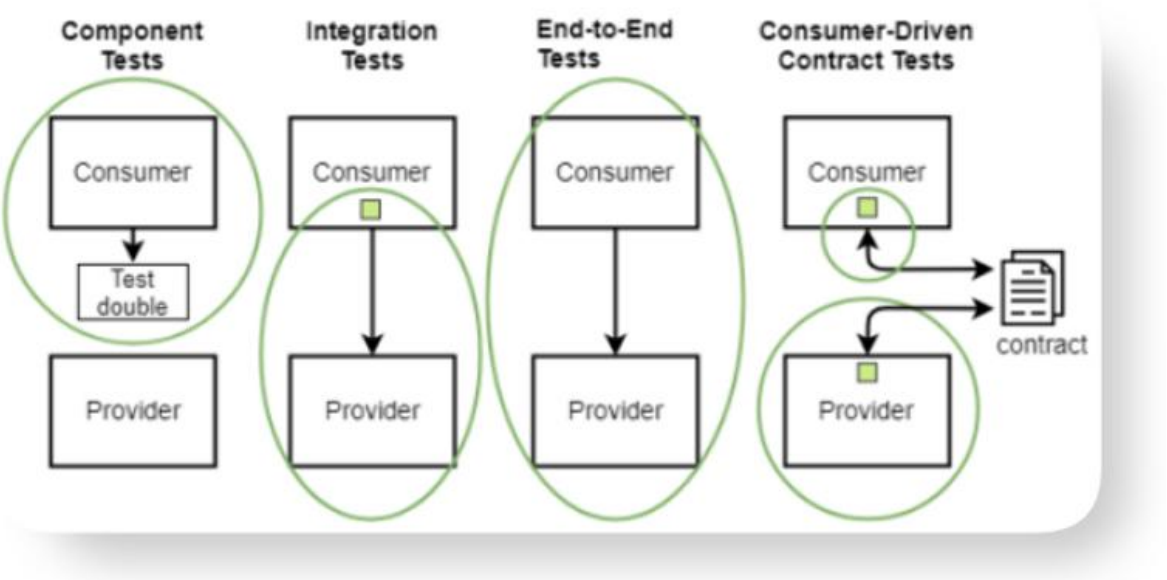
Where the Testing Pyramid lacks when it comes to testing microservices?

Initially, the testing pyramid worked well for testing monolith architecture. But when microservices came into picture, **unit tests** and E2E tests simply failed. Some major issues with the Martin Fowler's pyramid are:

➡️ **Testing for System-Level Confidence:** Component level tests are comfortable and fast to implement. But the confidence slowly fades away when the number of services, endpoints, and integrations keeps on growing when new features are implemented. While it's important to test individual components, the real test comes when everything is integrated.

➡️ **Hurdles with High-level testing:** End-to-end tests are slow, prone to errors, hard to debug, and non-deterministic in general. Sometimes teams even avoid running them because of the high effort. They fail to data errors, and no one wants to spend time debugging them. It is not rewarding to debug and fix errors that are only related to the testing environment and not to the actual features of the system.



And that's where devs and QA folks started exploring other approaches for microservices testing. This is where contract testing came forward as a tailored solution for microservices. It offers a simpler and more manageable way to ensure these services talks and performs as decided.

What is Contract Testing?

Contract testing is a way to test integrations between services and ensure that all the integrations are still working after new changes have been introduced to the system. This allows for faster, more focused testing since only the interactions are tested.

