Writing Tests for Microservices

Mark Heath
CLOUD ARCHITECT

@mark_heath www.markheath.net

Overview

Test pyramid

Unit tests

- Test-Driven Development (TDD)
- Code coverage

Service-level (integration) tests

End-to-end tests

Test automation

Your microservices testing strategy should utilize multiple types of tests

The Test Pyramid

Brittle

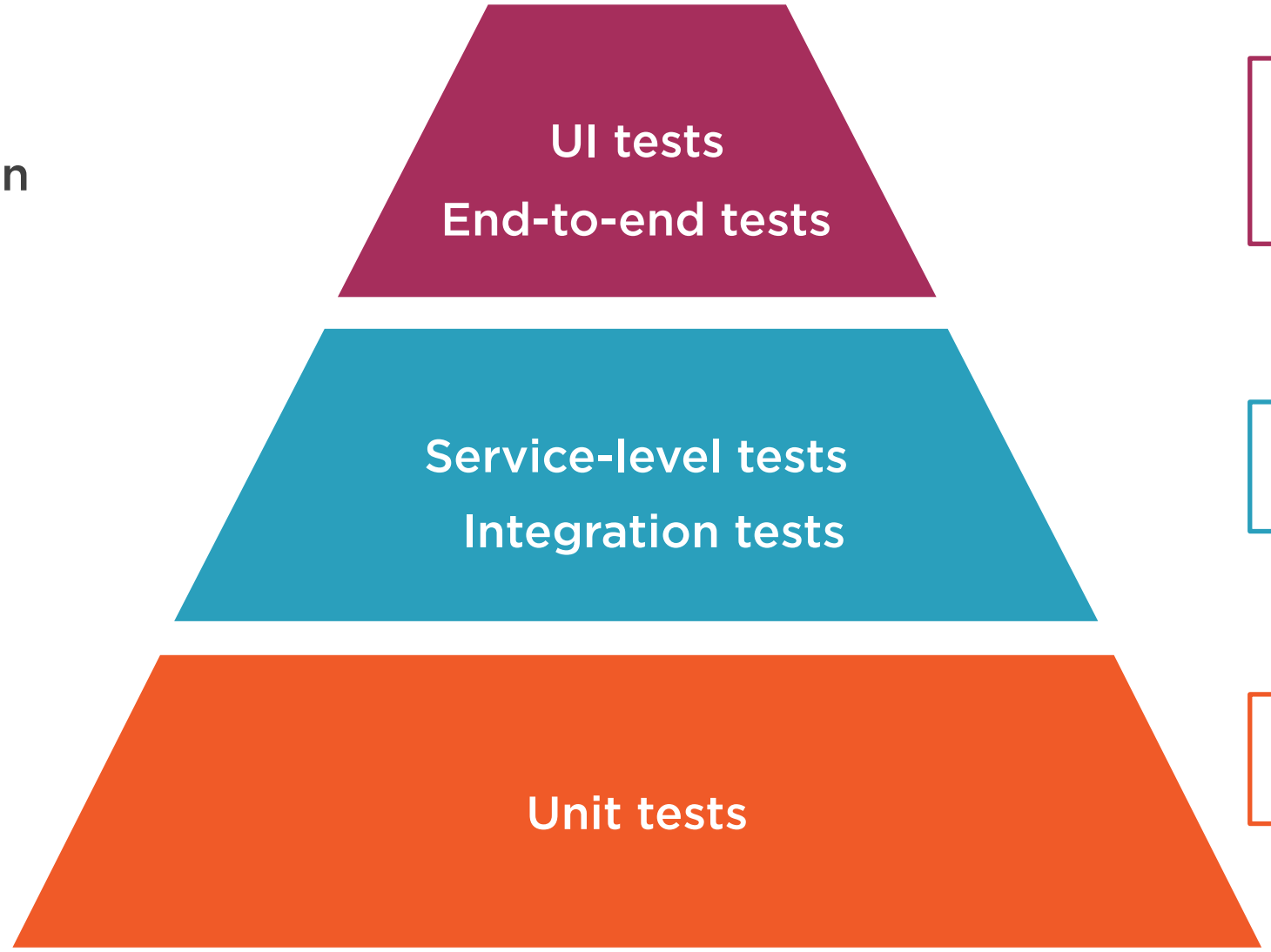
Slower

More integration

More isolation

Faster

Repeatable



Everything connected together

Talk to a real database

Dependencies are mocked

Unit Tests

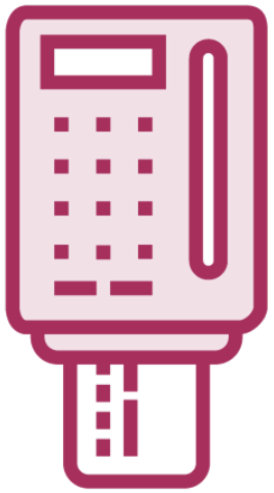
Fast

Isolated

In-process

Business logic

Example: Sales Tax



Calculate correct sales tax rate for an order

Unit tests can verify correct behaviour

Potentially complex rules

- Is customer a business or private individual?
- What country is the customer from?
- What type of item is being purchased?

Identifying Test Cases

Agree test cases with business stakeholders

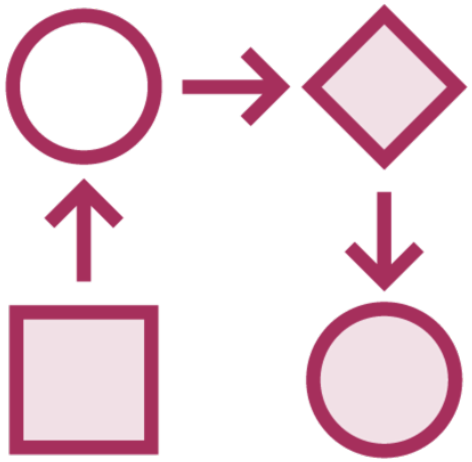
Customer Type	Location	Item Type	Sales Tax
Business	USA	Clothing	5%
Business	USA	Software	0%
Business	France	Clothing	0%
Business	France	Software	15%
Individual	N/A	Clothing	20%
Individual	N/A	Software	10%

Create a unit test for each test case

Protection against regressions

Add additional test cases if we discover bugs

Unit Testing Business Logic



Test the business logic in isolation

Don't test IO

- e.g. Network, database and disk access

But business logic often uses data from the database or calls external APIs

Mocking Dependencies

Replace database access with a mock

- Return hard-coded values

