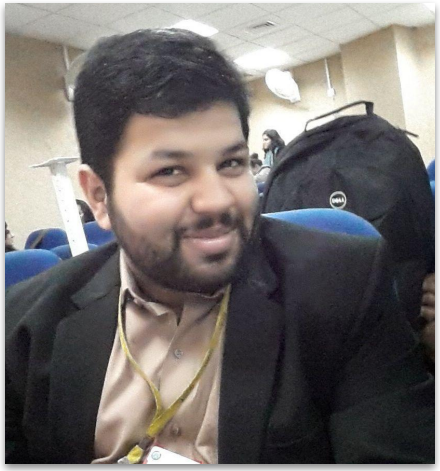
Why you should contract test your Microservices

Agenda

- Introduction to Microservices
- How Contract Testing Works
- Comparison with Traditional testing
- Introduction to Pact.io
- Closing Thoughts

whoami

- 🤓🤓 Software Engineer @ Arbisoft, Lahore, Pakistan
- Python/Django
- Contractor/Service provider as part of Arbisoft
  - edX/2U
- Internal products development



Syed Dawoud Sheraz

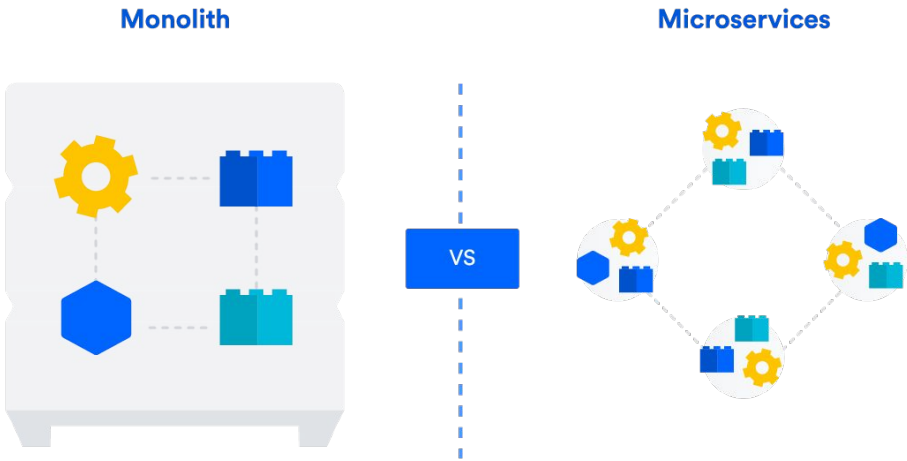
DawoudSheraz

@dawoud.sheraz

syed-muhammad-dawoud-sheraz-ali

Microservices

- Highly maintainable and testable
- Loosely coupled
- Independently deployable
- Owned by small teams



Microservices Architecture Industry Examples

Pros vs Cons

- Continuous Delivery
- Small in size
- Fault isolation/tolerance
- Commitment to a tech-stack

- Increased complexity of ***Communication***
- Infrastructure management
- **Integration Testing Challenges**

https://twitter.com/elonmusk/status/1632810081497513993?s=20

Contract Testing

“Contract tests assert that **inter-application messages** conform to a **shared understanding** that is **documented in a contract**. Without contract testing, the only way to ensure that applications will **work correctly together** is by using expensive and brittle integration tests.” - Pact.io

Contract Testing

- “Contract testing is a technique for testing an integration point by checking each application in isolation to ensure the messages it sends or receives conform to a shared understanding that is documented in a "contract".” - Pact.io
- Message types
  - HTTP communication → HTTP requests and responses
  - Queue → Events

```
{
  "name": "Dummy",
  "age": 30,
  "city": "Barcelona",
  "gender": "Male",
  "active": true,
  "country": "Spain"
}
```

Provider API response

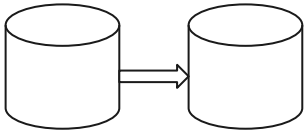
```
{
  "name": "Dummy",
  "age": 30
}
```

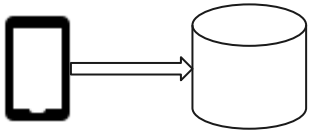
Consumer requirement

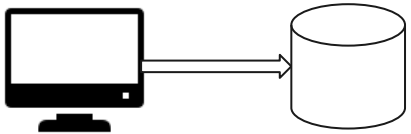
Contract tests will only assert the name and age fields