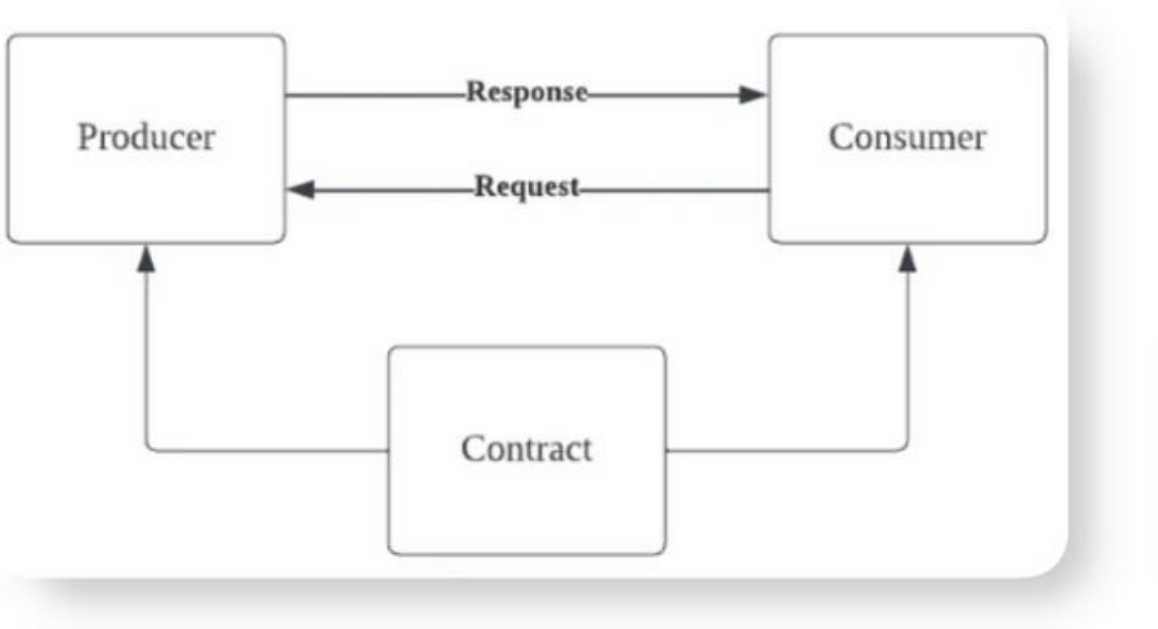
💡 **The main idea is that when an application or a service (consumer) consumes an API provided by another service (provider), a contract is formed between them.**

Contract testing breaks the testing boundaries between the services when compared to the component tests.

This means that the services are no longer fully isolated from each other. The services are not directly connected either, like it happens with end-to-end tests. Instead, they are indirectly connected, and they communicate with each other using the contracts as a tool.

How Does Contract Testing Works?

Contract testing involves establishing and validating an agreement between two parties: the "Provider" (service owner) and the "Consumer" (service user).



There are two main approaches to contract testing: **consumer-driven and provider-driven.**

👉Consumer-driven Contract Testing:

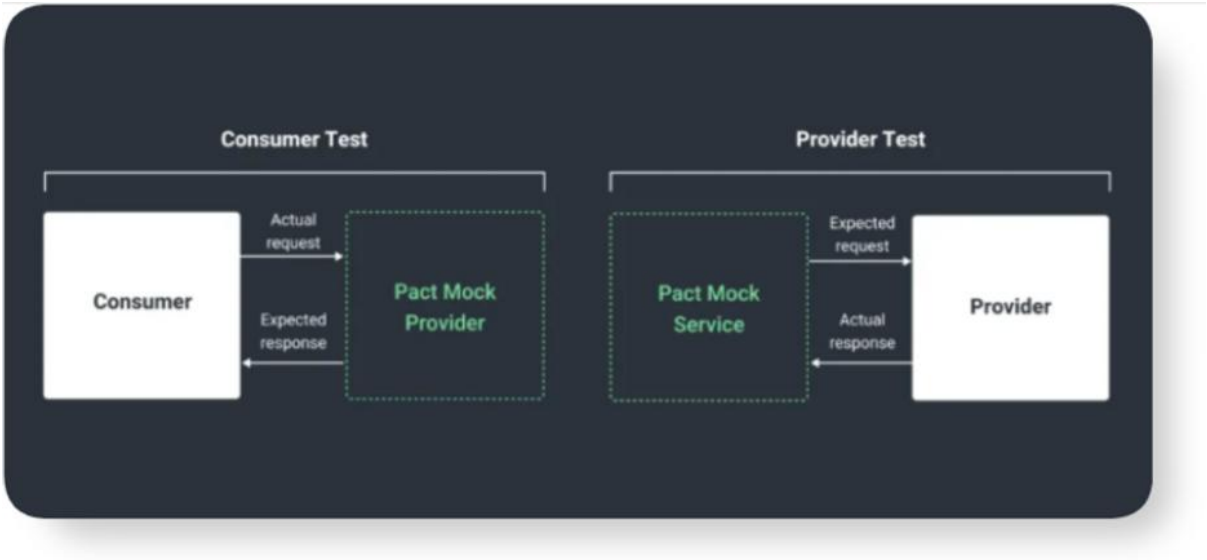
Consumer-driven contract testing is an approach where the consumer of a service takes the lead in defining the expected behavior of the service they depend on. They specify the contracts (expectations) that the service provider should meet, and then run tests to ensure that the provider complies with these contracts.

This approach puts consumers in control of their service dependencies and helps prevent unexpected changes or regressions in the service they rely on.

👉Provider-driven Contract Testing:

Provider-driven contract testing is initiated by the service provider. In this approach, the provider defines the contracts that consumers should adhere to when interacting with the service. The provider sets the expectations and provides these contracts to consumers.

