

# Testing Microservices Part 1

Wiwat Vatanawood

Duangdao Wichadakul

Nuengwong Tuaycharoen

Pittipol Kantavat



# The Chapter covers

- Effective testing strategies for microservices
- Using mocks and stubs to test a software element in isolation
- Using the test pyramid to determine where to focus testing efforts
- Unit testing the classes inside a service



# Agenda

- Overview of testing microservice
- Testing strategies for microservice architectures
- The challenge of testing microservices
- The deployment pipeline
- Writing unit tests for a service



# An Overview of Testing Microservice

- Testing is primarily an activity that happens after development.
- Why we need automated testing?
  - Manual testing is extremely inefficient — You should never ask a human to do what a machine can do better.
  - Manual testing is done far too late in the delivery process — A much better approach is for developers to write automated tests as part of development.
- Your development workflow should be: edit code → run tests (ideally with a single keystroke) → repeat

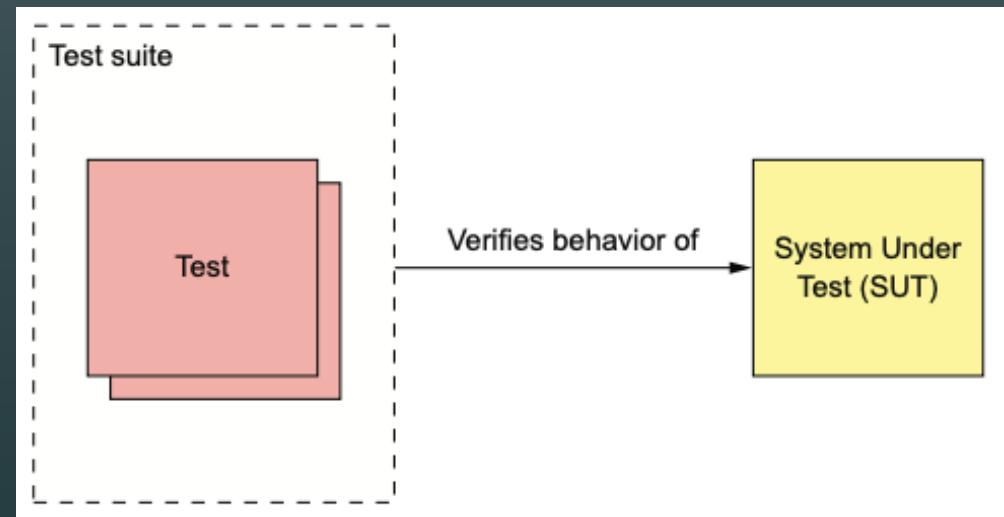
# Agenda

- Overview of testing microservice
- Testing strategies for microservice architectures
- The challenge of testing microservices
- The deployment pipeline
- Writing unit tests for a service



# Testing strategies for microservice architectures

- The goal of a test is to verify the behavior of the **System Under Test (SUT)**
- An SUT might be as small as a class or as large as an entire application.
- A collection of related tests are called a “**Test Suite**”.

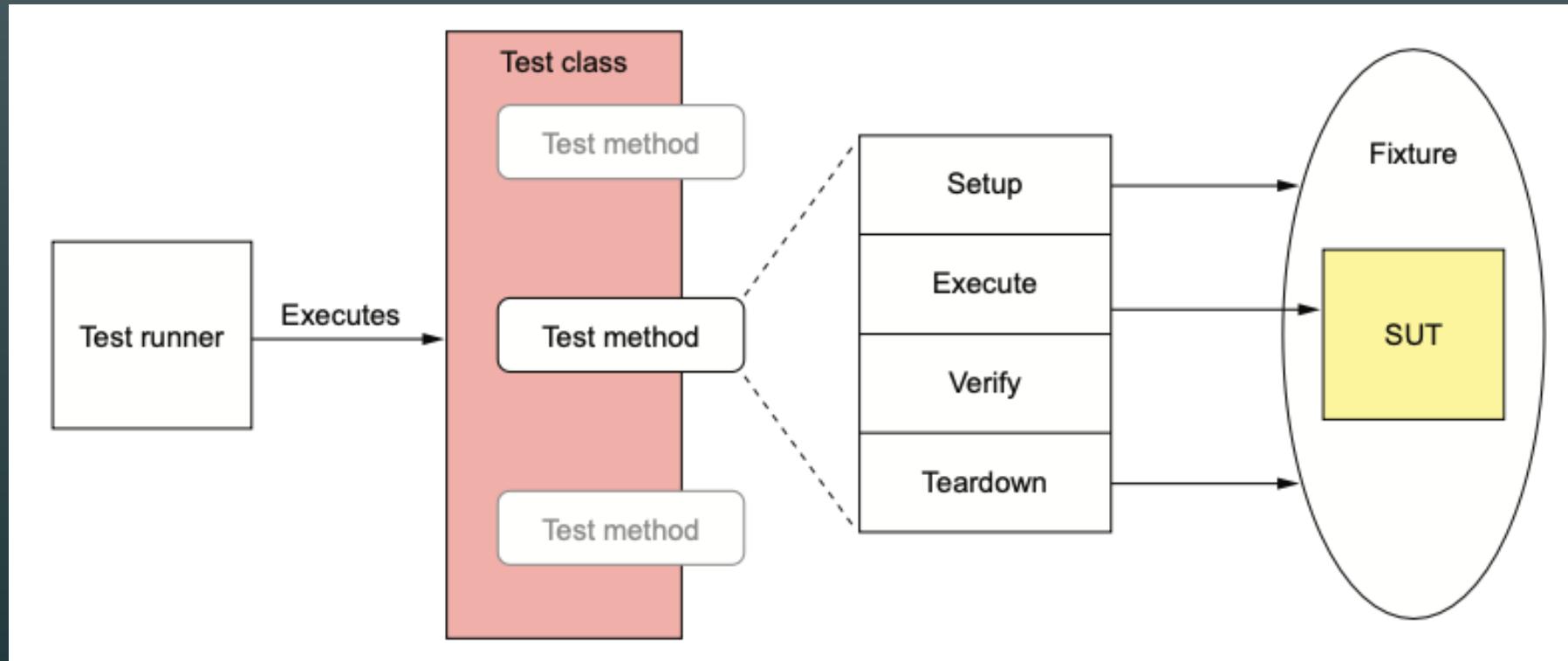


# Writing Automated Tests (1)

- An automated test typically consists of four phases:
  1. **Setup** — Initialize the test fixture, which consists of the SUT and its dependencies, to the desired initial state.
  2. **Exercise** — Invoke the SUT—for example, invoke a method on the class under test.
  3. **Verify** — Make assertions about the invocation's outcome and the state of the SUT.
  4. **Teardown** — Clean up the test fixture, if necessary. Many tests omit this phase, but some types of database test will, for example, roll back a transaction initiated by the setup phase.



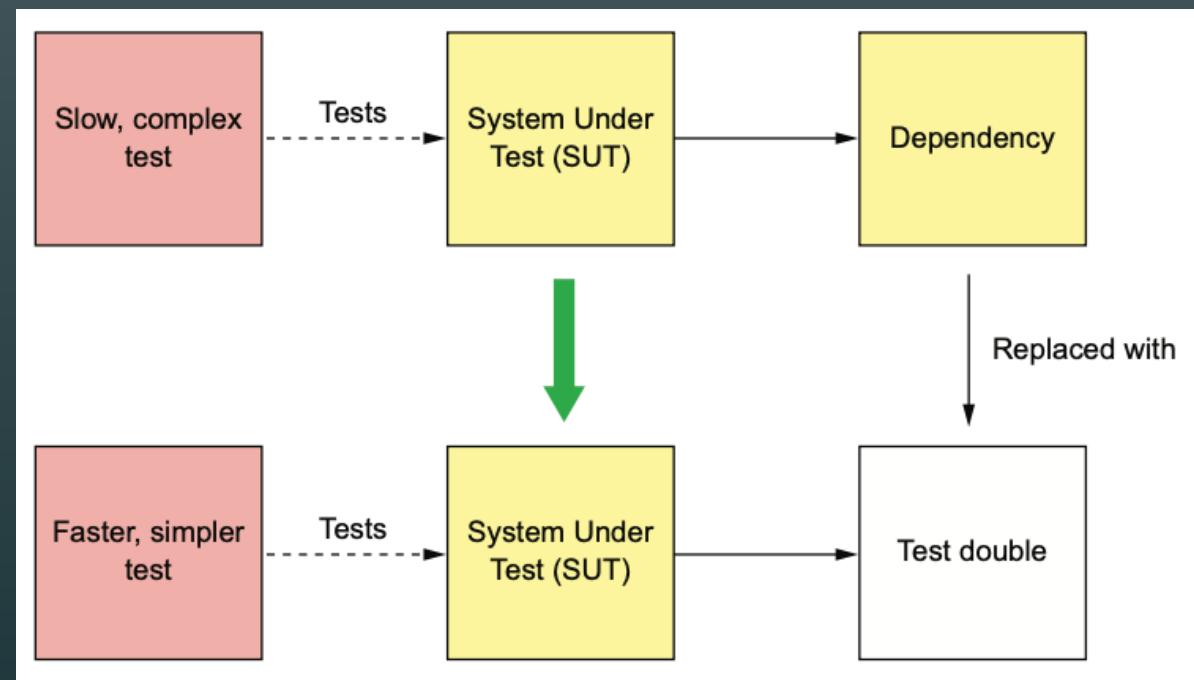
# Writing Automated Tests (2)



- A test consists of four phases: setup, execute, verify, and teardown.

# Testing Using Mocks and Stub (1)

- An SUT often has dependencies.
- The solution is to replace the SUT's dependencies with **Test doubles**.
- Replacing a dependency with a test double enables the SUT to be tested in isolation; The test is simpler and faster.



# Testing Using Mocks and Stub (2)

- There are two types of test doubles: **stubs** and **mocks**.
  - The terms stubs and mocks are often used interchangeably, although they have slightly different behavior.
- A **stub** is a test double that returns values to the SUT.
- A **mock** is a test double that a test uses to verify that the SUT correctly invokes a dependency. Also, a mock is often a stub.