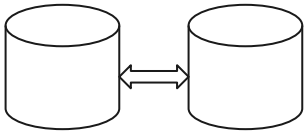
Important Terms

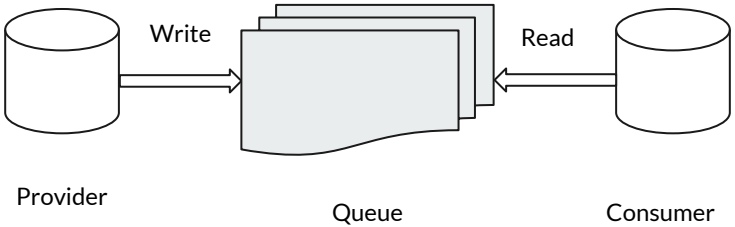
- Consumer: An application/service that uses data/functionality from another service(s)
  - HTTP → Application that initiates the request
  - Queues → Application reading from queue
  - Events → Application receiving the event
- Provider: An application/service that provides the data/functionality for another service(s)
  - HTTP → Application that returns the response
  - Queues → Application writing to the queue
  - Events → Application sending the event


Consumer
Provider


Consumer
Provider


Consumer
Provider


Consumer
Provider
Provider
Consumer


Write
Read
Provider
Queue
Consumer

Important Terms

- Interaction (HTTP)
  - An expected request by the consumer
  - A minimal expected response by the provider containing **only the data that consumer requested**
- Contract
  - Collection of interactions between a provider and a consumer
  - Each interaction in the contract is unique and independent

Consumer Driven Contract Testing

1. Contract/Pact is created by “Consumer” application that contains the “interactions” with a “Provider”application
2. Consumer generated contract is verified against provider by replaying the interactions against Provider

Generating the Contract(s): Consumer Tests

- Written as part of Consumer unit tests
- Interactions are registered when writing contract tests against a Mock Provider service
- During execution, unit tests emit a real request to Mock Provider
- Mock Provider compares the incoming request with registered request in interaction
- Upon match, return the registered response

Generating the Contract(s): Consumer Tests

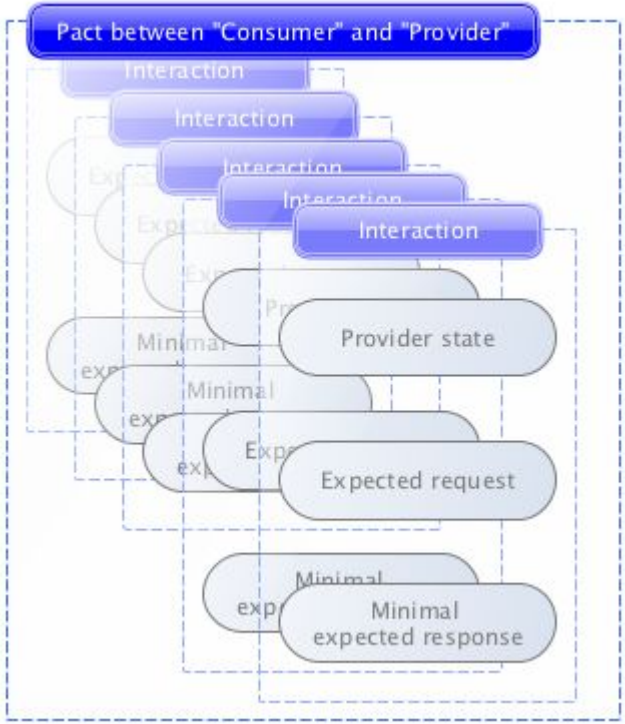
Consumer test

- Provided an User account exists
- Upon receiving a request for account
- With request GET /api/account/1
- Respond with {“account_owner”: “Dummy”}