Consumers, in turn, run tests to ensure that their usage of the service complies with the contracts specified by the provider. This approach allows the service provider to have a more proactive role in maintaining the integrity and stability of the service.



👉Working of Consumer-driven Contract Testing:

➡️The contract contains information about how the consumer calls the provider and what is being used from the responses.

➡️As long as both of the parties obey the contract, they can both use it as a basis to verify their sides of the integration. The consumer can use it to mock the provider in its tests.

➡️ The provider, on the other hand, can use it to replay the consumer requests against its API. This way the provider can verify that the generated responses match the expectations set by the consumer.

➡️ With consumer-driven contracts, the provider is always aware of all of its consumers. This comes as a side product when all the consumers deliver their contracts to the provider instead of consumers accepting the contracts offered by the provider.

Benefits of Contract Testing

➡️ **Maintenance:** They are easier to maintain as you don't need to have an in-depth understanding of the entire ecosystem. You can write and manage tests for specific components without being overwhelmed by the complexity of the entire system.

➡️ **Debugging and Issue Resolution:** Contract tests simplify the debugging process. When a problem arises, you can pinpoint it to the specific component being tested. This means you'll often receive precise information like a line number or a particular API endpoint that is failing, making issue resolution more straightforward.

➡️ **Local Bug Discovery:** Contract tests are excellent at uncovering bugs on developer machines during the development process. This early detection helps developers catch and fix issues before pushing their code, contributing to better code quality and reducing the chances of defects reaching production.

➡️ **Integration Failures:** If discrepancies arise during either phase of Contract testing, it signals a need for joint problem-solving between the Consumer and Provider.

Use-cases of Contract Testing

Contract testing is a useful way to make sure microservices and APIs work well together. But it's not the best choice for all testing situations. Here are some cases where contract testing is a good idea:

✅ **Microservices Communication Testing:**

Use Case: In a microservices architecture, different services need to communicate with each other. Contract testing ensures that these services understand and meet each other's expectations.

```
# Consumer Service Contract Test
def test_consumer_service_contract():
    contract = {
        "request": {"path": "/user/123", "method": "GET"},
        "response": {"status": 200, "body": {"id": 123, "name": "John"}}
    }
    response = make_request_to_provider_service(contract)
    assert response.status_code == 200
```

```
# Provider Service Contract Implementation
def make_request_to_provider_service(contract):
    # Code to handle the request and provide the expected response
    pass
```

✅ **API Integration/ Third-Party Integration Testing:**

Use Case: When integrating with external APIs, contract testing helps ensure that the application interacts correctly with all the third-party APIs, preventing any compatibility issues and ensure that the code is reliable and secure.

```
# Contract Test for External API Integration
def test_external_api_contract():
    contract = {
        "request": {"path": "/products/123", "method": "GET"},
        "response": {"status": 200, "body": {"id": 123, "name": "Product A"}}
    }
    response = make_request_to_external_api(contract)
    assert response.status_code == 200

# Code for Making Requests to External API
def make_request_to_external_api(contract):
    # Code to send the request to the external API and handle the response
    pass
```

✅ **UI Component Interaction Testing:**

Use Case: Contract testing can be applied to UI components, ensuring that interactions between different parts of a web application, like front-end and back-end, work as expected.

```
// Front-End UI Component Contract Test
it('should display user data when provided', () => {
   const userData = { id: 123, name: 'Alice' };
   const component = render(<UserProfile data={userData} />);
   const userElement = screen.getByText(/Alice/);
   expect(userElement).toBeInTheDocument();
});

// Back-End API Contract Implementation
app.get('/api/user/123', (req, res) => {
   res.status(200).json({ id: 123, name: 'Alice' });
});
```

Difference Between Contract Testing and Integration Testing

Contract testing and integration testing are sometimes mistaken for one another; despite sharing a common end goal, they diverge significantly in their approaches.

Without contract testing, the only way to ensure that applications will work correctly together is by using expensive and brittle integration tests. Pact.io introduction

	Contract testing	Integration testing
Scope	Focuses on verifying interactions at the boundary of an external service. It ensures that a service's output conforms to certain expectations and that it correctly handles input from another service. The primary concern is the "contract" or agreement between services.	Addresses the intera components or syste different component as expected.