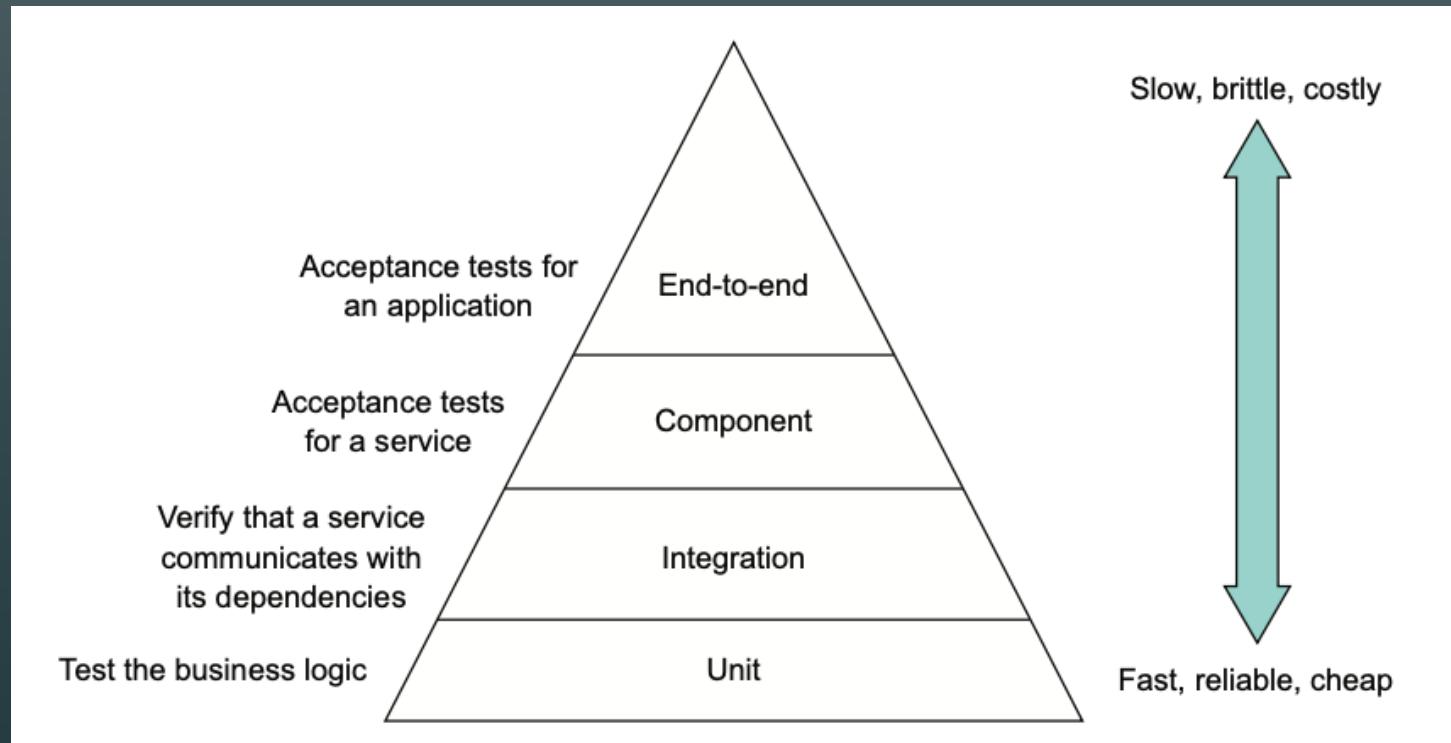


# The Different Type of Tests

- We focus on automated tests that verify the **functional aspects** of the application or service (Functional Requirements - FR).
- There are **four** different types of tests:
  1. **Unit tests** — Test a small part of a service, such as a class.
  2. **Integration tests** — Verify that a service can interact with infrastructure services such as databases and other application services.
  3. **Component tests** — Acceptance tests for an individual service.
  4. **End-to-end tests** — Acceptance tests for the entire application.



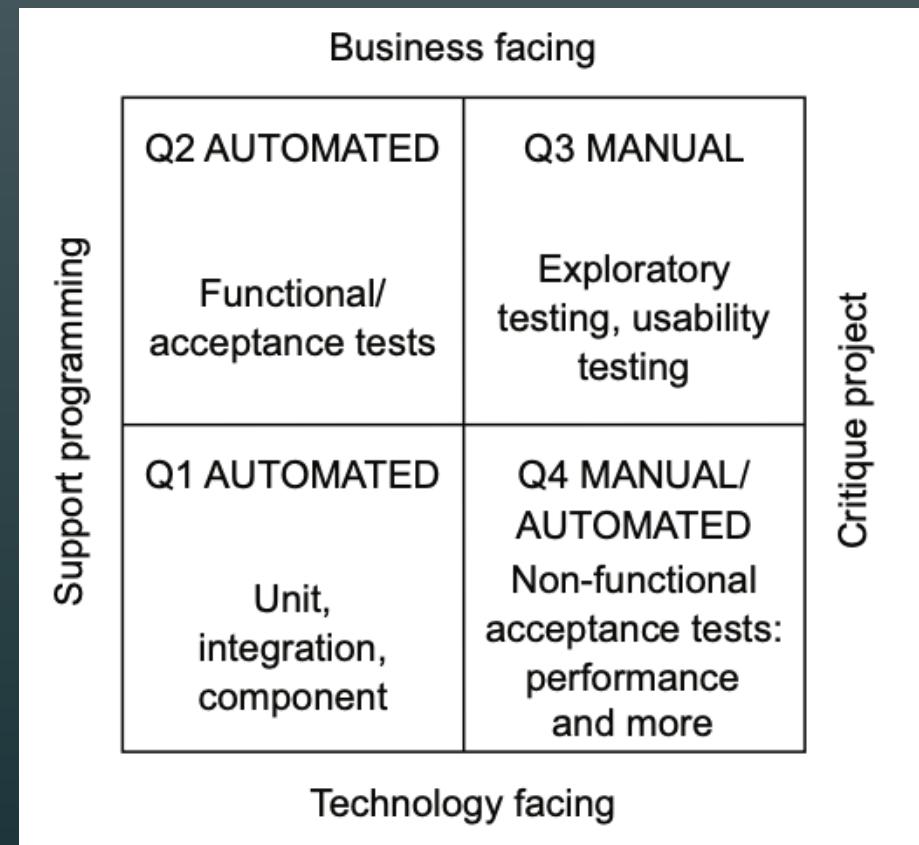
# Test Pyramid



- The **test pyramid** describes the relative proportions of each type of test.
- As you move up the pyramid, you should write fewer and fewer tests.

# Brian Marick's test quadrant

- The test quadrant categorizes tests along two dimensions.
- The first dimension is whether a test is business facing or technology facing.
- The second is whether the purpose of the test is to support programming or critique the application.



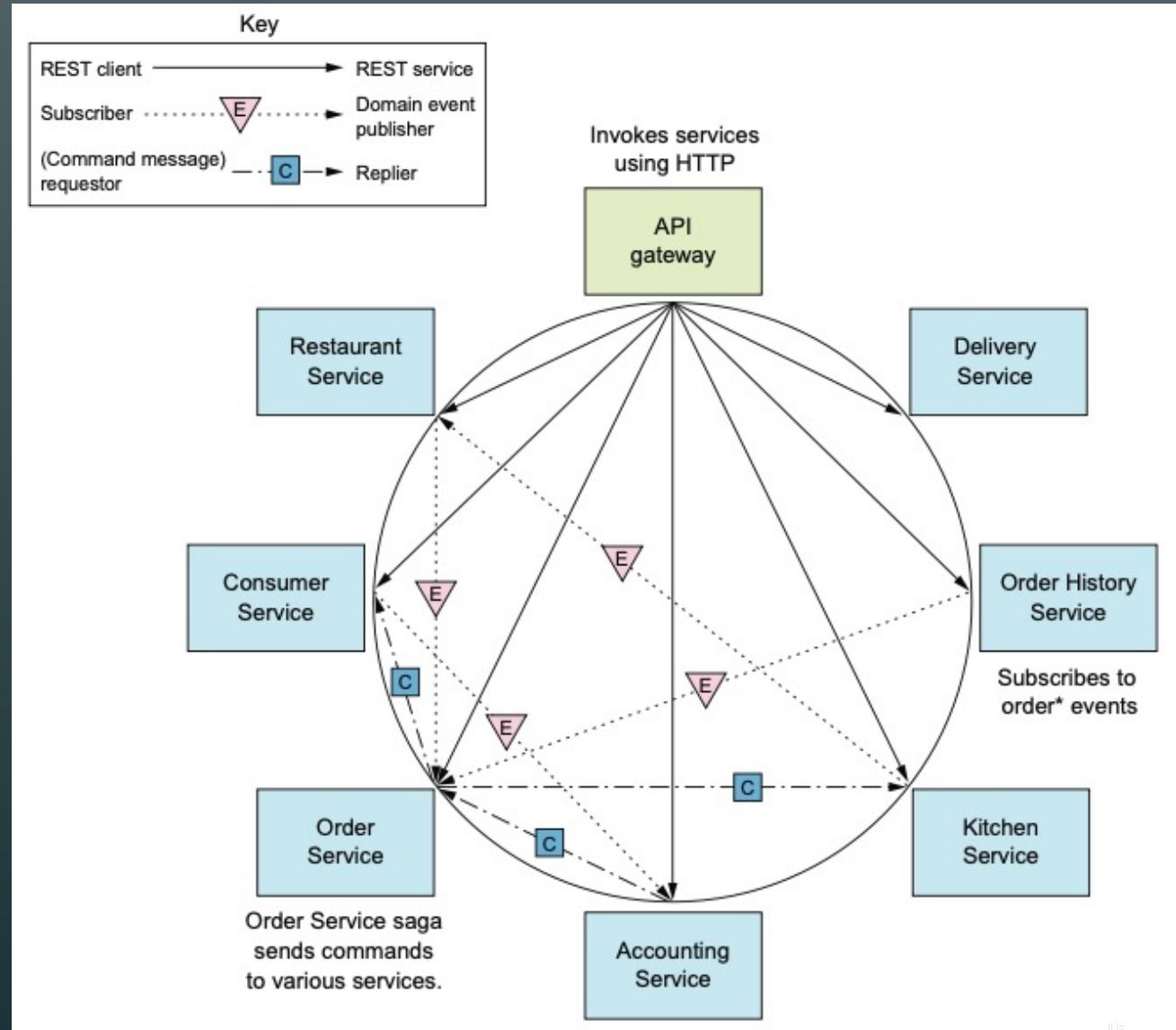
# Agenda

- Overview of testing microservice
- Testing strategies for microservice architectures
- The challenge of testing microservices
- The deployment pipeline
- Writing unit tests for a service



# The challenge of testing microservices

- The figure shows some interprocess communication in the FTGO application.
  - Each arrow points from a consumer service to a producer service.