Fast and repeatable unit tests

Simulate edge cases and errors

- e.g. database can't be accessed
- e.g. no tax rate available for a country

Use dependency injection to enable mocking external dependencies

Dependency Inversion Principle

High-level modules shouldn't depend directly on low-level modules. Instead, both should depend on abstractions (e.g. interfaces).

Unmockable Dependencies

```
public decimal CalculateTax(Order order)
{
    using (var db = new DatabaseConnection(connectionString))
    {
        var taxRate = db.Query("SELECT TaxRate FROM Tax "
            + $"WHERE CountryId={order.CountryId}");
        return taxRate * order.TotalAmount;
    }
}
```

Depending on Abstractions

```
interface ISalesTaxLookup
{
    decimal GetSalesRate(int countryId);
}
```

Unit tests can use a mock implementation of ISalesTaxLookup

Production code uses real implementation of ISalesTaxLookup

```
public decimal CalculateTax(Order order)
{
    var taxRate = salesTaxLookup.GetSalesRate(order.CountryId);
    return taxRate * order.TotalAmount;
}
```

Depending on abstractions also simplifies the consuming code

Creating Mock Implementations

```
public class MockSalesTaxLookup : ISalesTaxLookup
{
    public decimal GetSalesRate(int countryId) => 0.15;
}
```

```
var mockLookup = new Mock<ISalesTaxLookup>();
mockLookup.Setup(s => s.GetSalesRate(5)).Returns(0.15);
```

https://github.com/moq/moq4

Arrange, Act, Assert

Arrange

Create the object to be tested

e.g. new SalesTaxCalculator()

Create any necessary mocks

Keep it short

Act

Perform the action to be tested

e.g. calculator.CalculateTax(order)

