Testing

Unit Tests, Mocking, Integration Tests, Selenium

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Have a Question?

Software University

sli.do

#csharp-web

Table of Contents

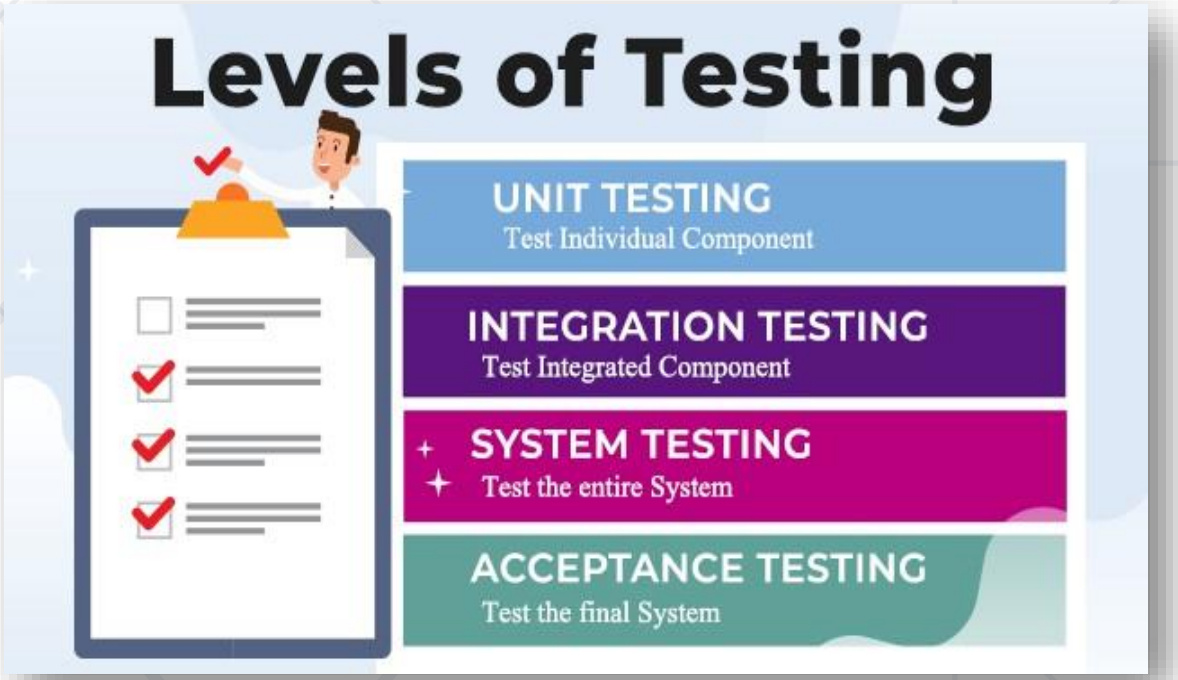
Testing

Testing

- **Testing** is an important part of the application lifecycle
  - New features need to be verified, before delivered to the clients
  - Testing checks whether software **conforms to the requirements**, aims to **find defects**
- **Testing** is a wide area of application development
  - There are several **levels** of testing, many **concepts** of development and **different types** of testing
- **Web applications** also need testing for "**unintentional features**"
  - Controllers, Services, Custom Components, etc.
  - Every component of the application must be tested

Main Test Levels

- Different **components** of the application are tested on different levels:
  - **Unit tests**
    - Test single component
    - Created by developers
  - **Integration tests**
    - Test interaction between components (e. g. APIs)
    - Created by developers / QA automation engineers
  - **System tests** / **end-to-end tests**
    - Test the entire system (created by QA automation engineers)



More Test Levels

Software University

Test Level	Description
Unit Testing	Tests individiual parts of the code, independent from the infrastructure
Component Testing	Testing of multiple functionalities (a single component)
Integration Testing	Testing of all integrated modules to verify the combined functionality
System Testing	Tests the system as a whole, once all the components are integrated
Regression Testing	Ensures that a fixed bug won't happen again
Acceptance Testing	Tests if the product meets the client's requirements. Purely done by QAs
Load / Stress Testing	Test the application's limits by attempting large data processing and introduci ng abnormal circumstances and conditions
Security Testing	Test if the application has any security flaws and vulnerabilities
Other Types of Testing	Manual, automation, UI, performance, black box, end-to-end testing, A/B, etc

Test Concepts

- There are also different concepts and practices of test development
  - **Code-first** approach (The usual Development)
  - **Test-first** approach (Test-Driven Development)
- Each has its own **advantages** and **disadvantages**
  - The **Code-first** approach ensures **flexibility** & **fast** development
  - The **Code-first** approach tends to create **tests based on the code**
  - The **Test-first** approach ensures **quality** and **edge case coverage**
  - The **Test-first** approach is **complicated** and is an "**initial delay**"