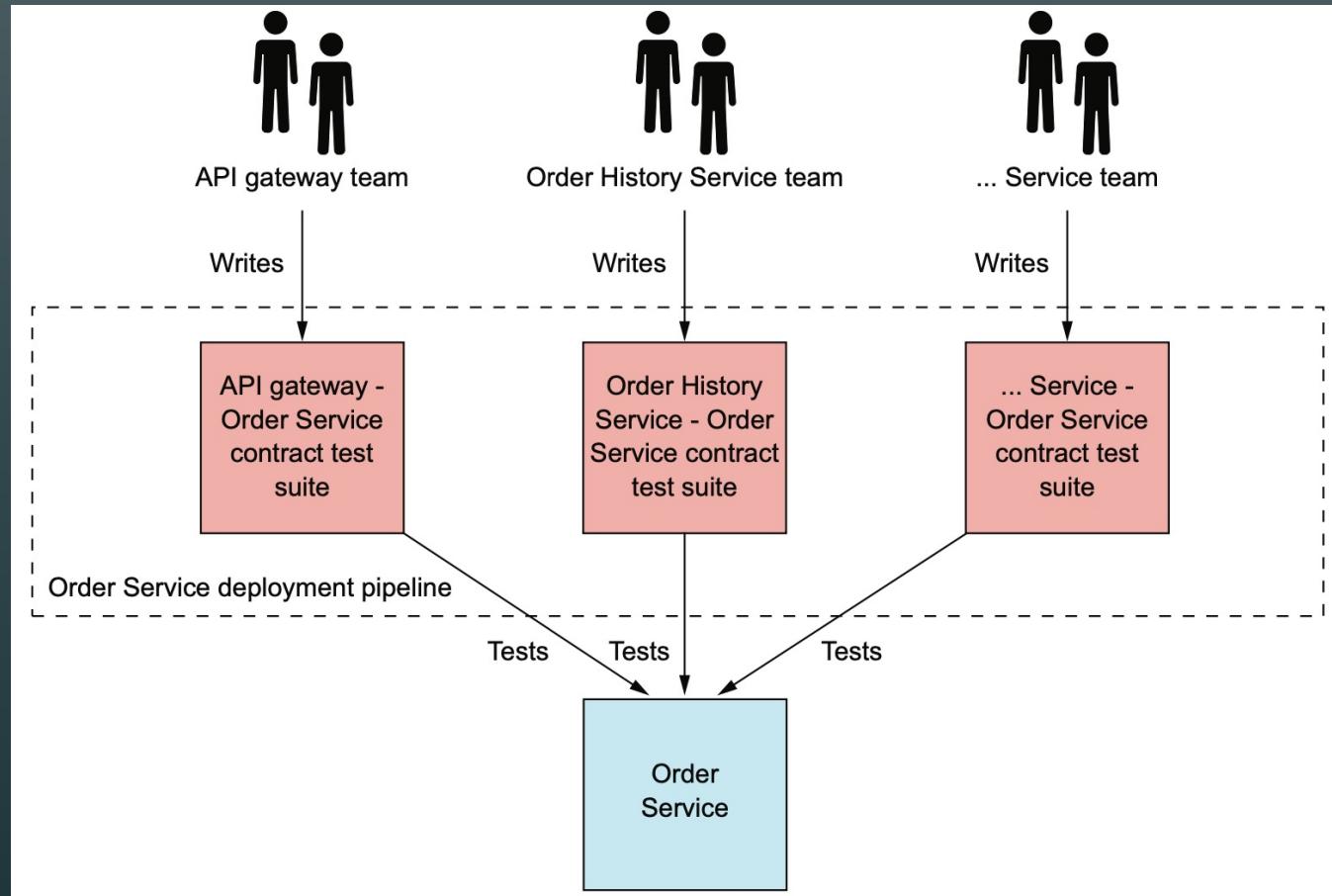
# Consumer-Driven Contract Testing (1)

- A **consumer contract test** focuses on verifying that the “shape” of a provider’s API meets the **consumer’s expectations**.
- For a REST endpoint, a contract test verifies that the provider implements an endpoint (**for the consumer**) that
  - Has the expected HTTP method and path
  - Accepts the expected headers, if any
  - Accepts a request body, if any
  - Returns a response with the expected status code, headers, and body



# Order Service's API testing

- Each team that develops a service that consumes Order Service's API contributes a contract test suite.
- The test suite verifies that the API matches the consumer's expectations.
- This test suite, along with those contributed by other teams, is run by Order Service's deployment pipeline.



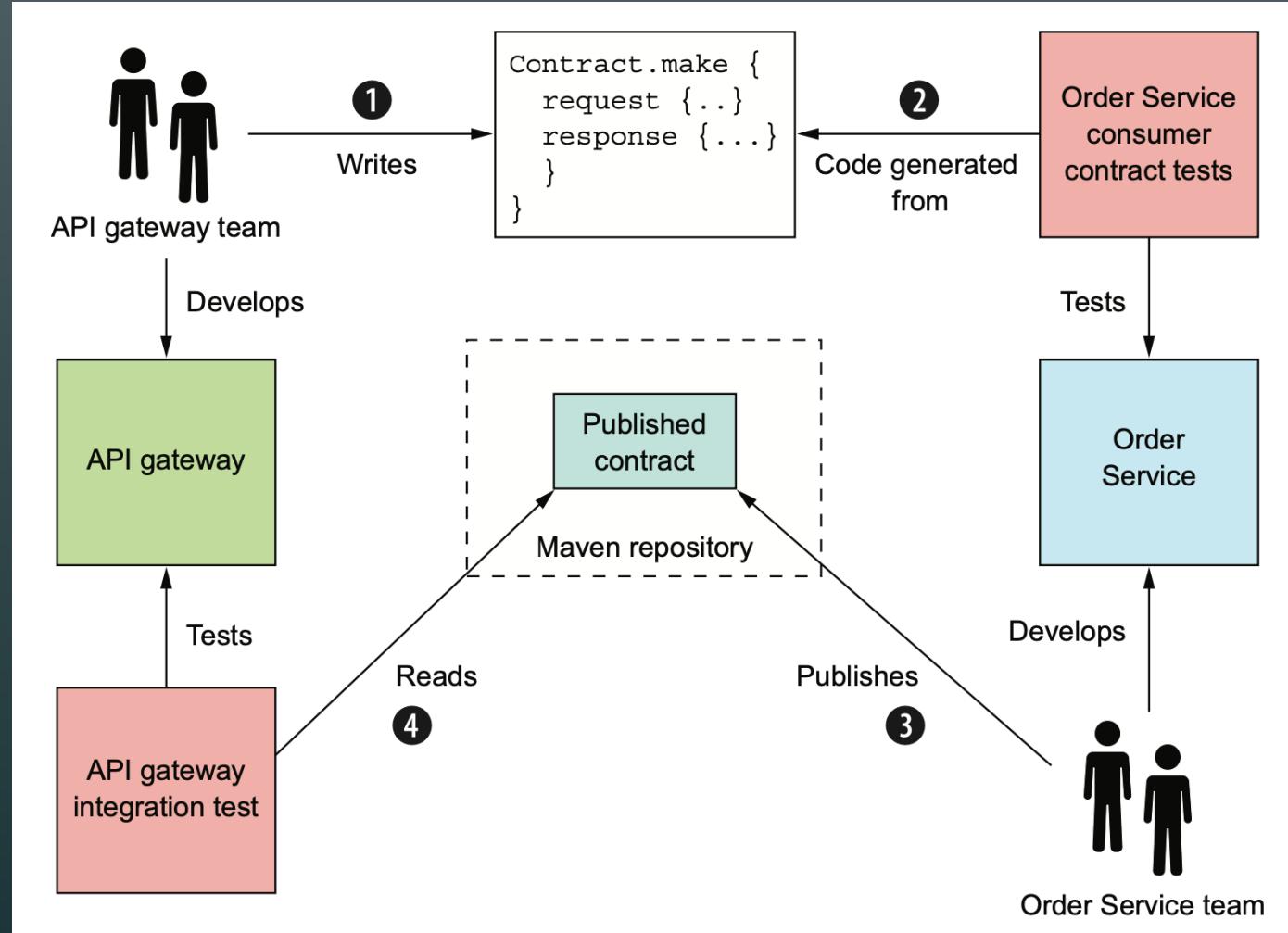
# Consumer-Driven Contract Testing (2)

- The interaction between a consumer and a provider is defined by a set of examples, known as **contracts**.
  - Each **contract** consists of example messages that are exchanged during one interaction.
- Even though the focus of consumer-driven contract testing is to test a **provider**, contracts are also used to verify that the **consumer** conforms to the contract.
  - Testing both sides of interaction ensures that the consumer and provider agree on the API



# Ex. Testing Service Using Spring Cloud Contract

1. The API Gateway team writes the contracts.
2. The Order Service team uses those contracts to test Order Service and publishes them to a repository.
3. The API Gateway team uses the published contracts to test API Gateway.



# Agenda

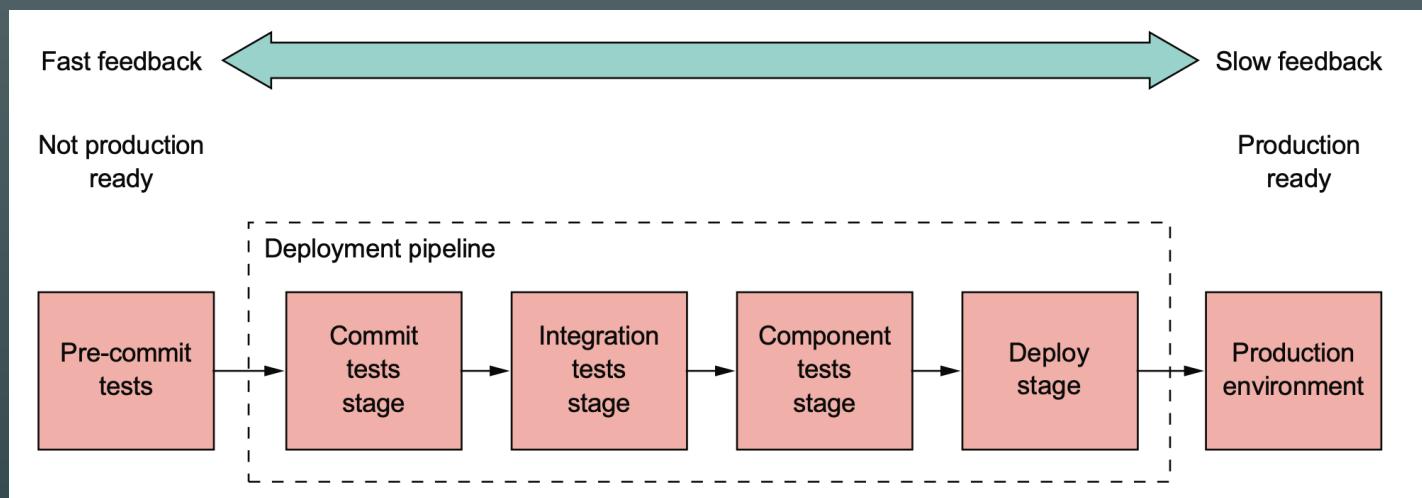
- Overview of testing microservice
- Testing strategies for microservice architectures
- The challenge of testing microservices
- The deployment pipeline
- Writing unit tests for a service



# The deployment pipeline

- Every service has a **deployment pipeline**.
- **Continuous Delivery (CD)** describes a deployment pipeline as the automated process of getting code from the developer's desktop into production.
  - Ideally, it's fully automated, but it might contain manual steps.
- A deployment pipeline is often implemented using a **Continuous Integration (CI)** server, such as **Jenkins**.





1. **Pre-commit tests stage** — Runs the unit tests. This is executed by the developer before committing their changes.
2. **Commit tests stage** — Compiles the service, runs the unit tests, and performs static code analysis.
3. **Integration tests stage** — Runs the integration tests.
4. **Component tests stage** — Runs the component tests for the service.
5. **Deploy stage** — Deploys the service into production.

