AT28

Agile and Continuous Testing
Thursday, November 7th, 2019 4:45 PM

Contract Testing with Pact: A Different Approach

Presented by:

Mihail Mikulaninec

MOO

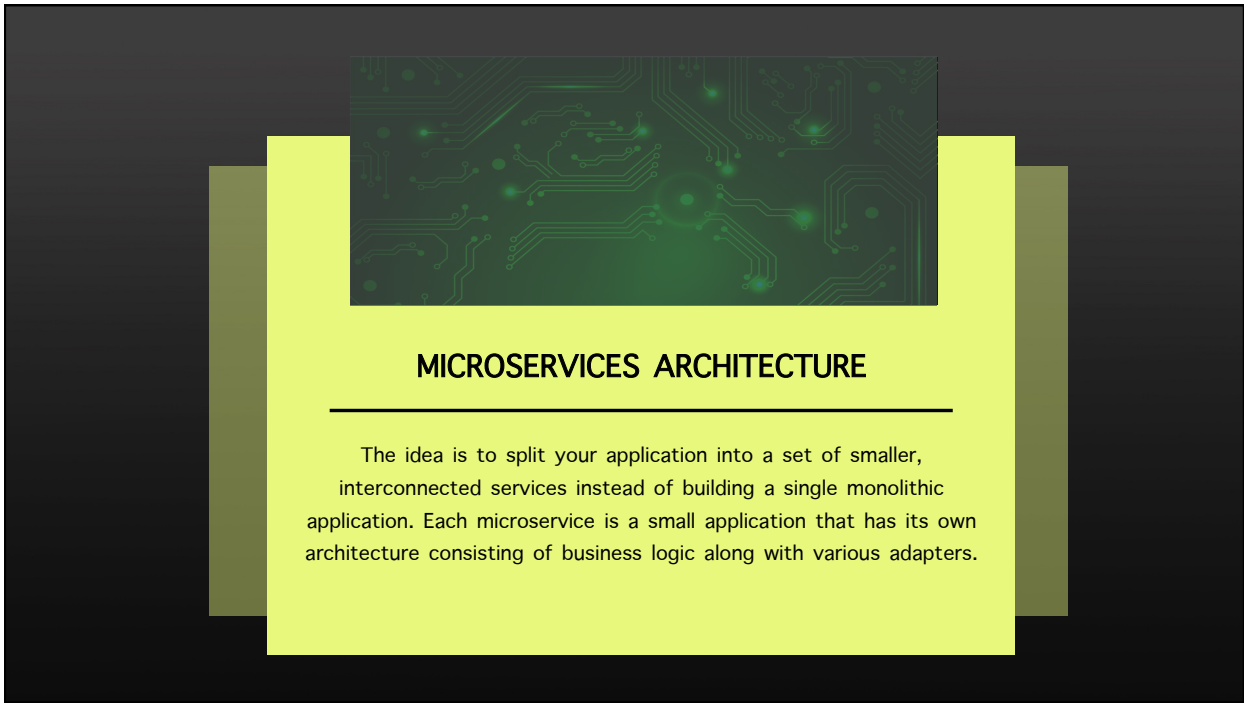
Brought to you by:

888-268-8770 · 904-278-0524 - info@techwell.com
https://agiledevopseast.techwell.com/

Mihail Mikulaninec

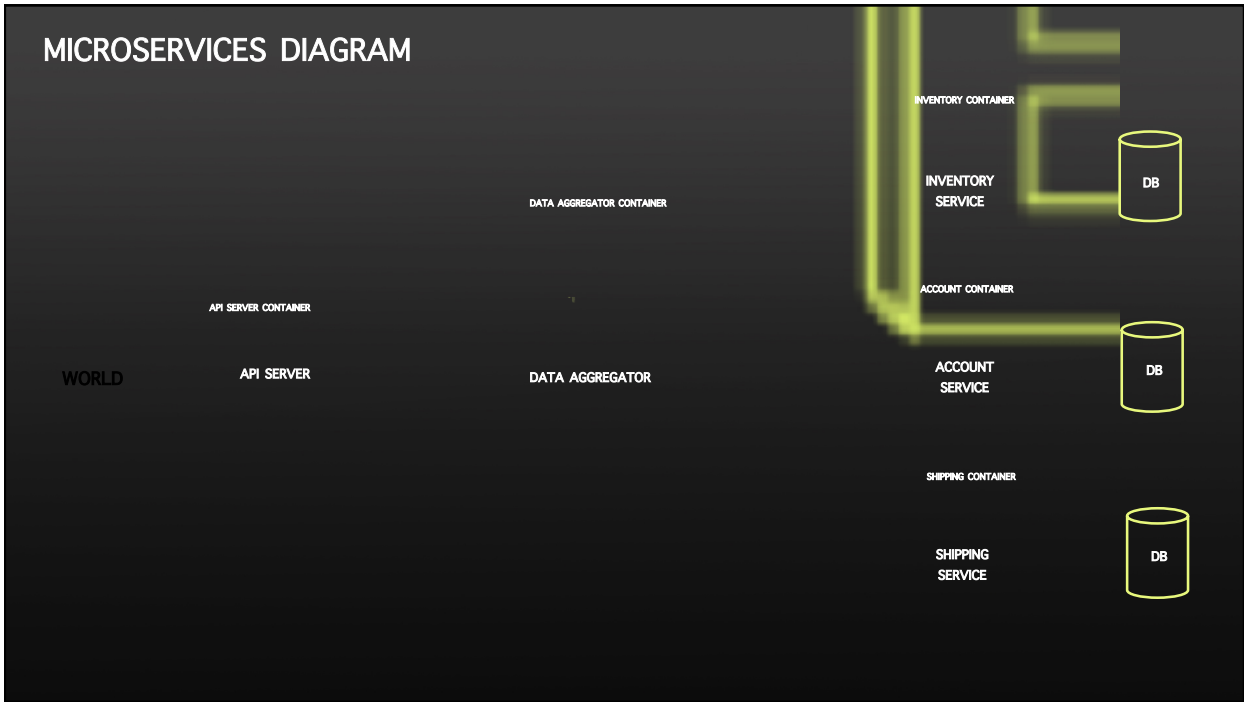
Mihail Mikulaninec is a physicist that evolved from a developer at a medical company to an automation QA engineer. He has worked with various businesses (fintech, medicine, design, printing) and tech stacks (C++, C#, Ruby, Python, React, Java). Mihail is passionate about microservices architecture and CI/CD with a focus on quality. He has been involved in the decoupling strategy for a number of companies in order to move toward a microservices world. He currently serves as the senior automation QA engineer at MOO, in London, UK.