Tests a single specific business rule

Ideally a single line of code

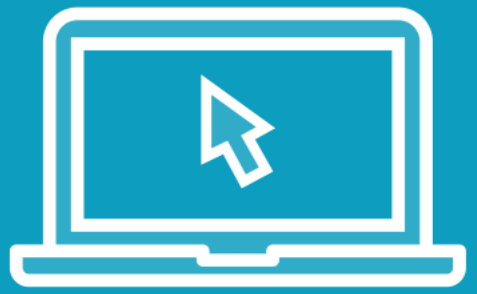
Assert

Expected return value

Expected exceptions

Expected invocations on mocks

Demo



eShopOnContainers unit tests

Running from command line

Running from Visual Studio

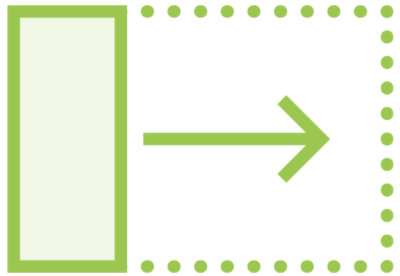
```
var order = new OrderBuilder(address)
            .AddOne(1,"cup",10.0m,0,string.Empty)
            .AddOne(1,"cup",10.0m,0,string.Empty)
            .Build();
```

Test Data Builder

Use a fluent syntax to construct the object under test

Makes unit tests easier to understand

Code Coverage



Percentage of lines of code covered by tests

Should we aim for 100% coverage?

Unit tests focus on business logic

- Some parts of the code are better tested by integration tests

Aim for complete coverage of domain logic

Can be measured with code coverage tools

Consider capturing code coverage metrics as part of your automated builds

Is 100% Coverage Sufficient?

```
public decimal CalculateTax(Order order)
{
    ISalesTaxLookup salesTaxLookup = null;
    if (order.IsBusinessCustomer)
    {
        salesTaxLookup = new BusinessTaxLookup();
    }
    var taxRate = salesTaxLookup.GetSalesRate(order.CountryId);
    return taxRate * order.TotalAmount;
}
```

Aim for 100% code coverage of business logic with unit tests

Why do many projects have low unit test code coverage?

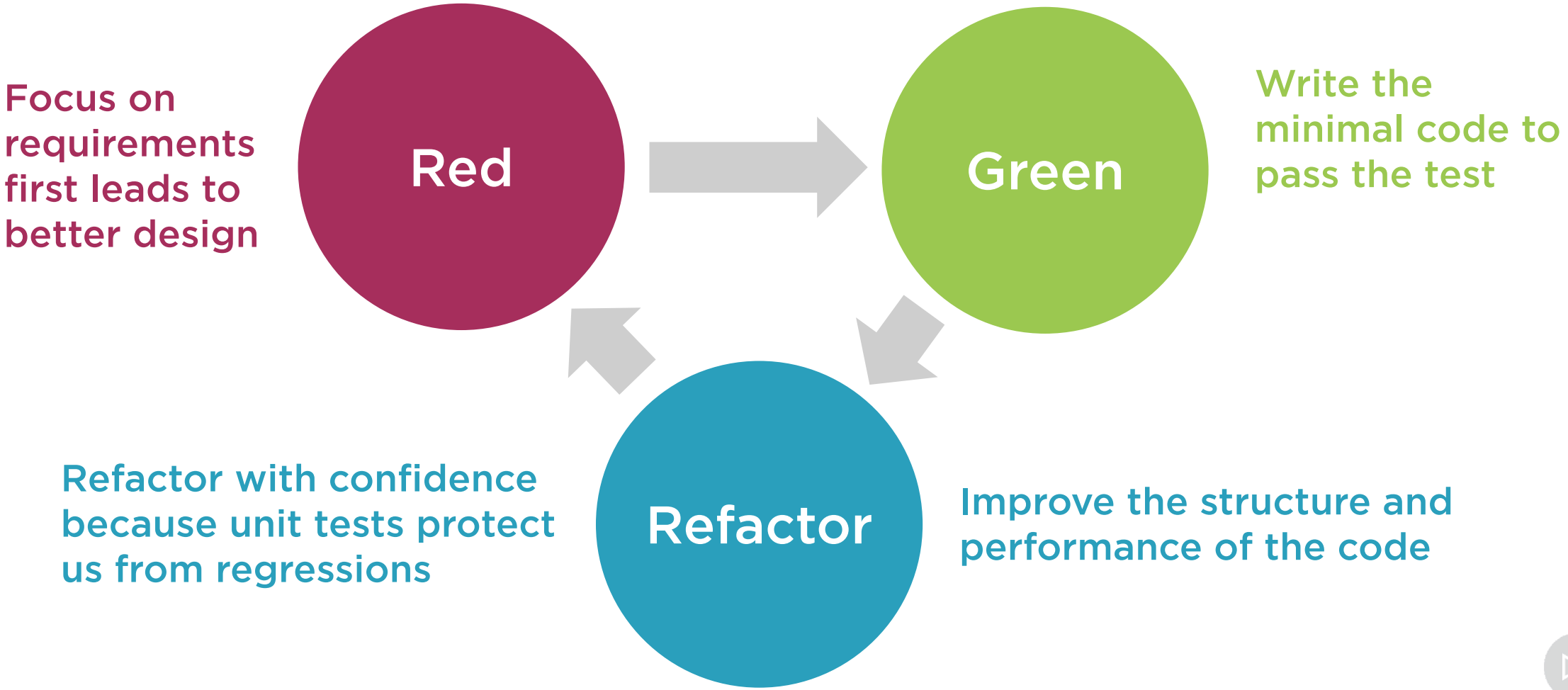
Writing unit tests last leads to poor code coverage.