# Agenda

- Overview of testing microservice
- Testing strategies for microservice architectures
- The challenge of testing microservices
- The deployment pipeline
- Writing unit tests for a service



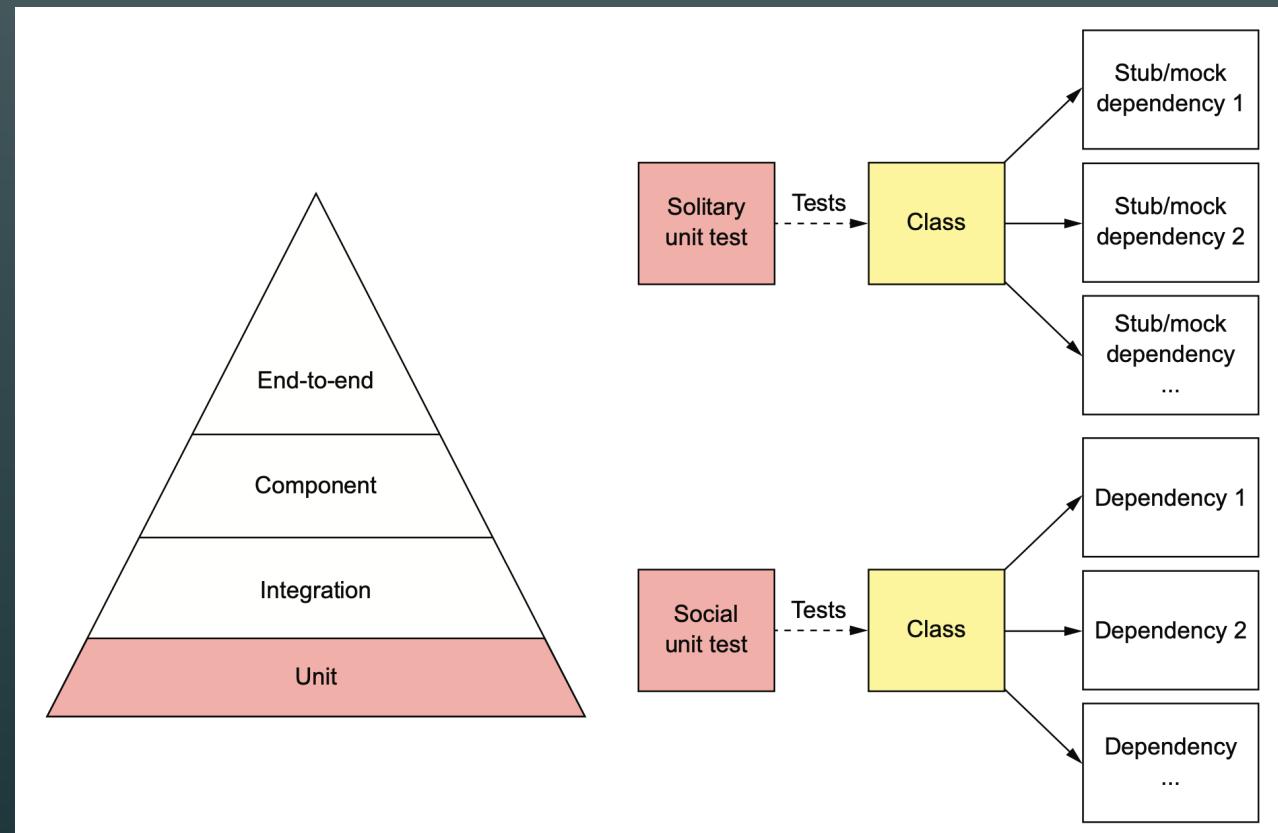
# Writing Unit Tests for a Service (1)

- **Unit tests** are the lowest level of the test pyramid.
- They're technology-facing tests that support development.
- A unit test verifies that a unit, which is a very small part of a service, works correctly.
- A unit is typically a class, so the goal of unit testing is to verify that it behaves as expected.



# Writing Unit Tests for a Service (2)

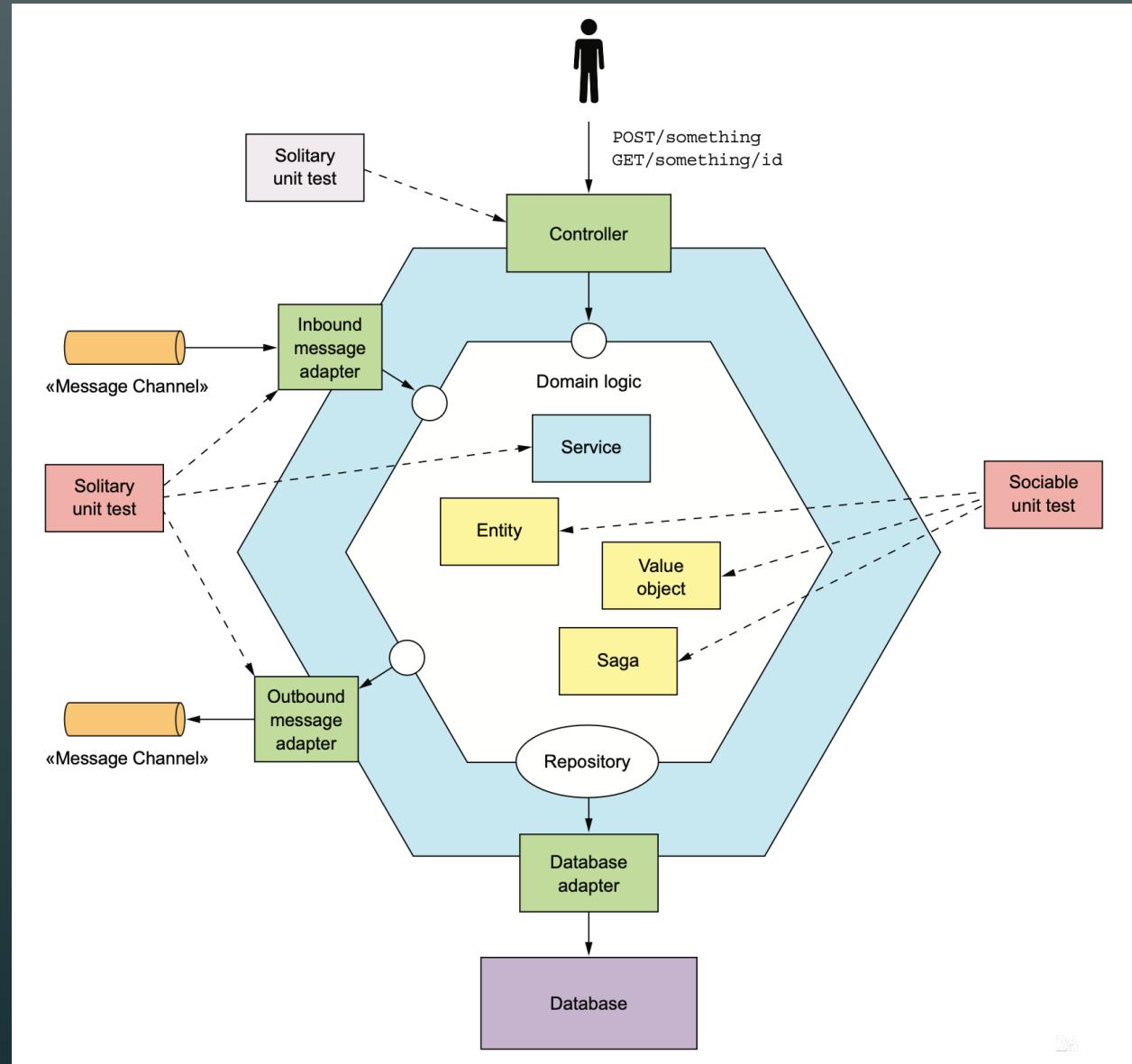
- There are two types of unit tests\*
  - **Solitary** unit test —  
Tests a class in isolation using mock objects for the class's dependencies
  - **Sociable** unit test —  
Tests a class and its dependencies



\*<https://martinfowler.com/bliki/UnitTest.html>

# Writing Unit Tests for a Service (3)

- The responsibilities of the class and its role determine which type of test to use.
  - Controller and service classes are often tested using **solitary** unit tests.
  - Domain objects, such as entities and value objects, are typically tested using **sociable** unit tests.



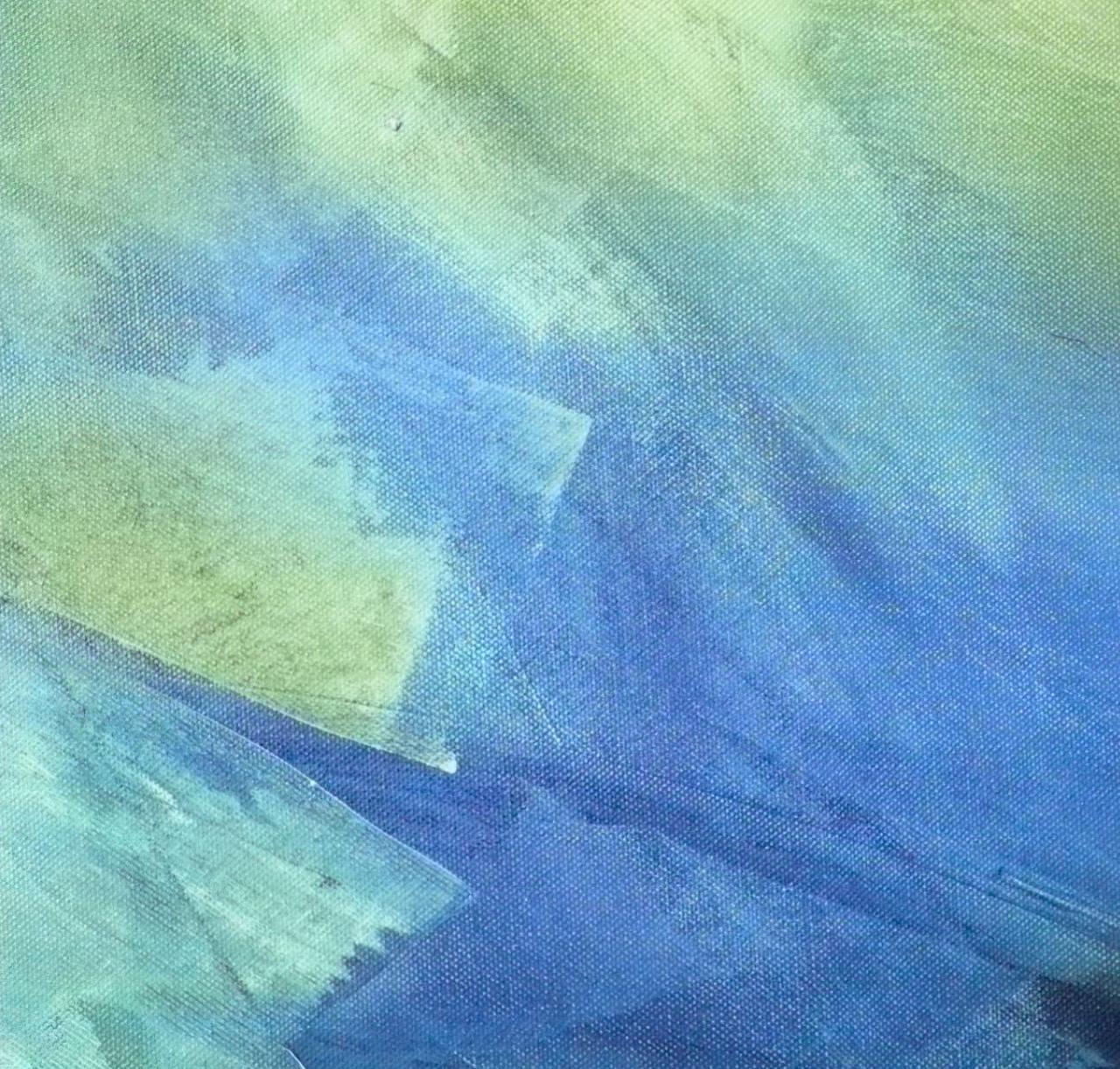
# Writing Unit Tests for a Service (4)

- Entities, such as Order, are objects with persistent identity → **sociable** unit tests.
- Value objects, such as Money, are objects that are collections of values → **sociable** unit tests.
- Sagas, such as CreateOrderSaga, maintain data consistency across services → **sociable** unit tests.
- Domain services, such as OrderService, are classes that implement business logic that doesn't belong in entities or value objects → **solitary** unit tests.
- Controllers, such as OrderController, which handle HTTP requests → **solitary** unit tests.
- Inbound and outbound messaging gateways → **solitary** unit tests.