Interaction

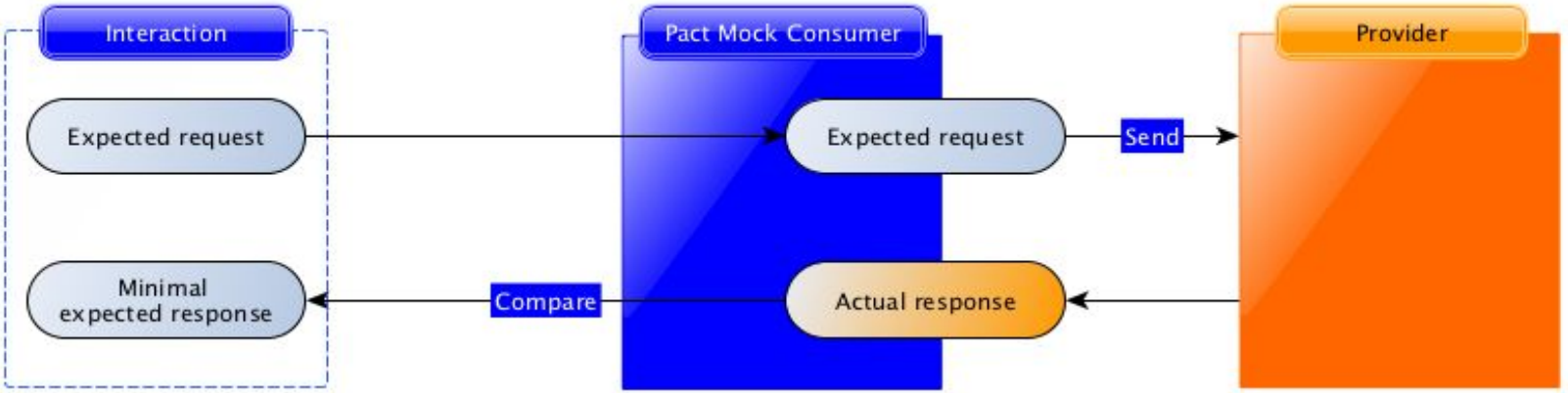
- If an user account exists
- GET /api/account/1
- Return {“account_owner”: “Dummy”}