"I don't have enough time to write the tests!"

"This code is too hard to write unit tests for!"

Test-driven Development

Write the unit tests first, before implementing business logic



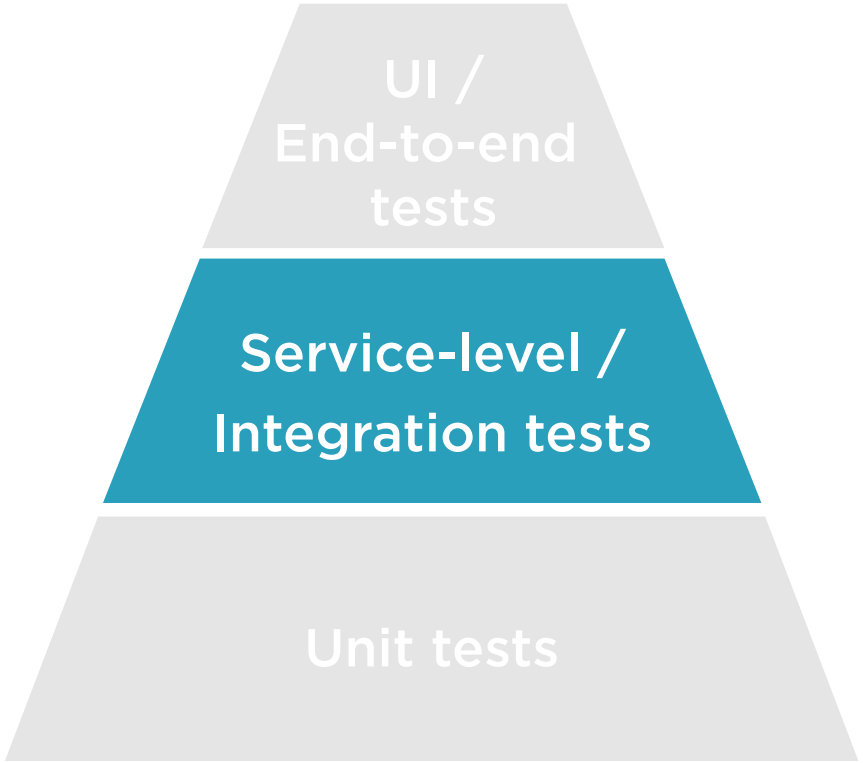
Getting good at TDD takes practice: don’t give up too quickly!