# Summary

- Automated testing is the key foundation of rapid, safe delivery of software
- The purpose of a test is to verify the behavior of the system under test (SUT).
- A good way to simplify and speed up a test is to use test doubles.
- Use the test pyramid to determine where to focus your testing efforts for your services.





# Testing Microservices Part 2

Wiwat Vatanawood

Duangdao Wichadakul

Nuengwong Tuaycharoen

Pittipol Kantavat

# The Chapter covers

- Techniques for testing services in isolation
- Using consumer-driven contract testing to write tests that quickly yet reliably verify interservice communication
- When and how to do end-to-end testing of applications



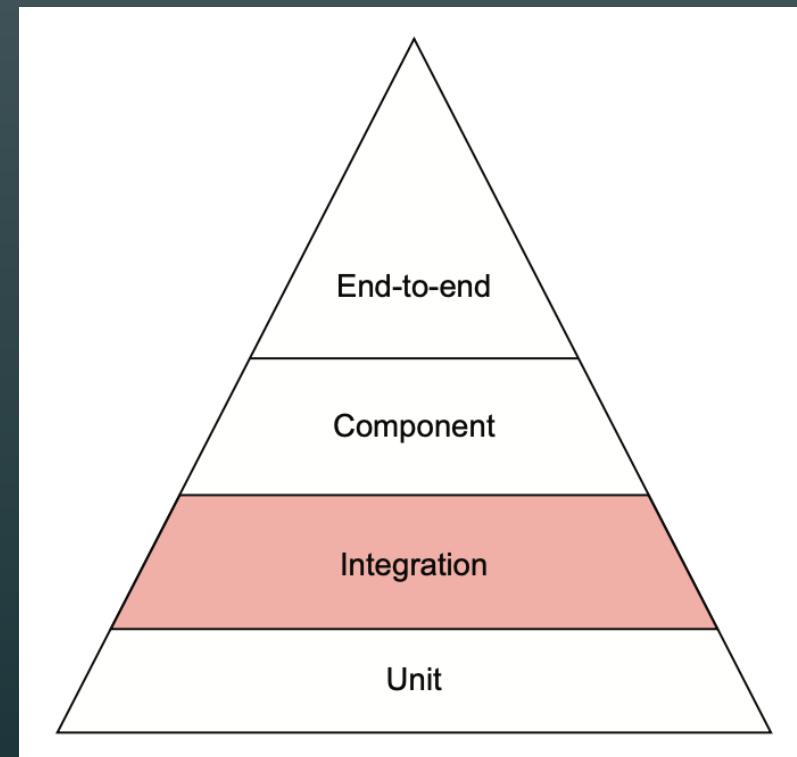
# Agenda

- Writing integration tests
- Developing component tests
- Writing end-to-end tests



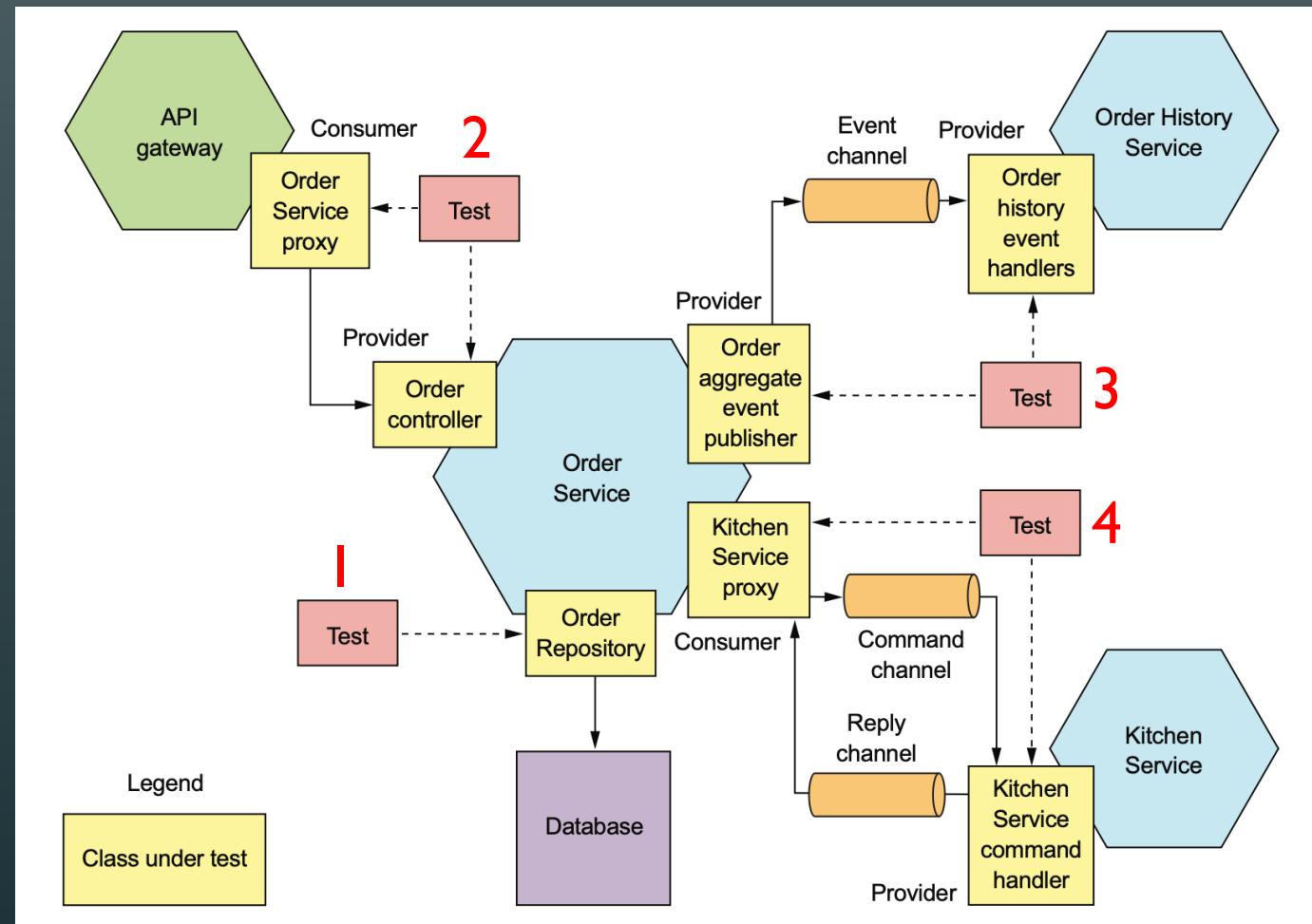
# Writing Integration Tests

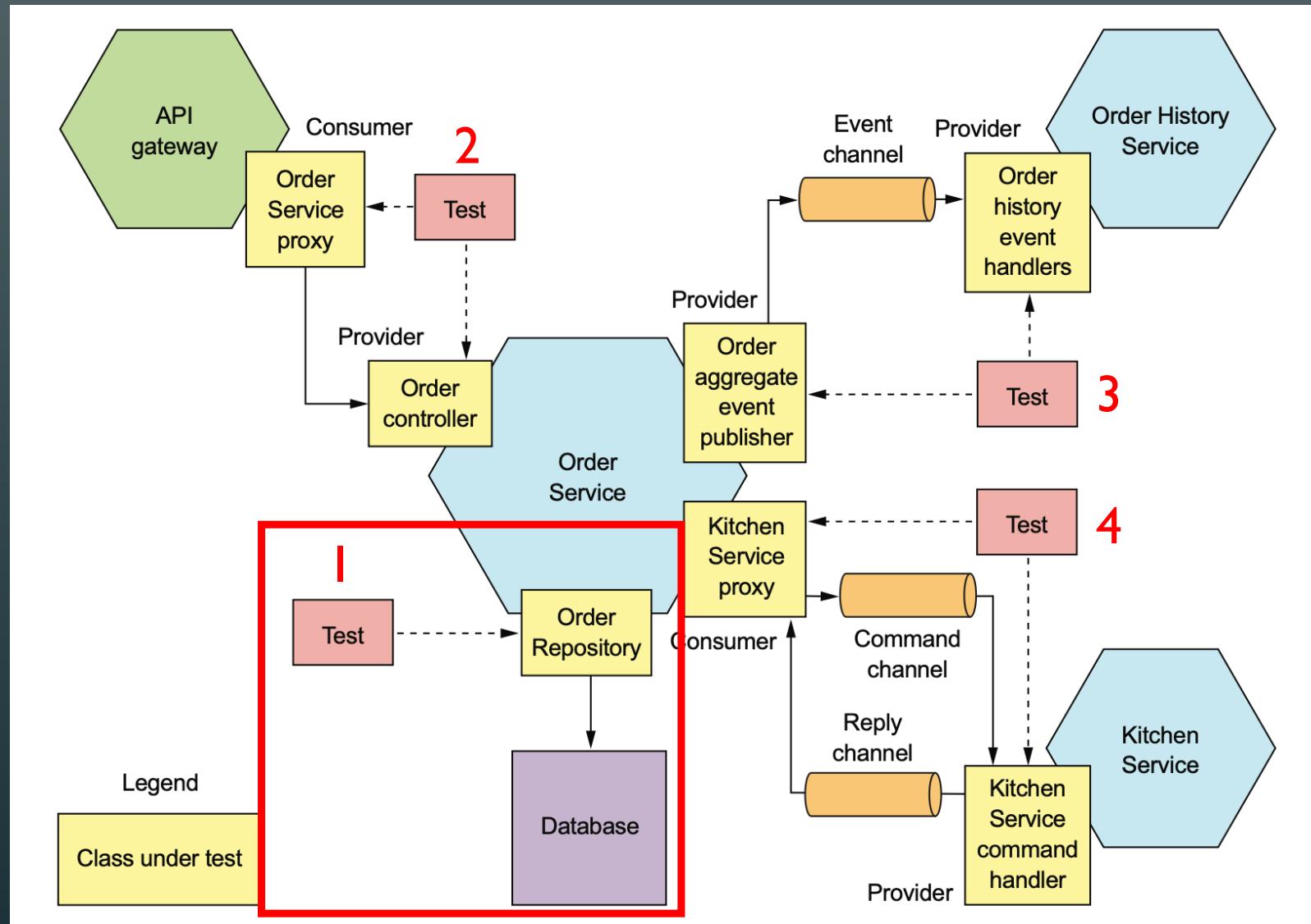
- Integration tests are the layer above unit tests.
- They verify that a service can communicate with its dependencies, which includes infrastructure services, such as the database, and application services.