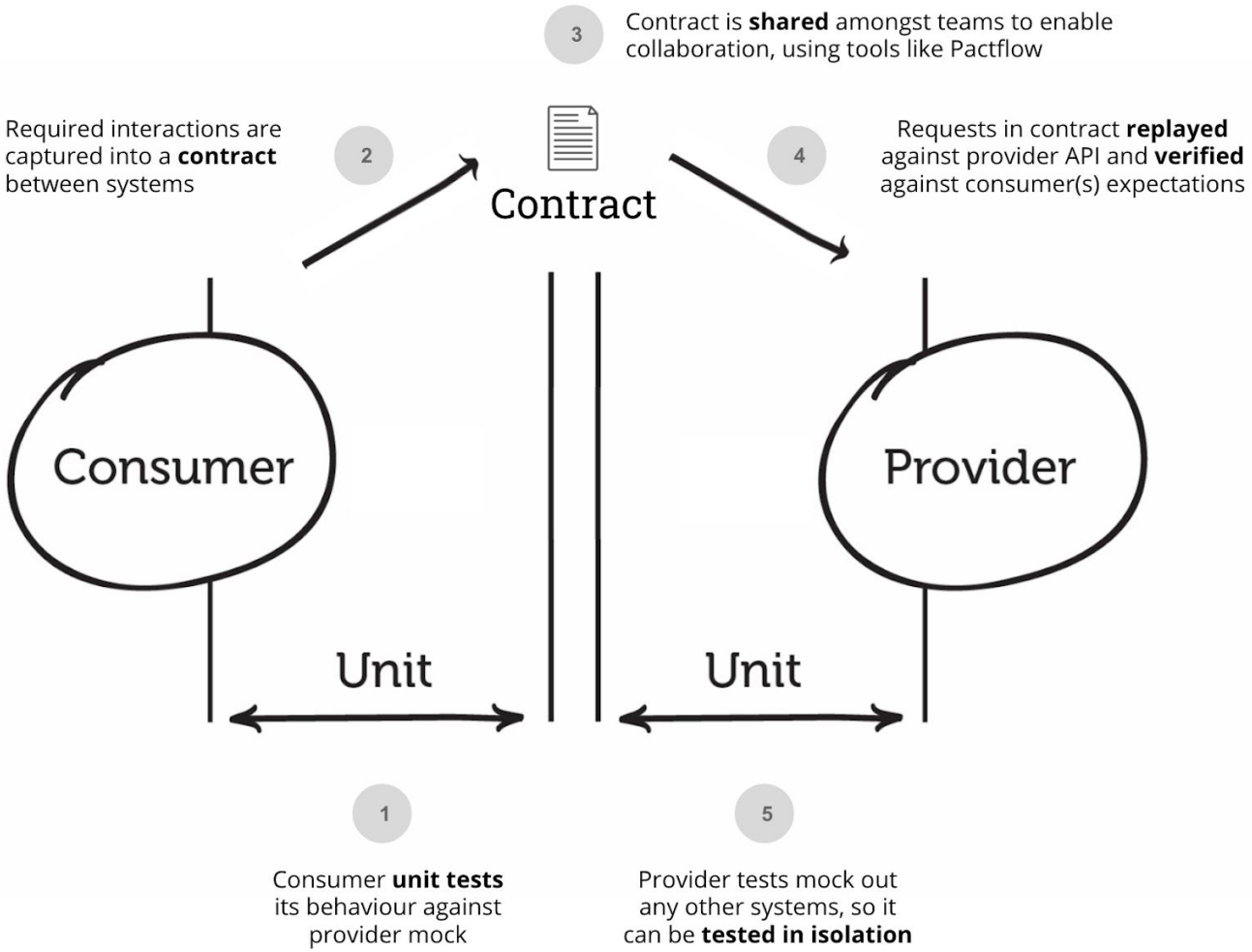
Consumer Test Output



Verifying the Contract: Provider Verification

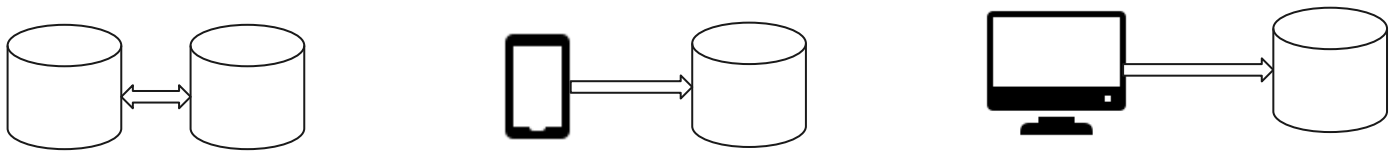
- Replay each interaction from contract against an actual provider
- Compare the response from provider with the expected minimal response in interaction
- Contract will be verified once every interaction is replayed successfully


Interaction
Expected request
Minimal
expected response
Pact Mock Consumer
Expected request
Actual response
Compare
Send
Provider


3
Contract is **shared** amongst teams to enable collaboration, using tools like Pactflow
Required interactions are captured into a **contract** between systems
2
Contract
4
Requests in contract **replayed** against provider API and **verified** against consumer(s) expectations
Consumer
Provider
Unit
Unit
1
Consumer **unit tests** its behaviour against provider mock
5
Provider tests mock out any other systems, so it can be **tested in isolation**

When to Contract Test

- 2 or more services, communicating with each other

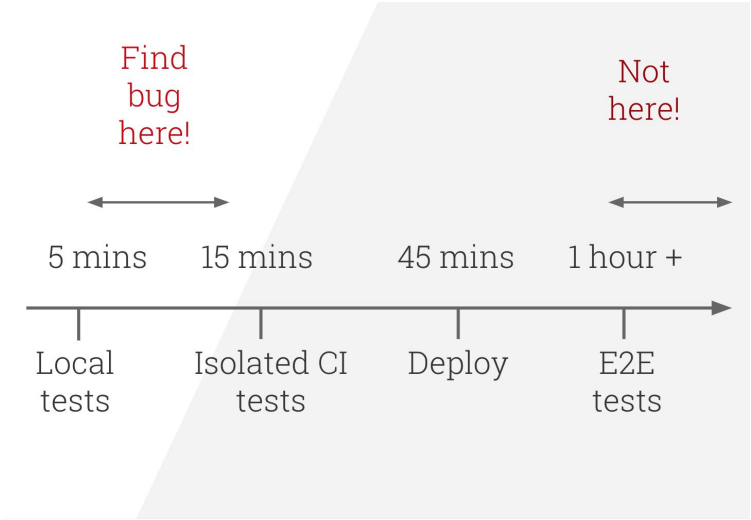


Need for Contract Testing in MicroServices

- Integration and communication → Backbones of Microservices architecture
- E2E/Integration testing requires infrastructure to be setup for “communication integration testing”
- Infrastructure setup is expensive in both time and resources
    - Can’t set the house on fire to test smoke alarm