Test Depth	Is concerned with the correctness of interactions, not the internal workings of each service. It verifies if a service lives up to its "contract" or promised behavior.	Goes beyond just int the integrated syste correct data exchang
Test Maintenance	Contracts can be stable and act as a form of documentation. If teams adhere to the contract, changes in implementation shouldn't necessitate changes in the test.	Tends to be more br often break integrati end-to-end in nature
Isolation	Uses mocked providers, allowing for testing in isolation. A consumer can be tested against a contract without having a running instance of the provider and vice versa.	Requires a more inte interaction between usually need to be u
Feedback Loop	Provides quicker feedback since it tests against contracts and doesn't require setting up the entire integrated environment.	Might have a slower end-to-end tests, be and require more se
Purpose	Aims to give confidence that services fulfill their promises to their consumers. It's particularly useful in a microservices environment where teams develop services independently.	Ensures that when d together, they opera catching problems th contract tests.

Contract testing is about adhering to agreements between services, while integration testing is about achieving seamless operation when components are combined. Both types of testing are complementary and important in their own right.

Tools to Perform Contract Testing

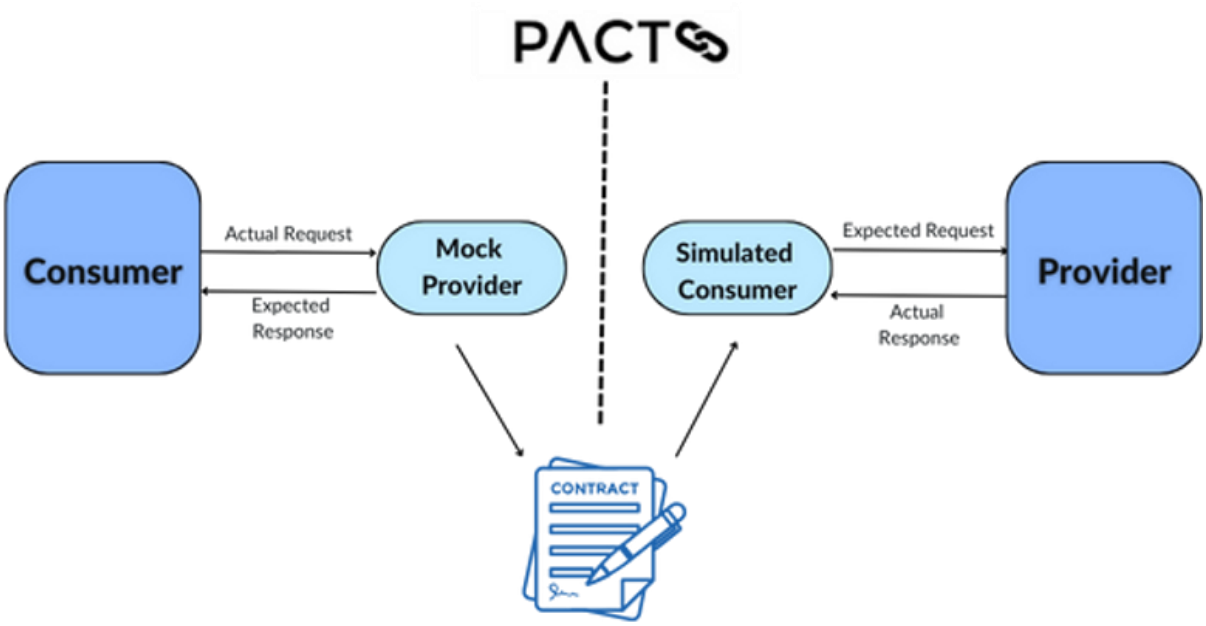
Numerous tools exist in the market for conducting contract testing on APIs and microservices. Pact and Spring Cloud Contract are two of the most prominent tools in the realm of contract testing, widely recognized for their specific features and capabilities:

Let’s discuss the two of them:

➡️ **PACT:**

PACT stands out in the contract testing domain for its user-friendly interface and flexibility across various programming languages. Pact enables both consumers and providers to define and verify their contracts effectively.

It operates on a consumer-driven approach, meaning the consumer code generates the contract. This approach encourages consumer specifications to be fully considered in the contract, ensuring that the provided service meets these requirements.



Pact's versatility extends to support for numerous programming languages, including Ruby, JavaScript, and JVM languages, catering to a wide range of development environments.

In our previous **contract testing** article, we covered the basics of what contract testing is and how it works. Now, in this blog post, we'll introduce you to a popular tool for contract testing—PACT Contract testing.

What is PACT contract testing?

Let's understand why PACT contract testing became essential through a real team retrospective about a production failure.

Q: Why did our user profile feature break in production when the Auth service team said they only made a "minor update"?

***A:** We were consuming their /user endpoint expecting the response to always include a phone field, but they changed it to optional without telling us.*

Q: But didn't we have unit tests covering the user profile logic?