ABOUT MICROSERVICES



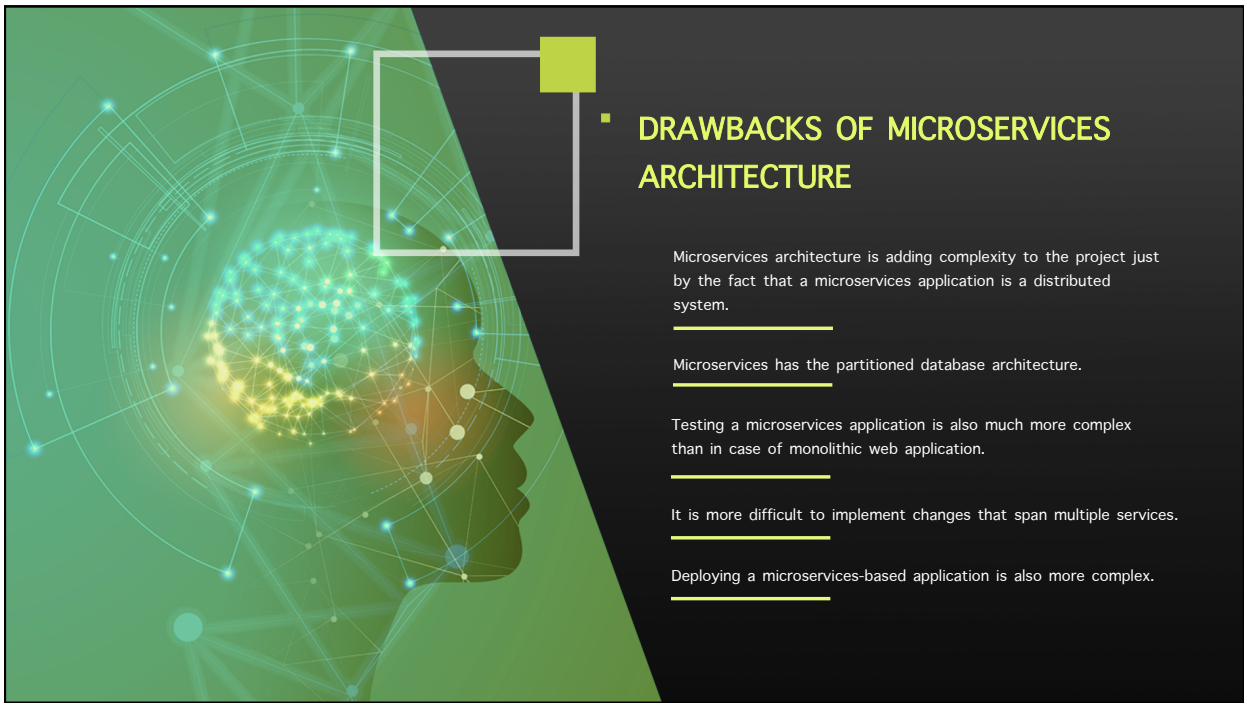
MICROSERVICES ARCHITECTURE

The idea is to split your application into a set of smaller, interconnected services instead of building a single monolithic application. Each microservice is a small application that has its own architecture consisting of business logic along with various adapters.



BENEFITS OF MICROSERVICES ARCHITECTURE

It tackles the problem of complexity by decomposing application into a set of manageable services	It enables each service to be developed independently	It reduces barrier of adopting new technologies since the developers are free to choose	Microservice architecture enables each microservice to be deployed independently.	Microservice architecture enables each service
which are much faster to develop, and much easier to understand and maintain.	by a team that is focused on that service.	whatever technologies make sense for their service and not bound to the choices made at the start of the project.	As a result, it makes continuous deployment possible for complex applications.	to be scaled independently.