Need for Contract Testing in MicroServices

- Find integration failures prior to deployments and deployment pipeline



Comparison with Unit & E2E Testing

Unit Tests

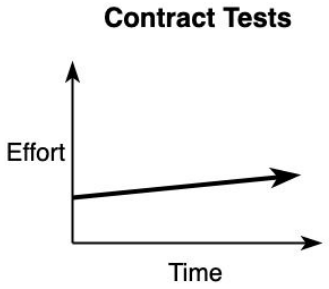
- Individual modules/units
- Verify the side-effects
- Testing each and every scenario
- Fast execution

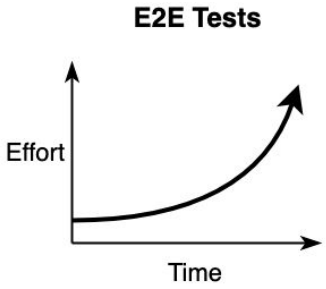
E2E Tests

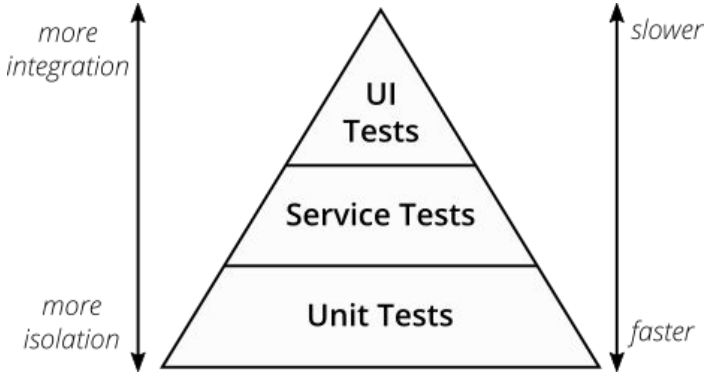
- Multiple applications/modules
- End-end user behaviors
- Time consuming
- Prone to flakiness

Comparison with Unit & E2E Testing

- Contract test sit in middle of testing pyramid
- Fast like unit tests
- Verify integration like E2E tests
- No Flakiness
- Effort to maintain does not scale exponentially







Pact.io

- Open source tool for testing HTTP and Event driven communications in Web apps
- Provides various tools and native libraries for writing consumer tests, provider verification and automating CI
    - Pact-python
    - Pact-js
    - Pact-ruby
    - Pact-swift
    - Pact-c++
    - Pact-broker
    - Pactflow.io

pact-python: consumer tests

```python
@pytest.fixture(scope='session')
def pact(request):
    pact = Consumer('pactflow-example-consumer-python').has_pact_with(
        Provider('pactflow-example-provider-python'), host_name=PACT_MOCK_HOST, port=PACT_MOCK_PORT,
        pact_dir="./pacts", log_dir="./logs")
    try:
        print('start service')
        pact.start_service()
        yield pact
    finally:
        print('stop service')
        pact.stop_service()

def test_get_product(pact, consumer):
    expected = {
        'id': "27",
        'name': 'Margharita',
        'type': 'Pizza'
    }

    (pact
     .given('a product with ID 10 exists')
     .upon_receiving('a request to get a product')
     .with_request('GET', '/product/10')
     .will_respond_with(200, body=Like(expected)))

    with pact:
        user = consumer.get_product('10')
        assert user.name == 'Margharita'
```

pact-js: consumer tests

```
const mockProvider = new Pact({
  consumer: 'pactflow-example-consumer',
  provider: process.env.PACT_PROVIDER
    ? process.env.PACT_PROVIDER
    : 'pactflow-example-provider'
});

describe('API Pact test', () => {
  describe('retrieving a product', () => {
    test('ID 10 exists', async () => {
      // Arrange
      const expectedProduct = {
        id: '10',
        type: 'CREDIT_CARD',
        name: '28 Degrees'
      };

      // Uncomment to see this fail
      // const expectedProduct = { id: '10', type: 'CREDIT_CARD', name: '28 Degrees', price: 30.0, newField: 22}

      mockProvider
        .given('a product with ID 10 exists')
        .uponReceiving('a request to get a product')
        .withRequest({
          method: 'GET',
          path: '/product/10',
          headers: {
            Authorization: like('Bearer 2019-01-14T11:34:18.045Z')
          }
        })
        .willRespondWith({
          status: 200,
          headers: {
            'Content-Type': 'application/json; charset=utf-8'
          },
          body: like(expectedProduct)
        });
      return mockProvider.executeTest(async (mockserver) => {
        // Act
        const api = new API(mockserver.url);
        const product = await api.getProduct('10');

        // Assert - did we get the expected response
        expect(product).toStrictEqual(new Product(expectedProduct));
        return;
      });
    });
```