The "AAA" Testing Pattern

- Automated tests usually follow the "**AAA**" **pattern**
  - **Arrange**: prepare the **input** data and entrance conditions
  - **Act**: invoke the **action** for testing
  - **Assert**: check the **output** and exit conditions

```
[Test]
public void Test_SumNumbers()
{
    // Arrange
    var nums = new int[] {3, 5};

    // Act
    var sum = Sum(nums);

    // Assert
    Assert.AreEqual(8, sum);
}
```

Unit Testing

Unit Testing

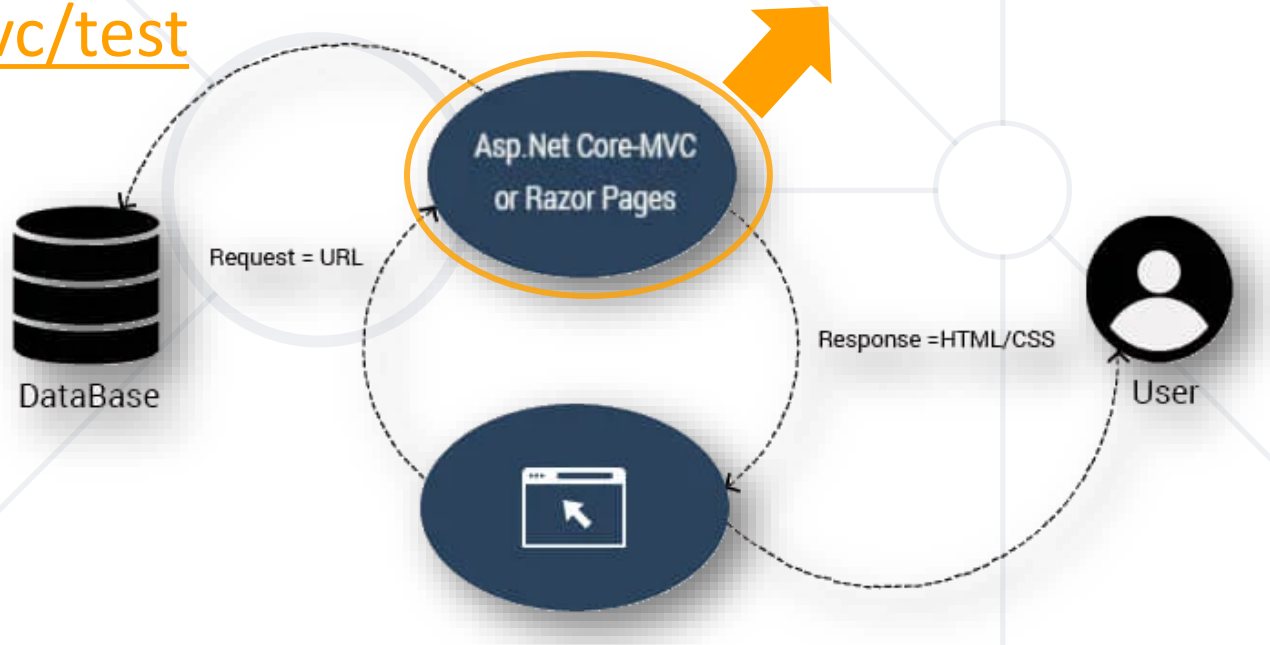
- **Unit Testing** web apps is pretty much like casual unit testing
  - Writing test methods to test classes and methods (functionalities)
    - Testing individual code components (**units**), not the **infrastructure**
  - You still use the same testing frameworks as in casual unit testing

```
public class Summator
{
   public int Sum(int[] arr)
   {
      int sum = arr[0];
      for (int i = 1; i < arr.Length; i++)
         sum += arr[i];
      return sum;
   }
}
```

```
void Test_SumTwoNumbers() {
  var summator = new Summator();
  if (summator.Sum(new int[]{1, 2}) != 3)
    throw new Exception("1+2 != 3");
}
```

Unit Testing in ASP.NET Core

- When using a web frameworks such as **ASP.NET Core**
  - Built-in logic does not need to be tested
    - It is already tested during the development of the framework itself
    - Example of MVC tests: https://github.com/dotnet/aspnetcore/tree/main/src/Mvc/test
  - You still need to test your **custom functionality**



NUnit

Unit-Testing Framework for All .NET Languages

NUnit: Overview

Software University

- **NUnit** == popular **C# testing framework**
  - Supports test suites, test cases, before & after code, startup & cleanup code, timeouts, expected errors, …
  - Like **JUnit** (for Java)
  - Free, open-source
  - Powerful and mature
  - Wide community
  - Built-in support in Visual Studio
  - Official site: nunit.org

```
using NUnit.Framework;
0 references
public class SummatorUnitTests
{
    [Test]
    0 references
    public void Test_SumTwoNumbers()
    {
        var summator = new Summator();
        var sum = summator.Sum(new int[] { 1, 2 });
        Assert.AreEqual(3, sum);
    }
}
```