# Writing Integration Tests (cont.)

- Integration tests must verify that a service can communicate with its clients and dependencies.
- But rather than testing whole services, the strategy is to test the **individual adapter** classes that implement the communication

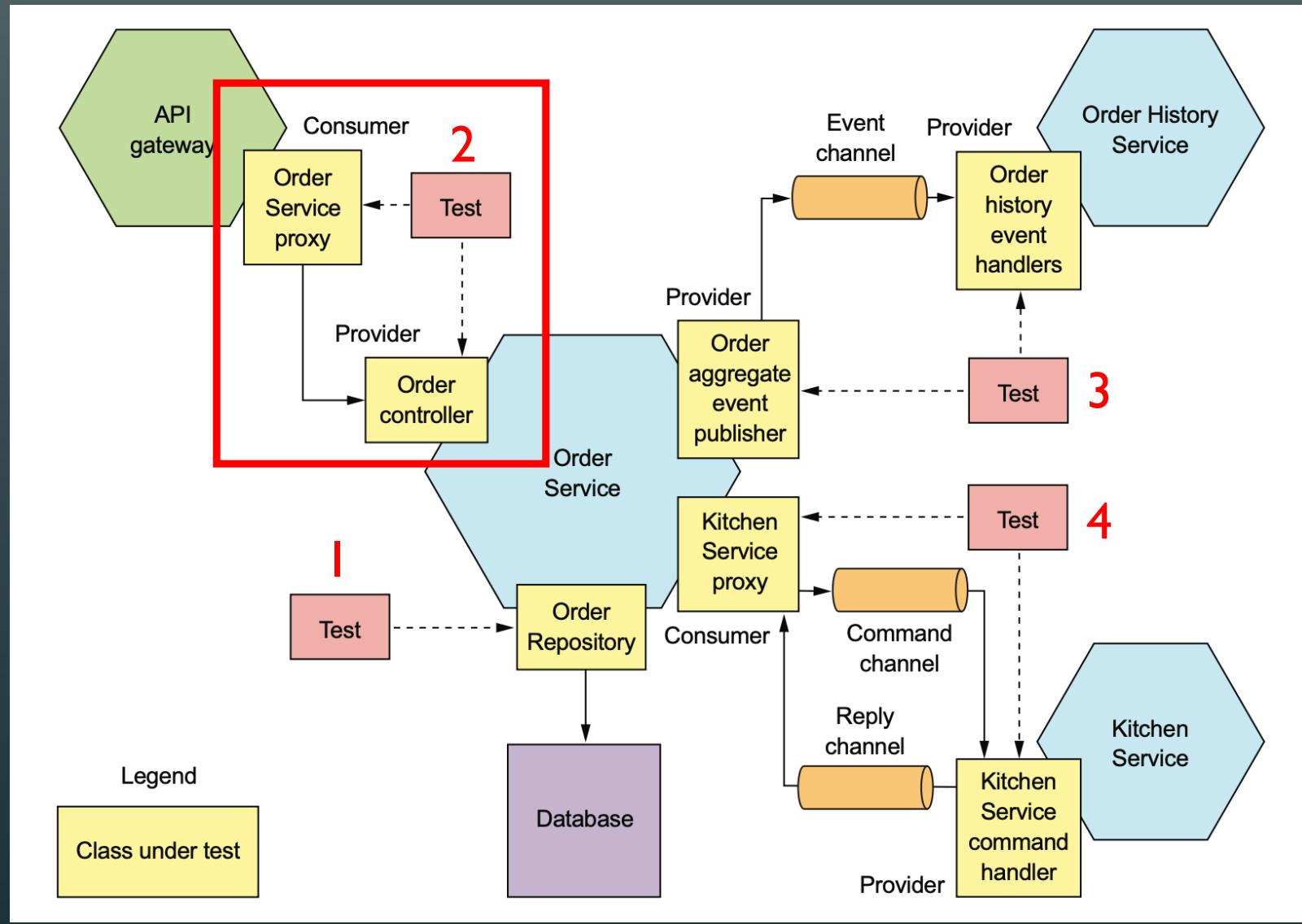




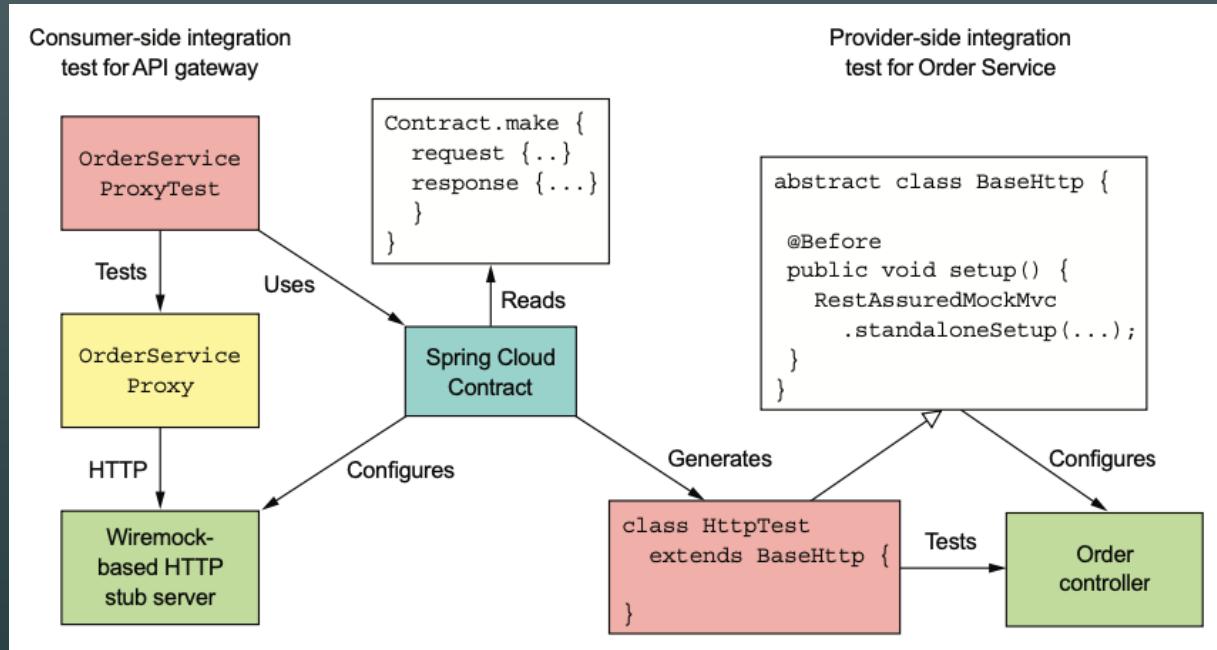
# I. Persistence Integration tests

- Verifying that a service's database access logic works as expected.
- Each phase of a persistence integration test behaves as follows:
  - **Setup** — Set up the database by creating the database schema and initializing it to a known state. It might also begin a database transaction.
  - **Execute** — Perform a database operation.
  - **Verify** — Make assertions about the state of the database and objects retrieved from the database.
  - **Teardown** — An optional phase that might undo the changes made to the database.
    - for example, rolling back the transaction that was started by the setup phase.