DRAWBACKS OF MICROSERVICES ARCHITECTURE

- Microservices architecture is adding complexity to the project just by the fact that a microservices application is a distributed system.
- Microservices has the partitioned database architecture.
- Testing a microservices application is also much more complex than in case of monolithic web application.
- It is more difficult to implement changes that span multiple services.
- Deploying a microservices-based application is also more complex.

ABOUT PACT

CONTRACT TESTING WITH PACT

Pact is a contract testing tool. Contract testing is a way to ensure that services (such as an API provider and a client) can communicate with each other. Without contract testing, the only way to know that services can communicate is by using expensive and brittle integration tests.

Contract testing is immediately applicable anywhere where you have two services that need to communicate - such as an API client and a web front-end.

In general, a contract is between a consumer (for example, a client that wants to receive some data) and a provider (for example, an API on a server that provides the data the client needs).

CONSUMER

A client that wants to receive some data

MICROSERVICE ARCHITECTURE

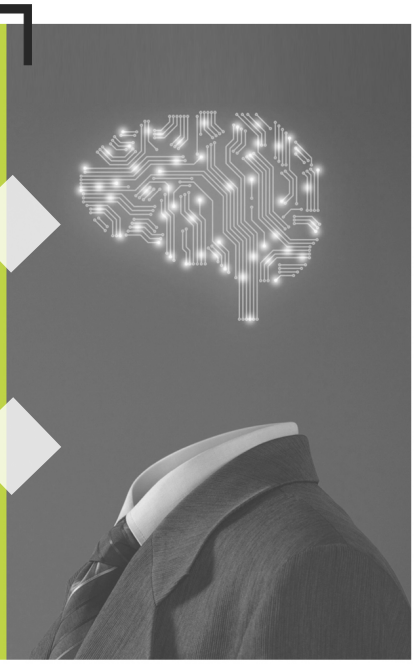
A service or server that provides the data

PROVIDER

A contract between a consumer and provider is called a PACT. Each pact is a collection of interactions. Each interaction describes:

An **EXPECTED REQUEST** - describing what the consumer is expected to send to the provider

A **MINIMAL** expected response - describing the parts of the response the consumer wants the provider to return.



CONSUMER TESTING

Consumer Pact tests operate on each interaction to say "assuming the provider returns the expected response for this request, does the consumer code correctly generate the request and handle the expected response?".