Pact-js: provider verification

```
describe('Pact Verification', () => {
  let server;
  beforeAll(() => {
    server = setupServer();
  });
  afterAll(() => {
    if (server) {
      server.close();
    }
  });
  it('validates the expectations of any consumers, by specified consumerVersionSelectors', () => {
    if (process.env.PACT_URL) {
      console.log('pact url specified, so this test should not run');
      return;
    }
    // For 'normal' provider builds, fetch the the latest version from the main branch of each consumer, as specified by
    // the consumer's mainBranch property and all the currently deployed and currently released and supported versions of each consumer.
    // https://docs.pact.io/pact_broker/advanced_topics/consumer_version_selectors
    const fetchPactsDynamicallyOpts = {
      provider: 'pactflow-example-provider',
      consumerVersionSelectors: [
        { mainBranch: true },
        { deployed: true },
        { matchingBranch: true }
      ],
      pactBrokerUrl: process.env.PACT_BROKER_BASE_URL,
      // https://docs.pact.io/pact_broker/advanced_topics/pending_pacts
      enablePending: true,
      // https://docs.pact.io/pact_broker/advanced_topics/wip_pacts
      includeWipPactsSince: '2020-01-01'
    };

    const opts = {
      ...baseOpts,
      ...fetchPactsDynamicallyOpts,
      stateHandlers: stateHandlers,
      requestFilter: requestFilter
    };
    return new Verifier(opts).verifyProvider().then((output) => {
      console.log('Pact Verification Complete!');
    });
  });
});
```

Pact Broker

- Open source application developed by Pact maintainers
- Publish contracts and contract verification results
  - Versioning
  - Tagging
- Provides webhooks against actions:
  - Contract published
  - Provider verification published
  - Provider verification successful
  - Contract requiring verification published
- Easy to configure CI using Pact Broker

Pacts

Consumer	Provider	Latest pact published	Last verified
Foo	Animals	2 minutes ago	2 days ago
Foo	Bar	7 days ago	15 days ago ⚠
Foo	Hello World App	1 day ago	
Foo	Wiffles	less than a minute ago	7 days ago
Some other app	A service	26 days ago	less than a minute ago
The Android App	The back end	less than a minute ago	