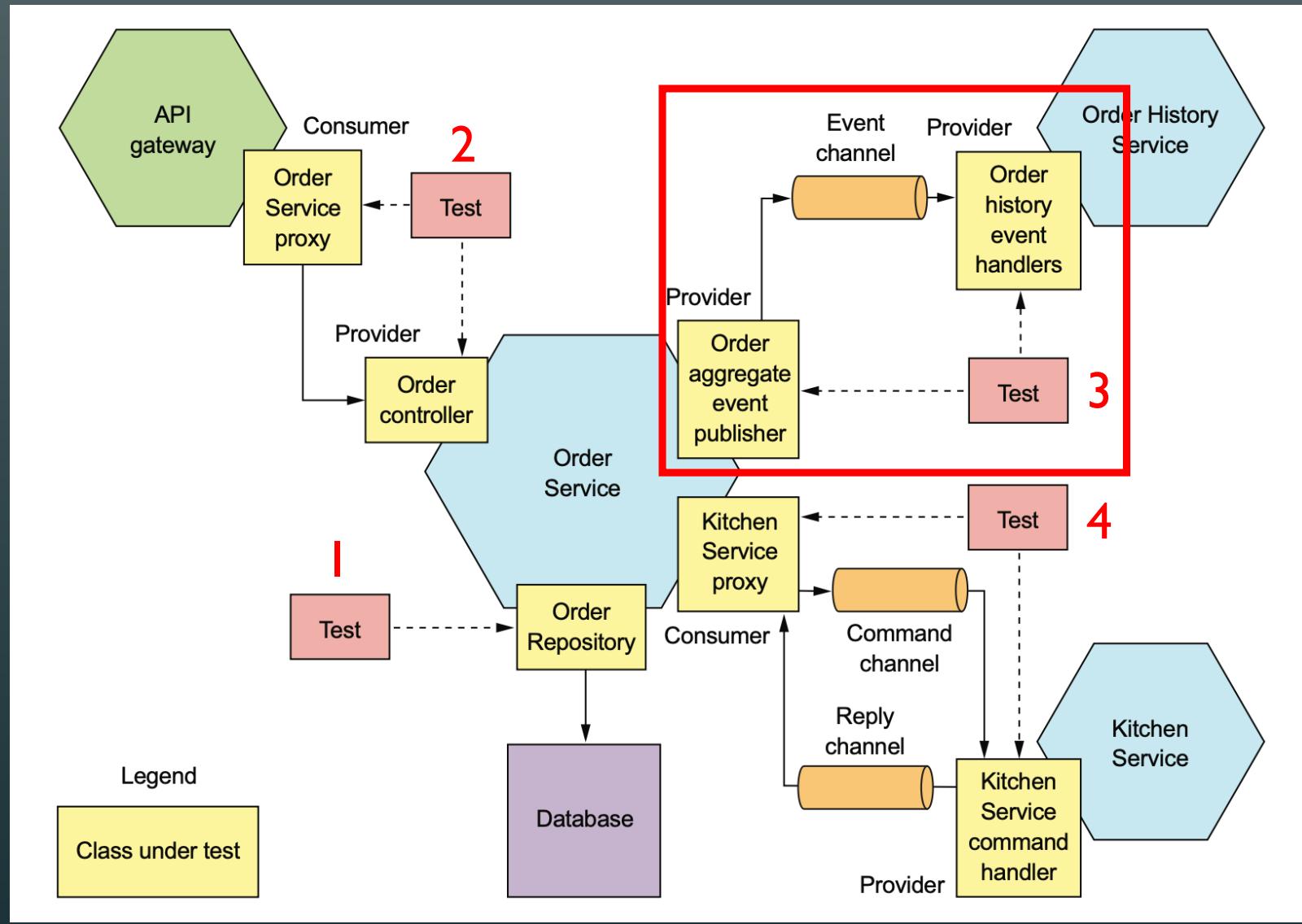




## 2. Integration testing REST-based request/response style interactions

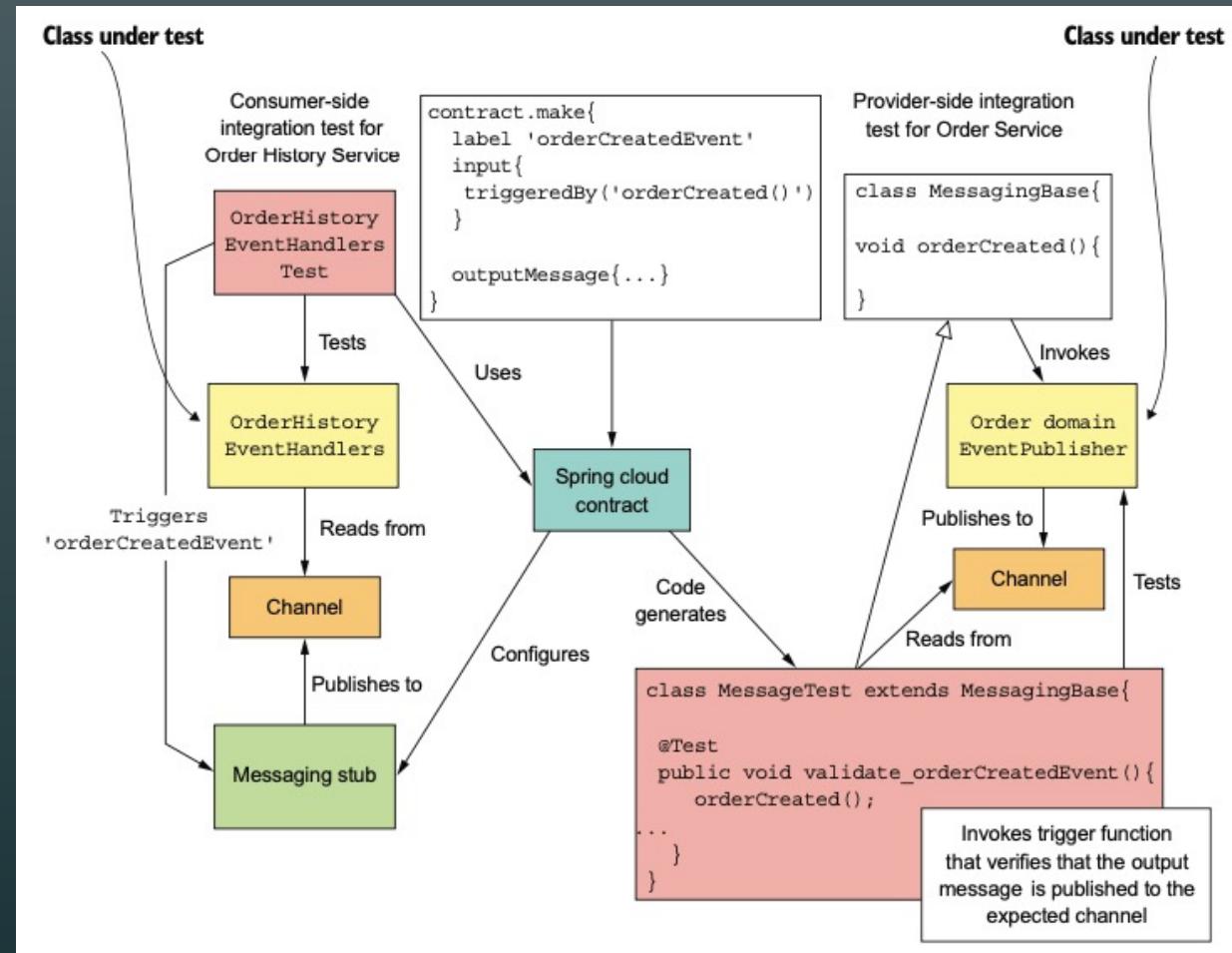


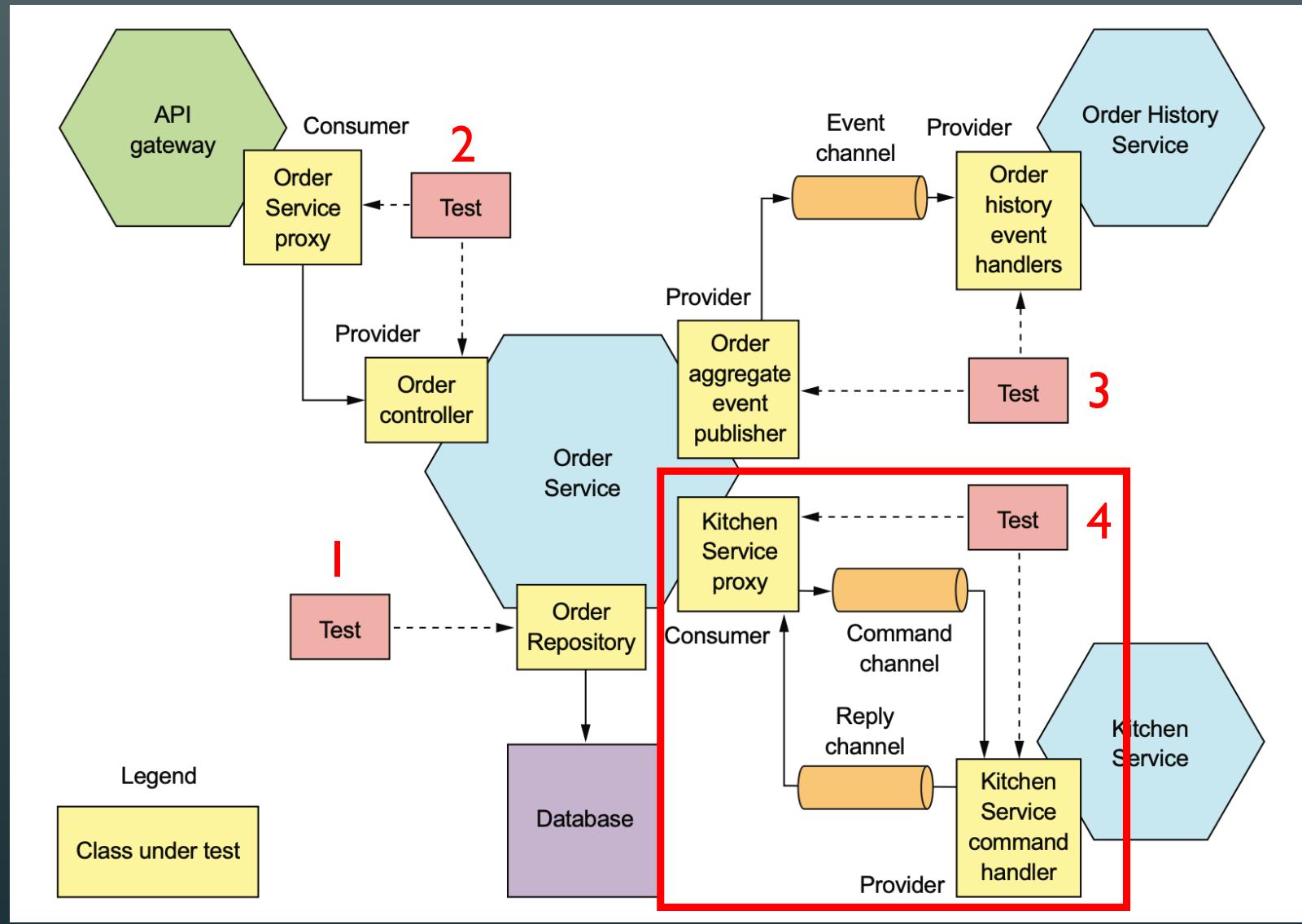
- The **contracts** are used to verify that the adapter classes on both sides.
- The consumer-side tests verify that **OrderServiceProxy** invokes Order Service correctly.
- The provider-side tests verify that **OrderController** implements the REST API endpoints correctly.



### 3. Integration testing publish/subscribe-style interactions

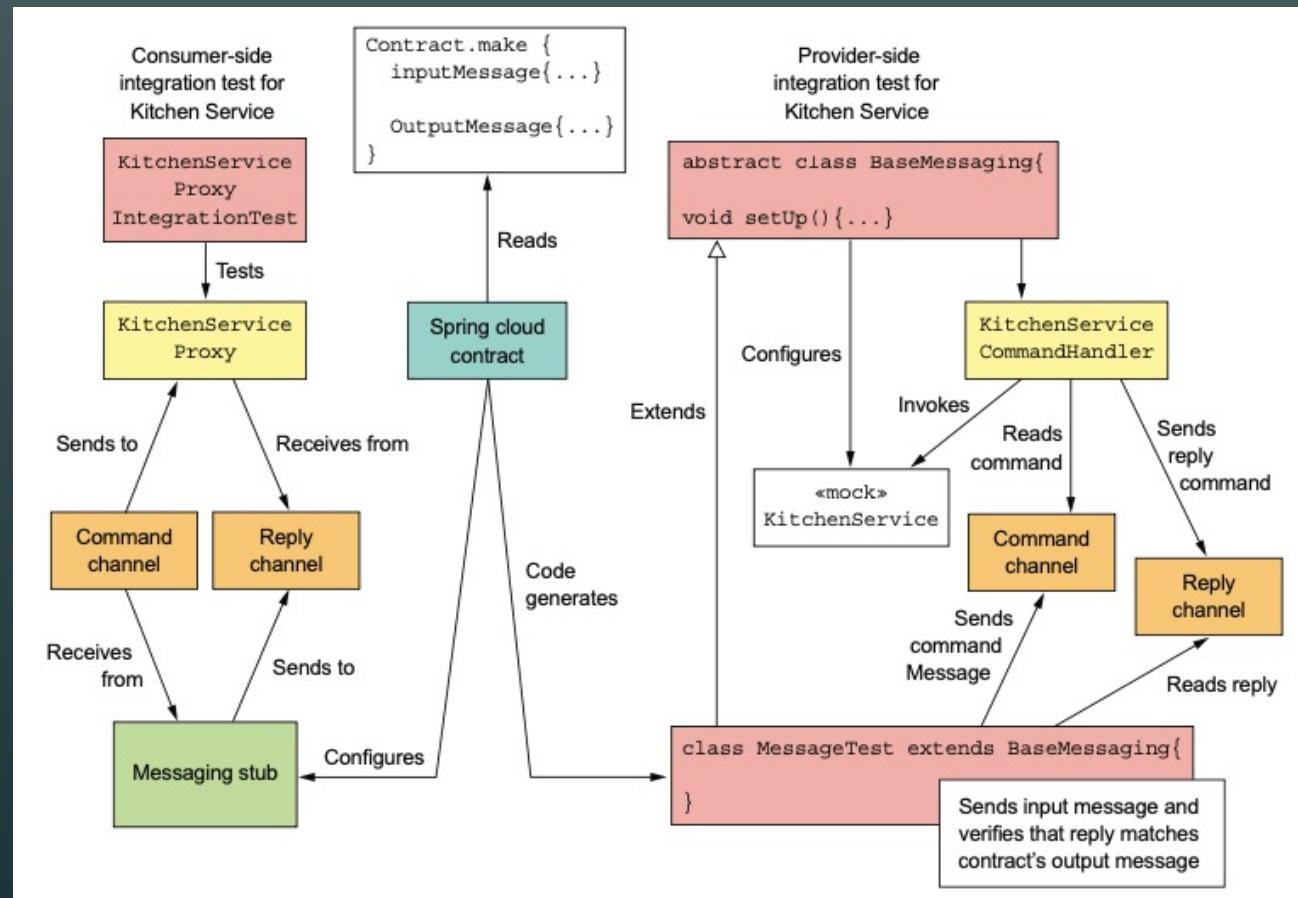
- The **contracts** are used to test both sides of the publish/subscribe interaction.
- The provider-side tests verify that **OrderDomainEventPublisher** publishes events that conform to the contract.
- The consumer-side tests verify that **OrderHistoryEventHandlers** consume the example events from the contract.