***A:** Yes, but our unit tests were mocking the Auth service response with the old structure. Our mocks had the phone field as required, so our tests passed even though the real service changed.*

Q: Why didn't integration tests catch this?

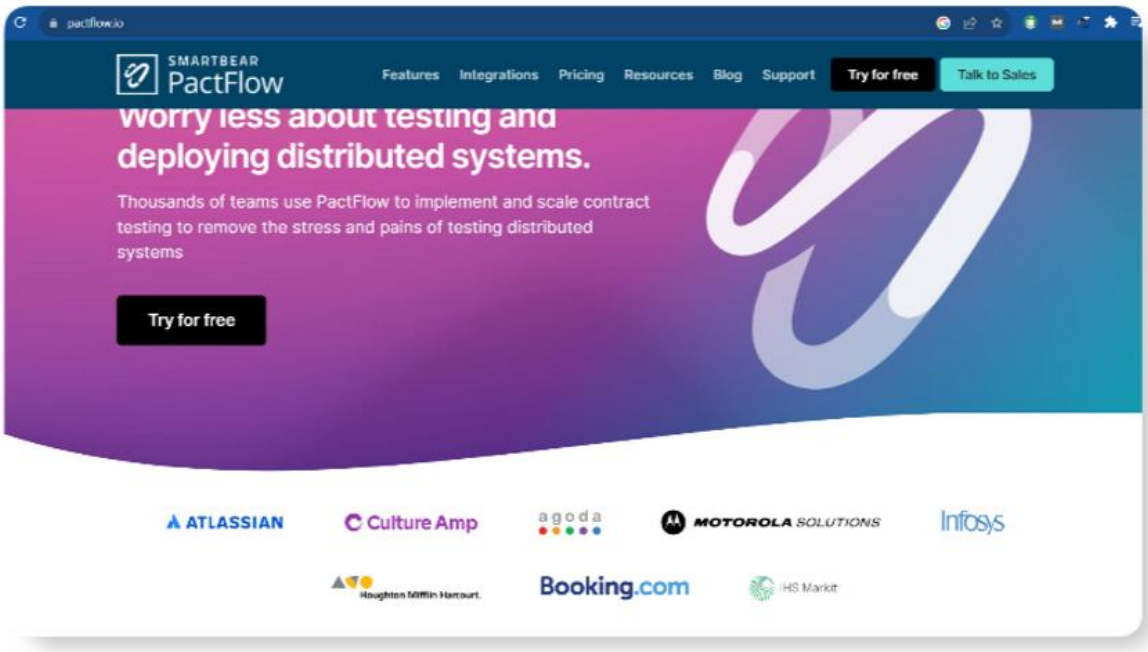
***A:** We only run full integration tests in staging once a week because they're slow and flaky. By then, the Auth team had already deployed to production and moved on to other features.*

Q: How could we have prevented this?

***A:** If we had a **contract** between our services - something that both teams agreed upon and tested against - this wouldn't have happened. That's exactly what PACT contract testing solves.*

Contract tests combine the lightness of unit tests with the confidence of integration tests and should be part of your development toolkit.

PACT is a code-based tool used for testing interactions between service consumers and providers in a microservices architecture. Essentially, it helps developers ensure that services (like APIs or microservices) can communicate with each other correctly by validating each side against a set of agreed-upon rules or "contracts".



Here's what PACT does in a nutshell:

- It allows developers to define the expectations of an interaction between services in a format that can be shared and understood by both sides.

- PACT provides a framework to write these contracts and tests for both the consuming service (the one making the requests) and the providing service (the one responding to the requests).

PACT has a lot of manual effort involved in generating the test cases, move beyond that and adopt in a fast-performing approach that auto generates test cases based on your application's network traffic. Curious to know more?

- When the consumer and provider tests are run, PACT checks whether both sides adhere to the contract. If either side does not meet the contract, the tests fail, indicating an issue in the integration.

- By automating these checks, PACT helps teams catch potential integration issues early and often, which is particularly useful in CI/CD environments.

So, PACT focuses on preventing breaking changes in the interactions between services, which is critical for maintaining a reliable and robust system when multiple teams are working on different services in parallel.