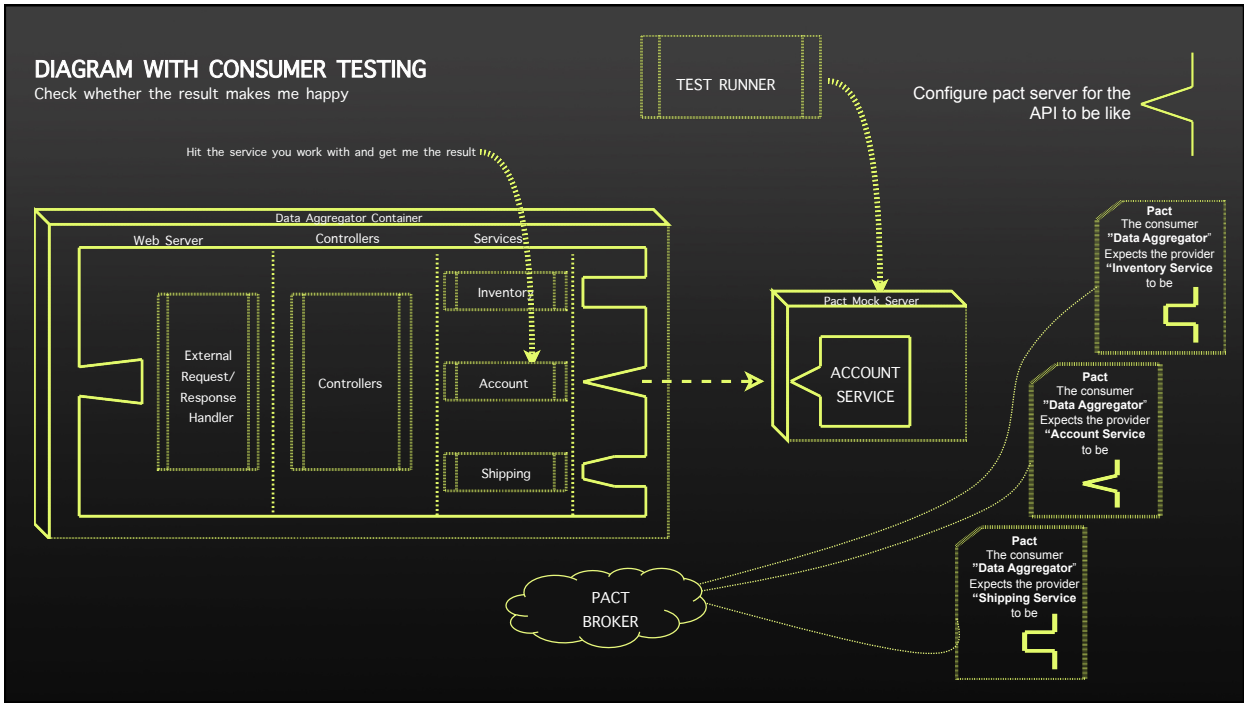
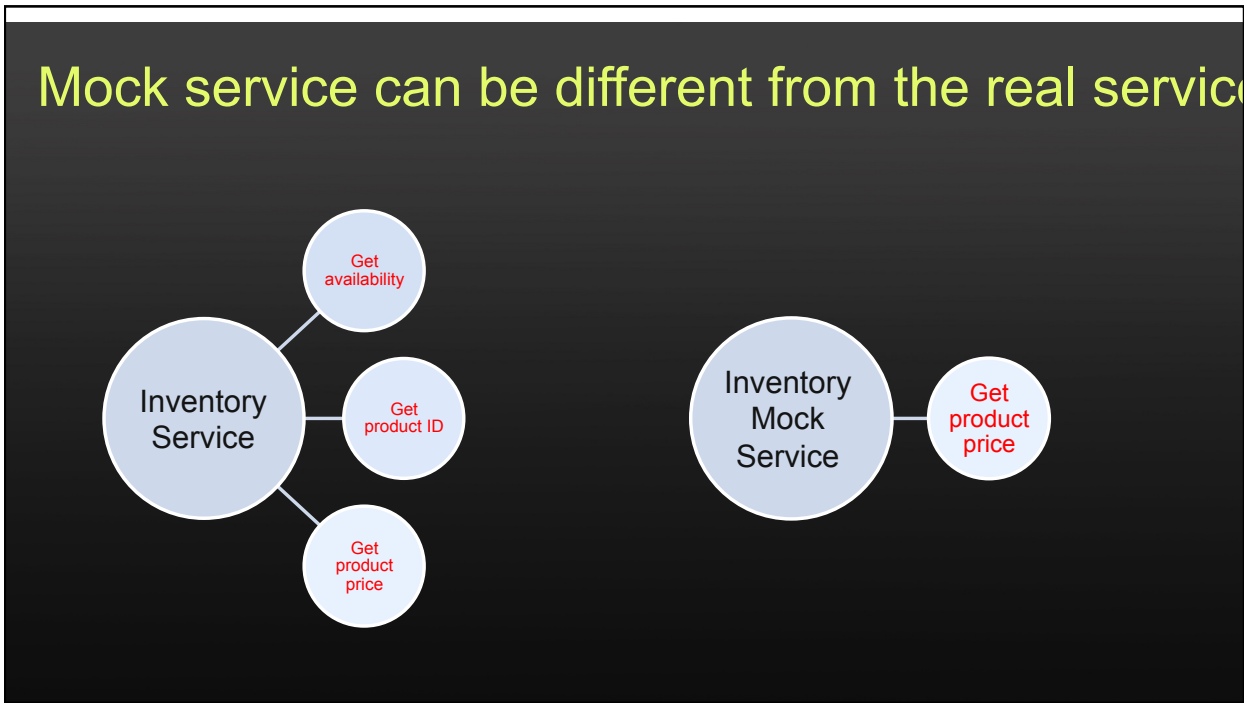