## 4. Integration tests for asynchronous request/response interactions

- The **contracts** are used to test the adaptor classes.
- The provider-side tests verify that **KitchenServiceCommandHandler** handles commands and sends back replies.
- The consumer-side tests verify **KitchenServiceProxy** sends commands that conform to the contract, and that it handles the example replies from the contract.



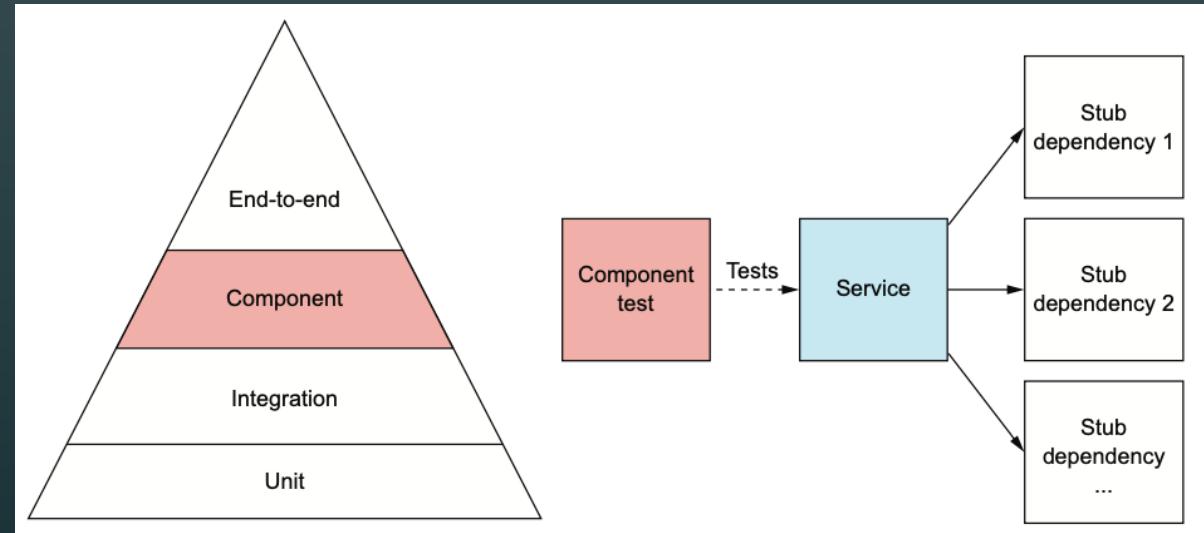
# Agenda

- Writing integration tests
- Developing component tests
- Writing end-to-end tests



# Developing Component Tests

- We now want to verify that Order Service works as expected.
- In other words, we want to write the service's **acceptance tests**, which treat it as a **black box** and verify its behavior through its API.
- Component testing verifies the **behavior of a service** in isolation.
- It replaces a service's dependencies with **stubs** that simulate their behavior.



# Defining Acceptance Tests

- Acceptance tests are **business-facing tests** for a software component.
- They describe the desired externally visible behavior from the perspective of the **component's clients** rather than in terms of the **internal implementation**.
- These tests are derived from user stories or use cases.
  - For example, one of the key stories for Order Service is the Place Order story:



As a consumer of the Order Service  
I should be able to place an order

# Defining Acceptance Tests (cont.)

- We can expand this story into scenarios such as the following:

```
Given a valid consumer  
Given using a valid credit card  
Given the restaurant is accepting orders  
When I place an order for Chicken Vindaloo at Ajanta  
Then the order should be APPROVED  
And an OrderAuthorized event should be published
```

- Each scenario defines an acceptance test.
  - The **givens** correspond to the test's setup phase.
  - The **when** maps to the execute phase.
  - The **then** and the **and** to the verification phase.



# Writing acceptance tests using Gherkin

- **Gherkin** is a domain specific language (DSL) for writing executable specifications.
- When using **Gherkin**, you define your acceptance tests using English-like scenarios.
- You then execute the specifications using **Cucumber**, a test automation framework for **Gherkin**.
- **Gherkin** and **Cucumber** eliminate the need to translate scenarios into runnable code manually.



# Example of Using Gherkin specification

## **Listing 10.11. The Gherkin definition of the Place Order feature and some of its scenarios**

Feature: Place Order

As a consumer of the Order Service  
I should be able to place an order

Scenario: Order authorized

Given a valid consumer  
Given using a valid credit card  
Given the restaurant is accepting orders  
When I place an order for Chicken Vindaloo at Ajanta  
Then the order should be APPROVED  
And an OrderAuthorized event should be published

Scenario: Order rejected due to expired credit card

Given a valid consumer  
Given using an expired credit card  
Given the restaurant is accepting orders  
When I place an order for Chicken Vindaloo at Ajanta  
Then the order should be REJECTED  
And an OrderRejected event should be published

...

# Executing Gherkin Specifications Using Cucumber

- “@Given”
  - The setup phase
- “@When”
  - The execute phase
- “@Then” and “@And”
  - The verification phase

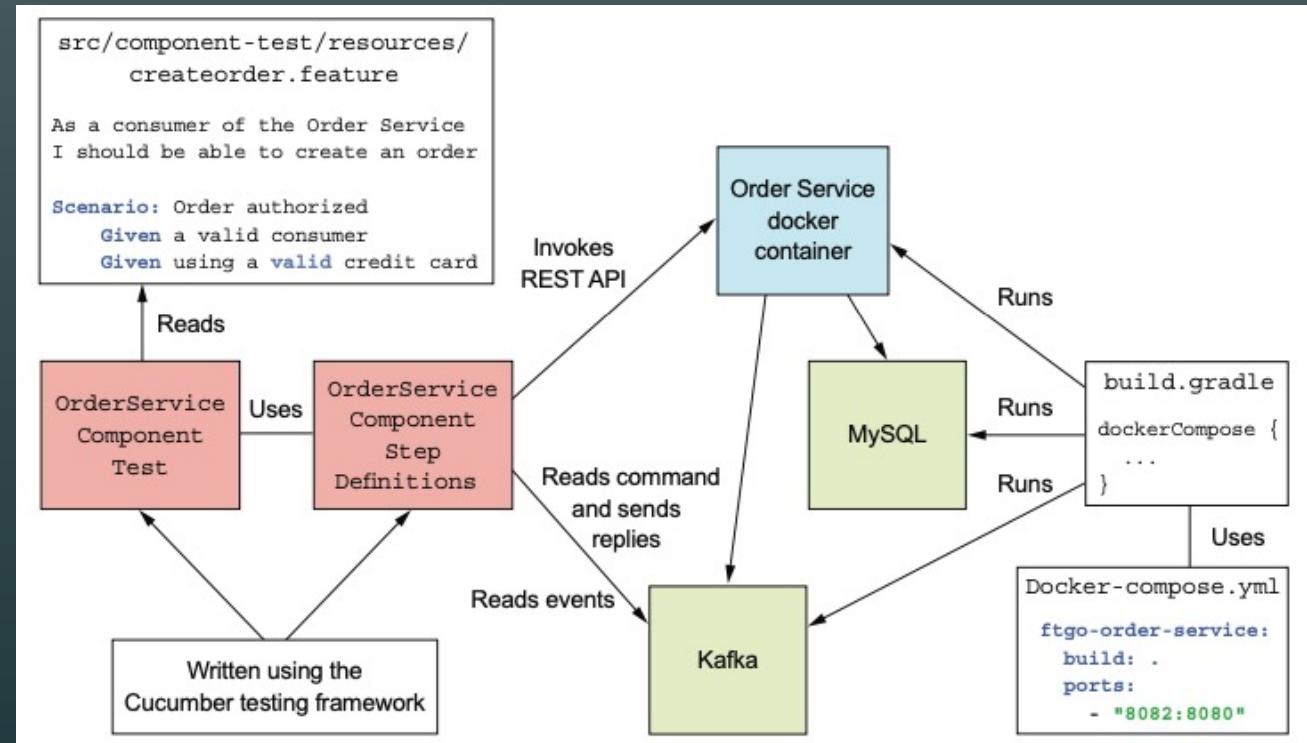
**Listing 10.12 The Java step definitions class makes the Gherkin scenarios executable.**

```
public class StepDefinitions ... {  
    ...  
  
    @Given("A valid consumer")  
    public void useConsumer() { ... }  
  
    @Given("using a(.*?) (.*) credit card")  
    public void useCreditCard(String ignore, String creditCard) { ... }  
  
    @When("I place an order for Chicken Vindaloo at Ajanta")  
    public void placeOrder() { ... }  
  
    @Then("the order should be (.*)")  
    public void theOrderShouldBe(String desiredOrderState) { ... }  
  
    @And("an (.*) event should be published")  
    public void verifyEventPublished(String expectedEventClass) { ... }  
}
```



# Writing component tests for the FTGO Order Service

- The component tests for Order Service use the **Cucumber** testing framework to execute tests scenarios written using **Gherkin** acceptance testing DSL.
- The tests use **Docker** to run Order Service along with its infrastructure services, such as **Apache Kafka** and **MySQL**.



# Agenda

- Writing integration tests
- Developing component tests
- Writing end-to-end tests



# Writing End-to-end Tests

- End-to-end tests are business-facing tests.
- You can write the end-to-end tests using **Gherkin** and execute them using **Cucumber**.
- The main difference is that rather than a single “**Then**”, this test has multiple actions.



# Example of Writing End-to-end Tests

## **Listing 10.17 A Gherkin-based specification of a user journey**

Feature: Place Revise and Cancel

As a consumer of the Order Service  
I should be able to place, revise, and cancel an order

Scenario: Order created, revised, and cancelled

Given a valid consumer

Given using a valid credit card

Given the restaurant is accepting orders

When I place an order for Chicken Vindaloo at Ajanta

Then the order should be APPROVED

Then the order total should be 16.33

And when I revise the order by adding 2 vegetable samosas

Then the order total should be 20.97

And when I cancel the order

Then the order should be CANCELLED

**Create Order.**

**Revise Order.**

**Cancel Order.**

# Running End-to-end Tests

- End-to-end tests must run the entire application, including any required infrastructure services.
- The implementation of the end-to-end test is quite similar to the implementation of the component tests.
- These tests are written using Gherkin and executed using Cucumber.



# Summary (1)

- Use **contracts** to drive the testing of interactions between services.
- Write tests that verify that the **adapters** of both services conform to the contracts.
- Write **component tests** to verify the behavior of a service via its API.
- You should simplify and speed up component tests by testing a service in isolation, using **stubs** for its dependencies.



# Summary (2)

- Write **user journey** tests to minimize the number of end-to-end tests, which are slow, brittle, and time consuming.
- A **user journey test** simulates a user's journey through the application and verifies high-level behavior of a relatively large slice of the application's functionality.