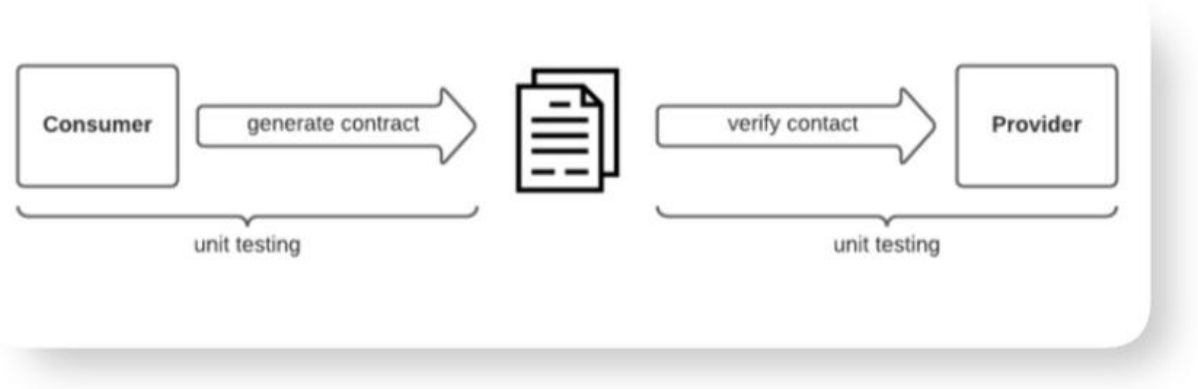
Importance of PACT Contract Testing

➡️ **PACT reduces the complexity of the environment that is needed to verify integrations**, as well as isolates changes to the specific interaction between services. This prevents cascading failures and simplifies debugging.

*Managing different environments for different purposes is definitely a tedious task to do, companies like **Zoop, Skaud, PayU, Nykaa** etc, uses a smart approach that takes away all the need to manage dedicated environments, allowing you to focus on more important things.*

➡️ **Decoupling for Independence:** PACT enables microservices to thrive on decoupled, independent development, testing, and deployment, ensuring adherence to contracts and reducing compatibility risks during the migration from monoliths to microservices.

➡️ **Swift Issue Detection:** PACT's early identification of compatibility problems during development means faster feedback, with precise, interaction-focused tests that expedite feedback and streamline change signoffs.



➡️ **Enhanced Collaboration and Confidence:** Clear, shared service interaction contracts reduce misunderstandings, fostering collaboration and developer confidence in releasing changes without breaking existing contracts.

➡️ **Living Documentation:** Pact contracts serve as dynamic, clear-cut documentation, simplifying developers' comprehension of integration points.

➡️ **Reduced Service Outages:** Pact contract tests swiftly highlight provider service changes that break consumer expectations, facilitating quick identification and resolution of disruptive modifications.

How does Pact implement contract testing?

Pact implements contract testing through a process that involves both the consumer and the provider of a service, following these steps:

➡️ **Consumer Testing:**

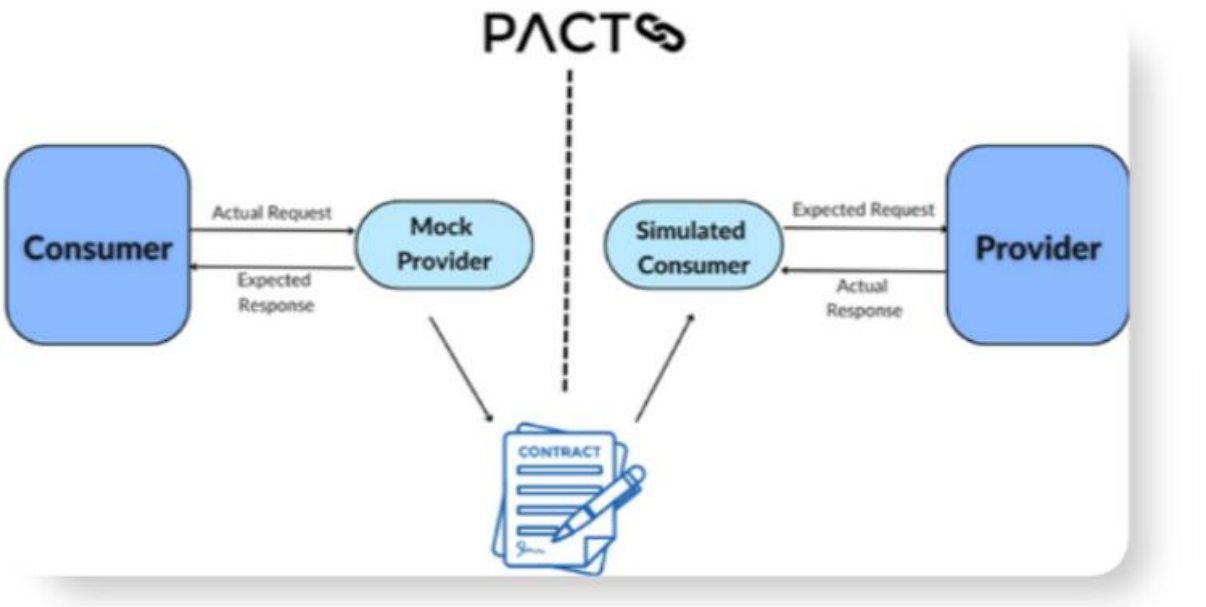
- The consumer of a service (e.g., a client application) writes a test for the expected interaction with the provider's service.