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

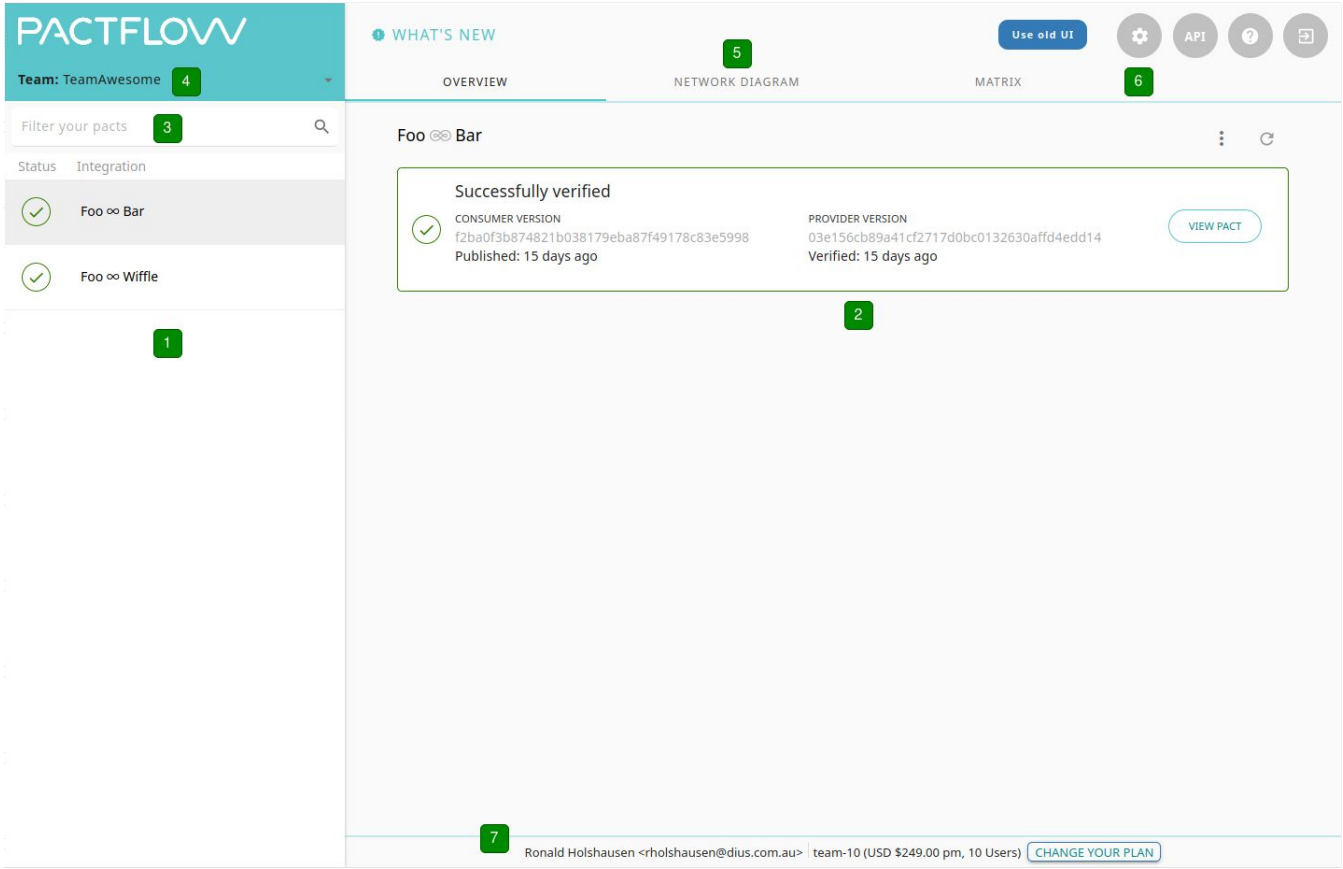
```
{
  "method": "get",
  "path": "/alligators/Mary",
  "headers": {
    "Accept": "application/json"
  }
}
```

Animal Service will respond with:

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

Pactflow.io

- Commercial Pact broker developed and maintained by Pact (not Open source)
- Important features provided
  - Better UX and UI
  - User account, role, and permissions management
  - Store secrets to use in Webhooks
  - API tokens
  - Audit logs of the changes
  - SSO



[Pactflow.io Dashboard]

Connecting it all

- Write consumer tests with desired pact native lib to generate pact
- Write provider verification against the generated pact
- Use pact-broker or Pactflow.io to:
  - Trigger provider flow when contract is updated
  - OR Verify the contracts whenever provider is updated

Closing Thoughts

- Contract Testing is not a substitute to good team level communication
- Contract testing is not functional testing of provider
- As a consumer
  - List down which services in the ecosystem are consumed
- As a provider
  - Verify which consumer applications consume the provider service
- Next steps
  - Communicate with respective team
  - Come to an understanding why contract testing will be helpful
  - Write consumer tests or provider verification
  - Develop CI flow

Closing Thoughts

- To learn more about Pact and how to use Pact for Contract Testing
  - Join Pact foundation slack
  - Get in touch with active community members
  - Or even provide technical open source support to pact-python or any other pact library

References

- 10 Companies that implemented the microservice architecture and paved the way for others
- https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith
- YOW! Perth 2019 - Beth Skurrie - Microservices. Test smarter, not harder
- YOW! 2017 Beth Skurrie - It's Not Hard to Test Smart: Delivering Customer Value Faster #YOW
- https://docs.pact.io/help/how_to_ask_for_help

References

- https://pactflow.io/oss/
- https://docs.pact.io/faq
- https://docs.pact.io/consumer/contract_tests_not_functional_tests
- https://pactflow.io/features/
- https://pactflow.io/blog/what-is-contract-testing/?utm_source=ossdocs&utm_campaign=convince_me_what_is
- https://docs.pact.io/faq/convinceme
- https://docs.pact.io/getting_started/how_pact_works
- https://docs.pact.io/pact_broker
- https://github.com/pactflow/example-consumer-python/blob/master/tests/consumer/test_products_consumer.py