Service-level tests

Also known as “integration tests”

Tests a single microservice in isolation from the others, by making calls to its public API

Integration Tests



Out of process

- Called from the outside
- HTTP requests
- Publishing messages

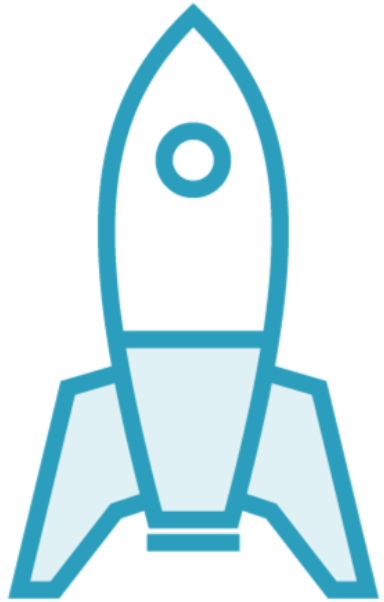
Talk to a real database

- Verify the data access layer

Mock external services

- e.g. sending emails or taking payments

Running Integration Tests



Run locally

- Start an instance of the microservice
- Send it HTTP requests
- Validate the responses

Running in the cloud or on-premises

- CI server triggers a deployment
- Automate the service-level tests

Benefits of Service-level Tests

Validate the data access layer

Ensure security is correctly configured

e.g. Call every endpoint without authorization headers and validate that it rejects the request

Benefits of Service-level Tests

Validate the data access layer

Ensure security is correctly configured

Measure performance with load tests

Simulate large amounts of traffic hitting your microservice

Gradually ramp up traffic until you find the breaking point

Easily identify where performance bottlenecks are

Benefits of Service-level Tests

Validate the data access layer

Ensure security is correctly configured

Measure performance with load tests

Maintain backwards compatibility

Demo

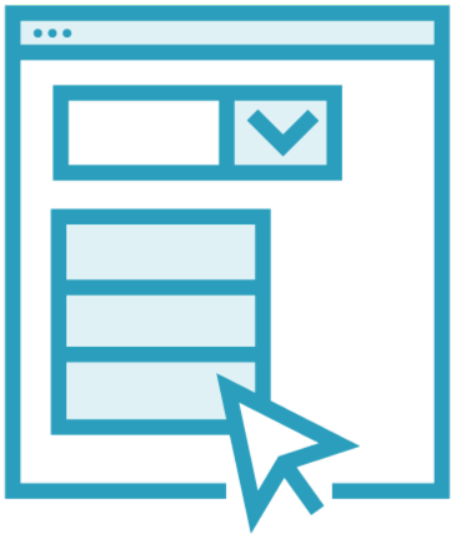
eShopOnContainers integration tests

Containerizing your integration tests simplifies running locally and on a CI build server

End-to-end tests

Test the entire microservices application. Usually driven through the user interface.

UI Tests



Automate user actions in the front-end

Can be difficult to maintain

- Frameworks like Selenium can help
- Simulate user actions in a browser
- e.g. visit store website and purchase an item