- While writing this test, Pact stubs out the actual provider service and records the expectations of the consumer—what kind of request it will make and what kind of response it expects—into a Pact file, which is a JSON file acting as the contract.

- The consumer test is run with the Pact mock service, which ensures the consumer can handle the expected response from the provider.

➡️ **Pact File Generation:**

- When the consumer tests pass, the Pact file (contract) is generated. This file includes the defined requests and the expected responses.



➡️ **Provider Verification:**

- The provider then takes this Pact file and runs it against their service to verify that the service can meet the contract's expectations.

- The provider's tests take each request recorded in the Pact file and compare it against the actual response the service gives. If they match, the provider is considered to be in compliance with the contract.

➡️ **Publishing Results:**

- Results of the provider verification can be published to a Pact Broker, which is a repository for Pact files. This allows for versioning of contracts and tracking of the verifications.

- Both the consumer and the provider use the Pact Broker to publish and retrieve Pact files. It helps to ensure that both parties in the service interaction are always testing against the latest contract.

➡️ **Continuous Integration:**

- Pact is often integrated into the CI/CD pipeline. Whenever changes are made to the consumer or provider, the corresponding contract tests are automatically run.

- This helps in identifying any breaches in the contract immediately when a change is made, ensuring that any integration issues are caught and addressed early in the development lifecycle.

➡️ **Version Control:**

- Pact supports semantic versioning of contracts, which helps in managing the compatibility of interactions between different versions of the consumer and provider services.

By automating the creation and verification of these contracts, Pact helps maintain a reliable system of independent services by ensuring they can communicate as expected, reducing the likelihood of integration issues in a microservices architecture.

How to perform Pact Contract Testing?

Now we all know that **Pact is a code-first tool for testing HTTP and message integrations** using contract tests. Instead of testing the internal details of each service, PACT contract testing focus on the "contract" or the agreement between services on how their APIs should behave.

For this example, we have created a hypothetical scenario where a client app expects to fetch user data from a service.

Step 1: Define the Consumer Test

In the consumer service, you would write a test that defines the expected interaction with the provider's API.

Step 2: Run the Consumer Test

When this test is executed, the **pact** context manager starts the mock service, and the defined interaction is registered with it. Then, the test makes a request to the mock service, which checks that the request matches the registered interaction. If it does, it responds with the predefined response.

Step 3: Generate the Contract (Pact File)

If all assertions pass and the test completes successfully, Pact will generate a **.json** file representing the contract. This file is then used by the provider to verify that their API meets the expectations defined by the consumer.


```
{
  "consumer": {
```

```
    "name": "ConsumerService"
  },
  "provider": {
    "name": "ProviderService"
  },
  "interactions": [
    {
      "description": "a request for user id 1",
      "providerState": "a user with id 1 exists",
      "request": {
        "method": "GET",
        "path": "/user/1"
      },
      "response": {
        "status": 200,
        "body": {
          "id": 1,
          "name": "John Doe",
          "email": "john.doe@example.com"
        }
      }
    }
  ],
  "metadata": {
    "pactSpecification": {
      "version": "2.0.0"
    }
  }
}
```

Step 4: Verify the Provider with the Pact File

The provider's test suite would use this **.json** Pact file to ensure their service can handle the requests and send the expected responses. The provider doesn't necessarily need to know the internals of the consumer; it just needs to satisfy the contract as outlined in the Pact file.

The **Verifier** uses the pact file to make requests to the actual provider service and checks that the responses match the contract. If they do, the provider has met the contract, and you can be confident that the provider and consumer can communicate correctly.

Problems with PACT

If your primary goal is keeping contract testing simple and with lesser overheads, PACT may not be the ideal tool.

PACT contract testing has become very popular among teams off late given its simplicity and effectiveness. But it comes with its own set of challenges, making adoption at scale a challenge.

It’s not always straightforward, it demands a considerable amount of manual effort and time.

There are some obvious challenges in getting started and also the manual intervention in contract maintenance doesn’t make it the perfect fit for testing microservices alone.

👉**Complex setup and high maintenance**

👉**CI/CD Pipeline Integration Challenges**

👉**High Learning Curve**

👉**Consumer Complexity**

👉**Test Data Management**

Let’s get started with all of them, one-by-one.