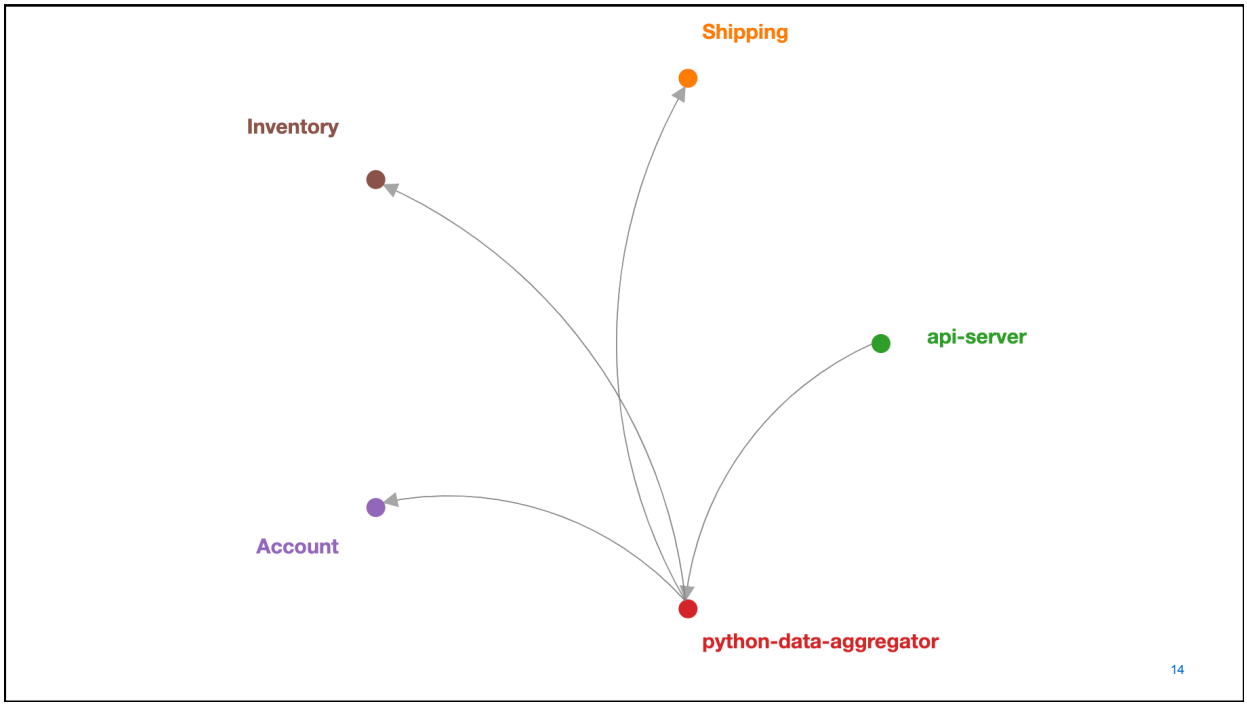
DIAGRAM WITH CONSUMER TESTING
Check whether the result makes me happy
TEST RUNNER
Configure pact server for the API to be like
Hit the service you work with and get me the result
Data Aggregator Container
Web Server
Controllers
Services
External Request/ Response Handler
Controllers
Inventory
Account
Shipping
Pact Mock Server
ACCOUNT SERVICE
Pact
The consumer "Data Aggregator"
Expects the provider "Inventory Service"
to be
Pact
The consumer "Data Aggregator"
Expects the provider "Account Service"
to be
Pact
The consumer "Data Aggregator"
Expects the provider "Shipping Service"
to be
PACT BROKER


Mock service can be different from the real servic
Inventory Service
Get availability
Get product ID
Get product price
Inventory Mock Service
Get product price

PACT BROKER

is an application for sharing for consumer driven contracts and verification results. It is optimised for use with "pacts" (contracts created by the Pact framework), but can be used for any type of contract that can be serialized to JSON.

1. Tells you which versions of your applications can be deployed safely together
   - Allows you to release customer value quickly and confidently
2. Allows you to ensure backwards compatibility between multiple consumer and provider versions
   - Shows you real examples of how your services interact
3. Provides API documentation of your applications that is guaranteed to be up-to date
   - Allows you to visualise the relationships between your services
4. Solves the problem of how to share contracts and verification results between consumer and provider projects
   - Automatically versions your contracts



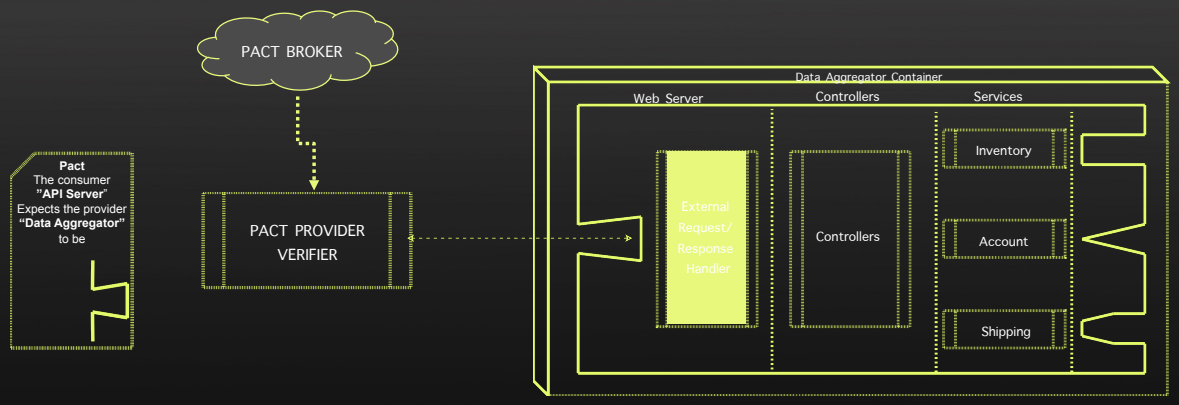
Pacts

Consumer		Provider	Latest pact published	Webhook status	Last verified	
api-server		python-data-aggregator	12 days ago	Create		•••
python-data-aggregator		Account	13 days ago	Create		•••
python-data-aggregator		Inventory	13 days ago	Create		•••
python-data-aggregator		Shipping	13 days ago	Create		•••

