 开发出一个版本代码后，需要验证 provider 端的代码是否能很好的支撑 consumer ， 那它也会先拿到 api sepcfication ， 接着用第二个工具 Dredd 去在 provider 端，使用真实的 API server 做 api specfication 的回放。

用 Swagger 做契约测试 - 2

Pact 是 Consumer Driven Testing， 这是 Pact 强制要求的。 Swagger 的契约测试可以做成 provider driven, 也可以做成 consumer driven 。

Thank you

ThoughtWorks®