1. Lots of Manual Effort Still Needed

Pact contracts need to be maintained and updated as services evolve. Ensuring that contracts accurately reflect real interactions can become challenging, especially in rapidly changing environments. Ensuring that contracts accurately reflect the expected interactions can become complex, especially when multiple consumers are involved.

Any time teams (especially producers) miss updating contracts, consumers start testing against incorrect behaviors which is when critical bugs start leaking into production.

➞**Initial Contract Creation**

Writing the first version of a contract involves a detailed understanding of both the consumer's expectations and the provider's capabilities. Developers must manually define the interactions in test code.

```
# Defining a contract in a consumer test
@pact.given('user exists')
@pact.upon_receiving('a request for a user')
@pact.with_request(method='GET', path='/user/1')
@pact.will_respond_with(status=200, body={'id': 1, 'name': 'John Doe'})
def test_get_user():
    # Test logic here
```

This change must be communicated and agreed upon by all consumers of the API, adding coordination overhead.

➞ Maintaining Contract Tests

The test suite for both the consumer and the provider will grow as new features are added. This increased test suite size can make maintenance more difficult.

Each function represents a new contract or a part of a contract that must be maintained.

```
# Over time, you may end up with multiple contract tests
def test_get_user():
    # ...

def test_update_user():
    # ...

def test_delete_user():
    # ...
```

2. Testing Asynchronous Patterns

Pact supports non-HTTP communications, like message queues or event-driven systems, but this support varies by language and can be less mature than HTTP testing.

```
// A JavaScript example for message provider verification
let messagePact = new MessageProviderPact({
  messageProviders: {
    'a user created message': () => Promise.resolve({ /*...message contents...*/ }),
  },
```

```
  // ...
});
```

This requires additional understanding of how Pact handles asynchronous message contracts, which might not be as straightforward as HTTP.

3. Consumer Complexity

In cases where multiple consumers interact with a single provider, managing and coordinating contracts for all consumers can become intricate.

➞ Dependency Chains

Consumer A might depend on Consumer B, which in turn depends on the Provider. Changes made by Provider could potentially impact both Consumer A and the Consumer B. This chain of dependencies complicates the contract management process.

💡 Let's understand this with an example:

Given Services:

- **Provider**: User Management API.

- **Consumer B**: Profile Management Service, depends on the Provider.

- **Consumer** A: Front-end Application, depends on Consumer B.

Dependency Chain:

- `**Consumer A**` depends on `**Consumer B**`, which in turn depends on the `Provider`.

Change Scenario:

- The `**Provider**` adds a new mandatory field `**birthdate**` to its user data response.

- `**Consumer B**` updates its contract to incorporate `**birthdate**` and exposes it through its endpoint.

- `**Consumer A**` now has a failing contract because it doesn't expect `birthdate` in the data it receives from `**Consumer B**`.

Impact:

- `**Consumer A**` needs to update its contract and UI to handle the new field.

- `**Provider**` needs to coordinate changes with both the `**Consumer B**` and `**Consumer A**` to maintain contract compatibility.

- The `**Provider**` must be aware of how its changes affect downstream services to avoid breaking their contracts.

</aside>

➞ Coordination Between Teams

When multiple teams are involved, coordination becomes crucial. Any change to a contract by one team must be communicated to and accepted by all other teams that are consumers of that API.

Communication overhead example

Team A sends a message to Team B:

"We've updated the contract for the /user endpoint, please review the changes."

This communication often happens outside of Pact, such as via team meetings, emails, or chat systems.

Ensuring that all consumer teams are aware of contract changes and aligned on the updates can require effective communication channels and documentation.

4. Test Data Management

Test data management in Pact involves ensuring that the data used during contract testing accurately represents real-world scenarios while maintaining consistency, integrity, and privacy. This can be a significant challenge, particularly in complex microservices ecosystems. The problems that might arise would be:

➞ Data Generation

Creating meaningful and representative test data for all possible scenarios can be challenging. Services might need specific data states to test different interactions thoroughly.

➞ Data Synchronization

PACT tests should use data that accurately reflects the behavior of the system. This means that the test data needs to be synchronized and consistent across different services to ensure realistic interactions. Mismatched or inconsistent data can lead to false positives or negatives during testing.

Example: *If the consumer's Pact test expects a user with ID 1, but the provider's test environment doesn't have this user, the verification will fail.*

➞ Partial Mocking Limitations

Because Pact uses a mock service to simulate provider responses, it's possible to get false positives if the provider's actual behavior differs from the mocked behavior. This can happen if the provider's implementation changes without corresponding changes to the contract.