4 pacts

15

PROVIDER VERIFICATION



ABOUT SPECIAL WAY OF IMPLEMENTATION

CONVENTIONAL IMPLEMENTATION
Consumer Tests

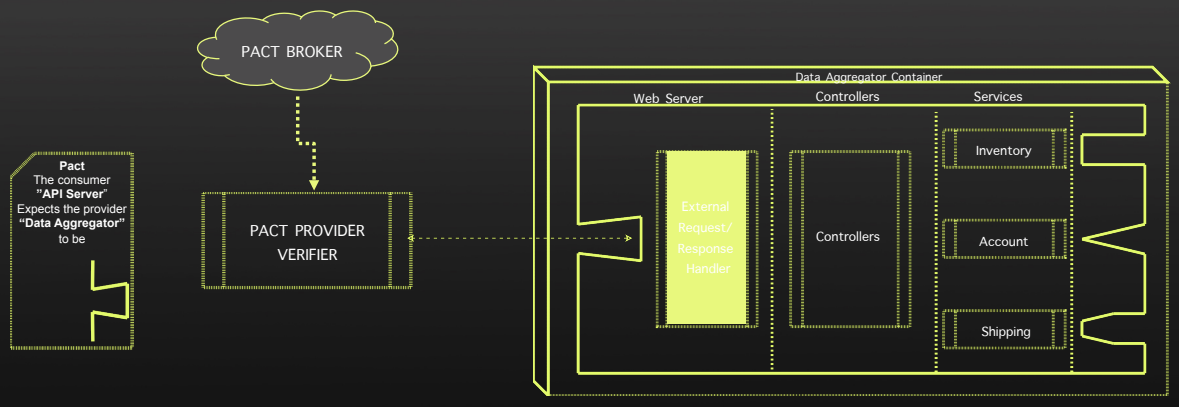
```
import atexit
import unittest

from pact import Consumer, Provider


pact =
Consumer('Consumer').has_pact_with(Provider('Provider'
))
pact.start_service()
atexit.register(pact.stop_service)


class GetUserInfoContract(unittest.TestCase):
 def test_get_user(self):
   expected = {
     'username': 'UserA',
     'id': 123,
     'groups': ['Editors']
   }
```

```
(pact
 .given('UserA exists and is not an administrator')
 .upon_receiving('a request for UserA')
 .with_request('get', '/users/UserA')
 .will_respond_with(200, body=expected))

with pact:
  result = user('UserA')

self.assertEqual(result, expected)
```

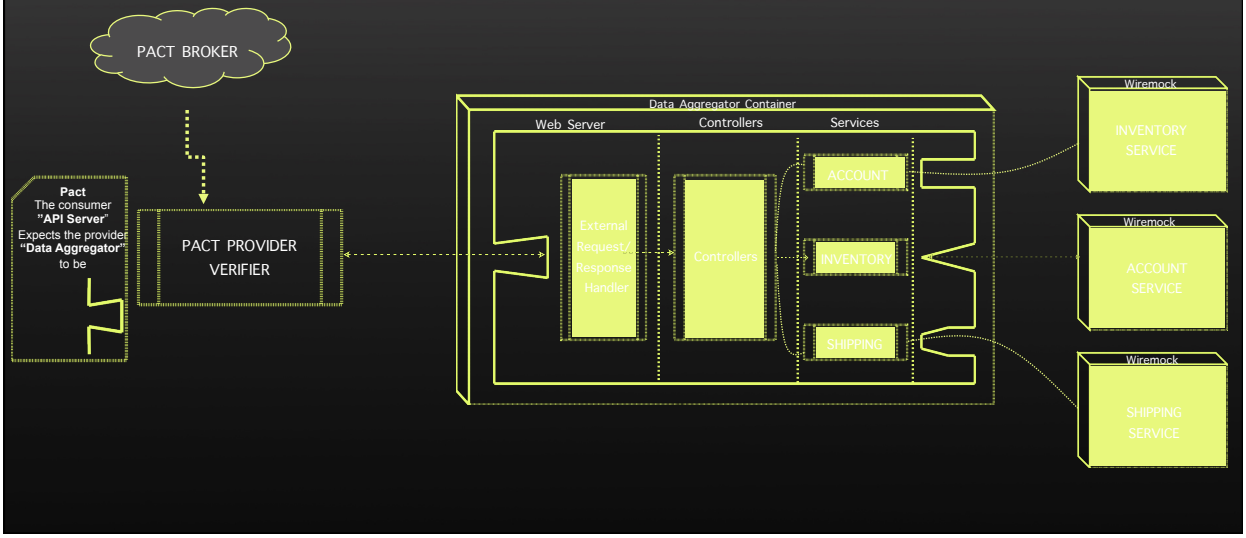
PROVIDER VERIFICATION

Comparison of conventional ways of implementation of provider verification

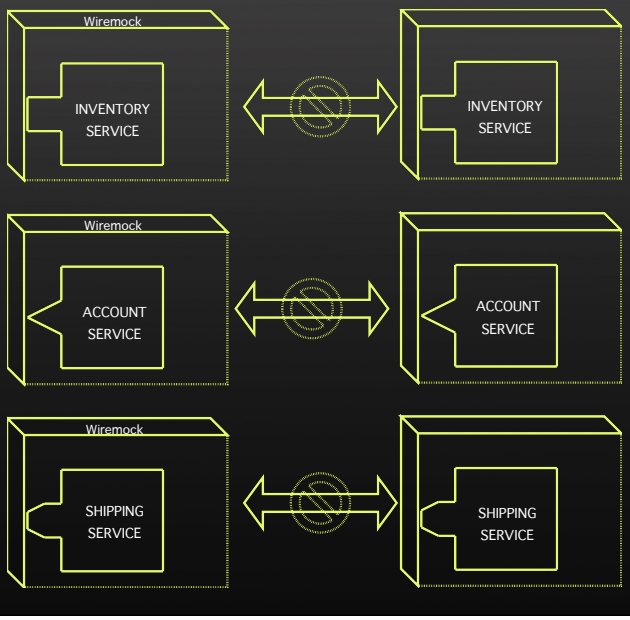
Provider verification with stubbed handler (conventional way)