Slow to run

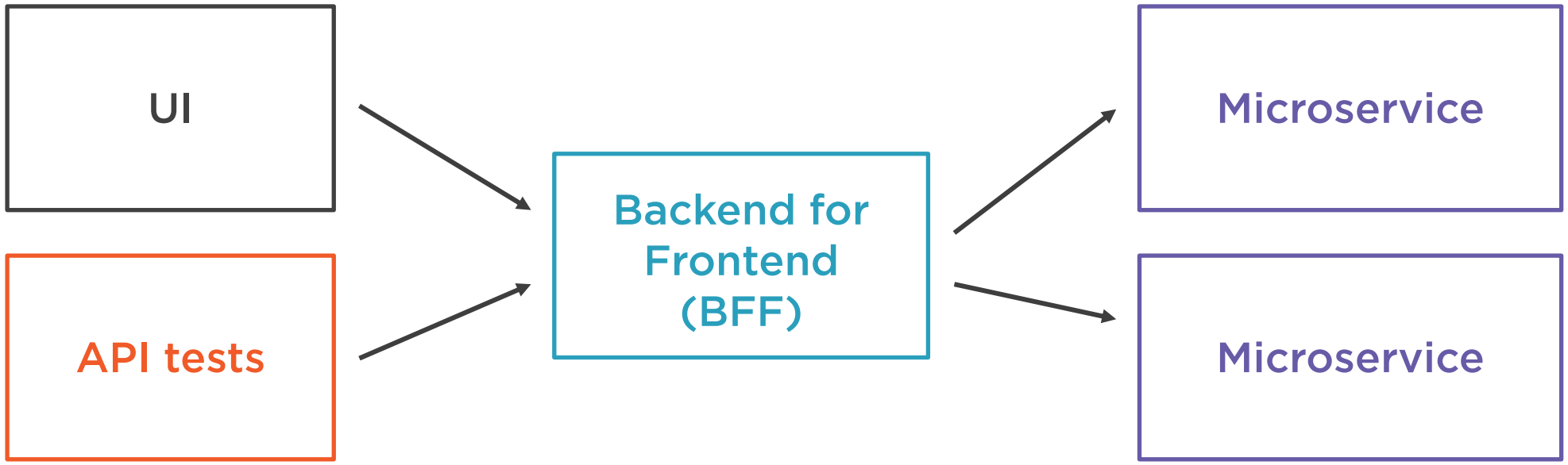
- Wait for web-pages to load

Can be fragile

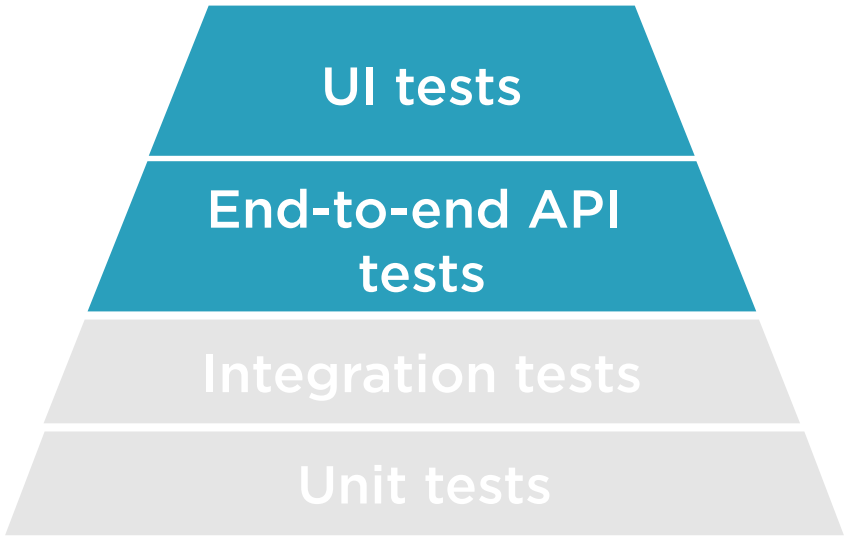
- Broken by different HTML structure

Not all UI problems are easily detected by automated UI tests

End-to-end tests don’t necessarily need to automate the UI



End-to-end API Tests



Test as much as possible through the BFF APIs

Fewer UI tests are needed

- Automate a few key journeys through the website
- “Smoke tests” quickly check that the application isn’t completely broken

Penetration Testing

Performed by security experts

- Attempt to breach your security defenses
- Test against a wide variety of known exploits and vulnerabilities

Test against a production-like deployment

Automate the running of all the tests you create

Automate Your Tests

Unit tests

Run in development environment

Run in every CI build

Fail the build if the tests fail

Service-level tests

Containerizing tests allows them to run on a CI build machine

Automate deployment to the cloud

End-to-end tests

Continuous deployment

Run long-running tests overnight

Don’t leave it too long between test runs

Summary

Test pyramid

Unit tests

- Cover business logic
- Rapid feedback

Test-driven development

Service-level (integration) tests

End-to-end / UI tests

Automate your tests

Deploy to production with confidence

Up next...

Authenticating and authorizing microservices