PROVIDER VERIFICATION PROVIDER VERIFICATION WITH 3-D PARTY MOCK SERVICES



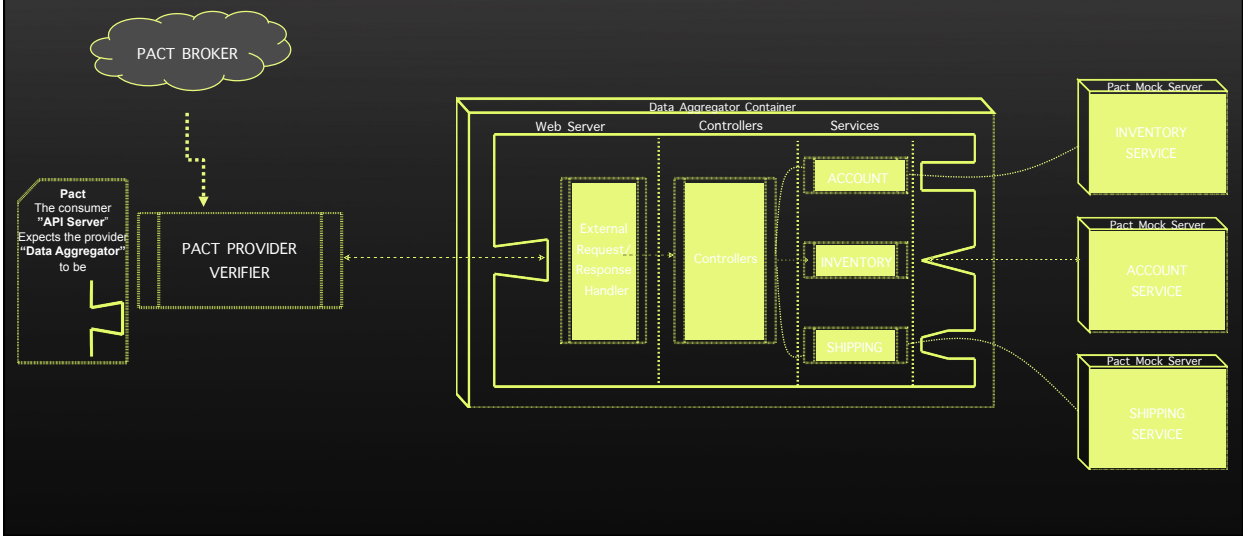
MOCK SERVICE CONFIGS CAN GO OUT OF SYNC



WHAT IF WE CAN USE PACT MOCK SERVICES FOR THE PROVIDER VERIFICATION?

It will save us from the potential huge code repetition (we need to configure the mock servers in consumer tests and in 3-d party mocks for provider verification). They both should be maintained and can eventually go out of sync. We should keep in mind that configurations can be MASSIVE

NEW VERIFICATION DIAGRAM



CODE SNIPPET FOR MOCK SERVER CONFIGURATIONS

```
from pact import EachLike, Like

BASE_CONFIG = {
  'host': '127.0.0.1',
  # path to the dir where the pact files will be stored. Eventually
should be centralised int the Pact Broker
  'path_to_pacts': 'pacts/python/',
  'consumer': 'python-data-aggregator'
}

# each service should contain name and the following keys (case
sensitive):
# 'GIVEN' - a description string about the initial data needed for
using of tht API,
# 'UPON_RECEIVING' - a string description of the expected
outcomes,
# 'REQUEST' - the expected request to the API (see pact.io for
details),
# 'RESPONSE' the expected response from the API (see pact.io for
the details)
SERVICES = {
  'Inventory': {
    'port': 1111,
    'interactions':
      [
        {
          'GIVEN': 'Provided we have the correct
project_id                                    and data to edit',
          'UPON_RECEIVING': 'the correct paper URL
                                              should be provided',
          'REQUEST': {
            'method': 'get',
            'path': '/'
          },
          'RESPONSE': {
            'status': 200, 'body': EachLike({
              "id": Like(1),
              "name": Like("Pants"),
              "quantity": Like(15)
            })
          }
        }
      ]
  }
```

CODE SNIPPET FOR MOCK SERVICE MANAGER

```
from pact import Consumer, Provider
import atexit
from tests.mock_servers_configurations import BASE_CONFIG, SERVICES


class MockApiServer:
  def __init__(self, service, log_dir='contract_logs/consumer'):
    config = SERVICES[service]
    self.pact=Consumer(BASE_CONFIG['consumer'])
              .has_pact_with(Provider(service),

host_name=BASE_CONFIG['host'],
                                port=config['port'],
                                pact_dir=BASE_CONFIG['path_to_pacts'],
                                log_dir=log_dir)
    for interaction in config['interactions']:
      (self.pact.given(interaction['GIVEN'])
       .upon_receiving(interaction['UPON_RECEIVING'])
       .with_request(**interaction['REQUEST'])
       .will_respond_with(**interaction['RESPONSE']))

  def stop(self):
    self.pact.stop_service()
    return self

  def start(self):
    self.pact.start_service()
    self.pact.setup()
    return self

  def safe_start(self):
    self.start()
    atexit.register(self.stop)
    return self.pact

  @staticmethod
  def safe_start_all_services():
    for service in SERVICES.keys():
      MockApiServer(service).safe_start()
```

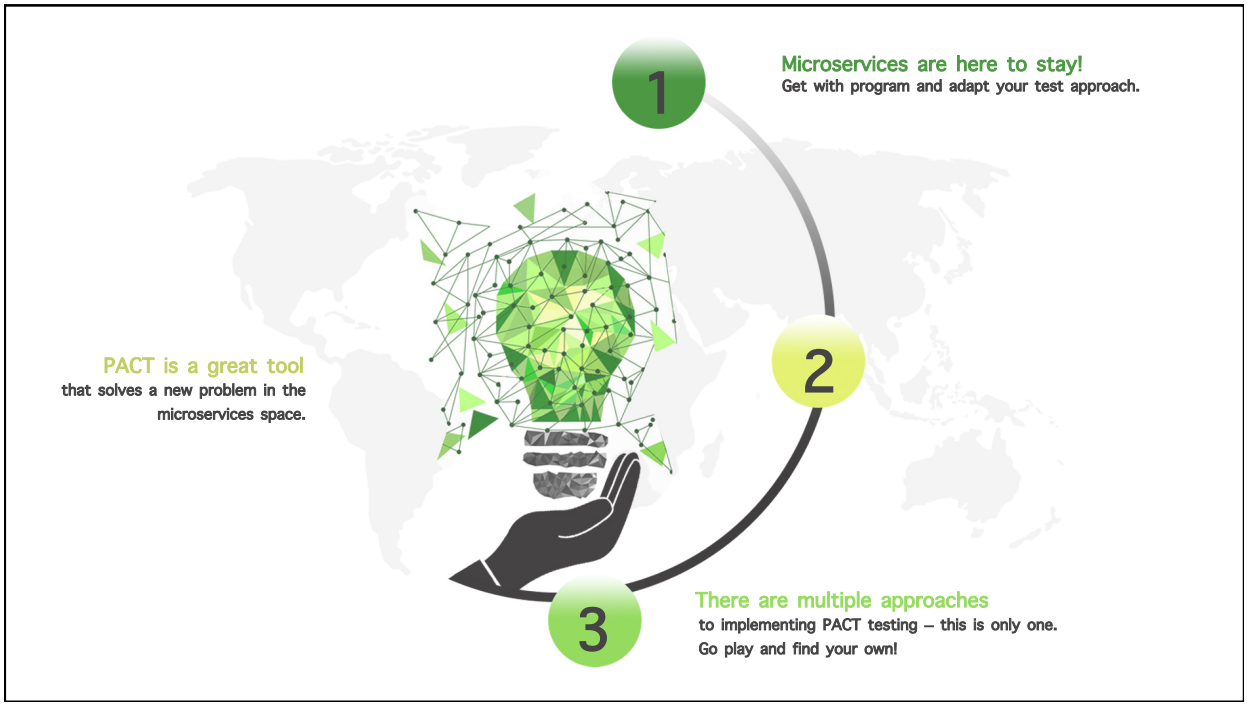
```
from tests.mock_server import MockApiServer
from services.inventory import Inventory
from services.shipping import Shipping
from services.account import Account

// Start the inventory server and stop it afterwards
inventory_pact_server = MockApiServer("Inventory").safe_start()

inventory_api = Inventory("http://localhost:
{}".format(inventory_pact_server.port))
```

```
def test_inventory_main():
    response = inventory_api.get_data()
    assert response == [{'id': 1, 'name': 'Pants',
                         'quantity': 15}]


def test_inventory_get_product_name():
    response = inventory_api.get_product_name(1)
    assert response == "Pants"
```

CODE SNIPPET FOR ACCOUNT SERVICE

```
from services.base import Base


class Account(Base):
    def get_data(self):
        return self.get("/")

    def get_account(self, ac_id):
        return [el for el in self.get_data() if el["id"] == ac_id]
[0]
```

SUMMARY


1
Microservices are here to stay!
Get with program and adapt your test approach.
2
PACT is a great tool
that solves a new problem in the
microservices space.
3
There are multiple approaches
to implementing PACT testing – this is only one.
Go play and find your own!

THANK YOU FOR YOUR ATTENTION

REFERENCES

1) https://www.mulesoft.com/resources/api/microservices-vs-monolithic
2) https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59
3) https://microservices.io/patterns/monolithic.html
4) https://labs.spotify.com/2018/01/11/testing-of-microservices/
5) https://github.com/dj-niobium/pact-testing