NUnit Assertions

- Assert that **condition** is true

```
Assert.That(bool condition);
```

- **Comparison** (equal, greater than, less than or equal, …)

```
Assert.AreEqual(expected, actual);
```

- **Range** assertions

```
Assert.That(number, Is.InRange(80, 100));
```

NUnit Assertions

- **String** assertions

```
Assert.That(string actual, Does.Contain(string expected));
```

- Assert if a string or collection is empty

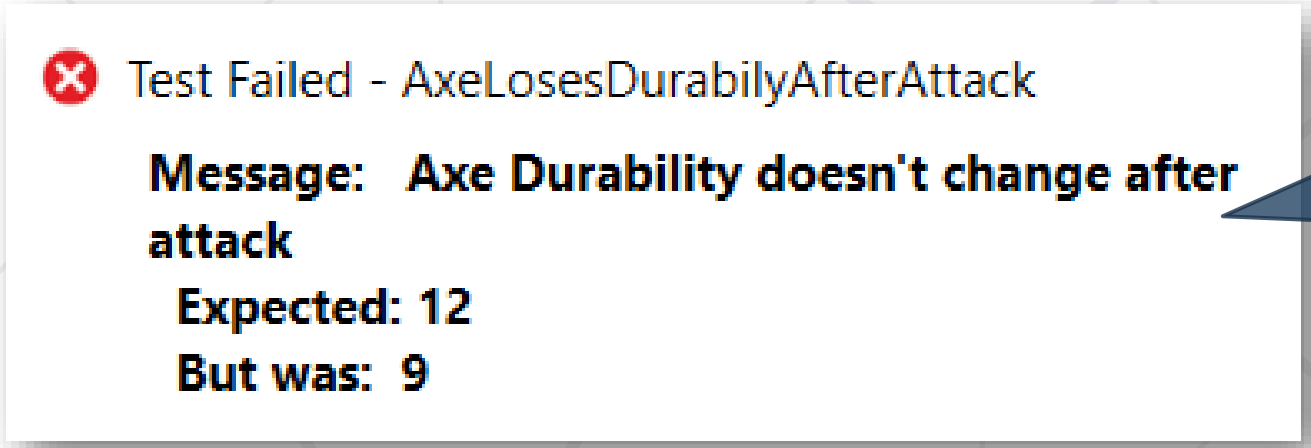
```
Assert.IsEmpty(string message);
Assert.IsEmpty(Ienumerable collection);
```

- And many more
  https://docs.nunit.org/articles/nunit/writing-tests/assertions/assertion-models/classic.html

NUnit Assertion Messages

- Assertions can **show messages** to helps with **diagnostics**

```
Assert.That(axe.DurabilityPoints, Is.EqualTo(12),
    "Axe Durability doesn't change after attack");
```



Failure messages in the tests help finding the problem

Test Classes and Test Methods

Software University

- Test **classes** hold test **methods**

```
using NUnit.Framework;

[TestFixture]
public class SummatorTests
{
  [Test]
  public void Test_SumTwoNumbers() {
    var summator = new Summator();
    var sum = summator.Sum(new int[] { 1, 2 });
    Assert.AreEqual(3, sum);
  }
}
```

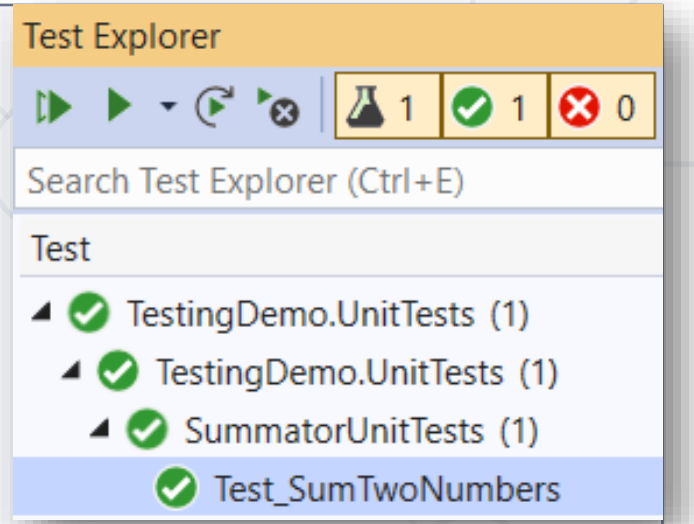
Import NUnit

Optional notation

Test class

Test method

Assertion



Initialization and Cleanup Methods

```
private Summator summator;

[SetUp] // or [OneTimeSetUp]
public void TestInitialize()
{
  this.summator = new Summator();
}

[TearDown] // or [OneTimeTearDown]
public void TestCleanup()
{
  // …
}
```

Executes before each test

Executes after each test

Mocking

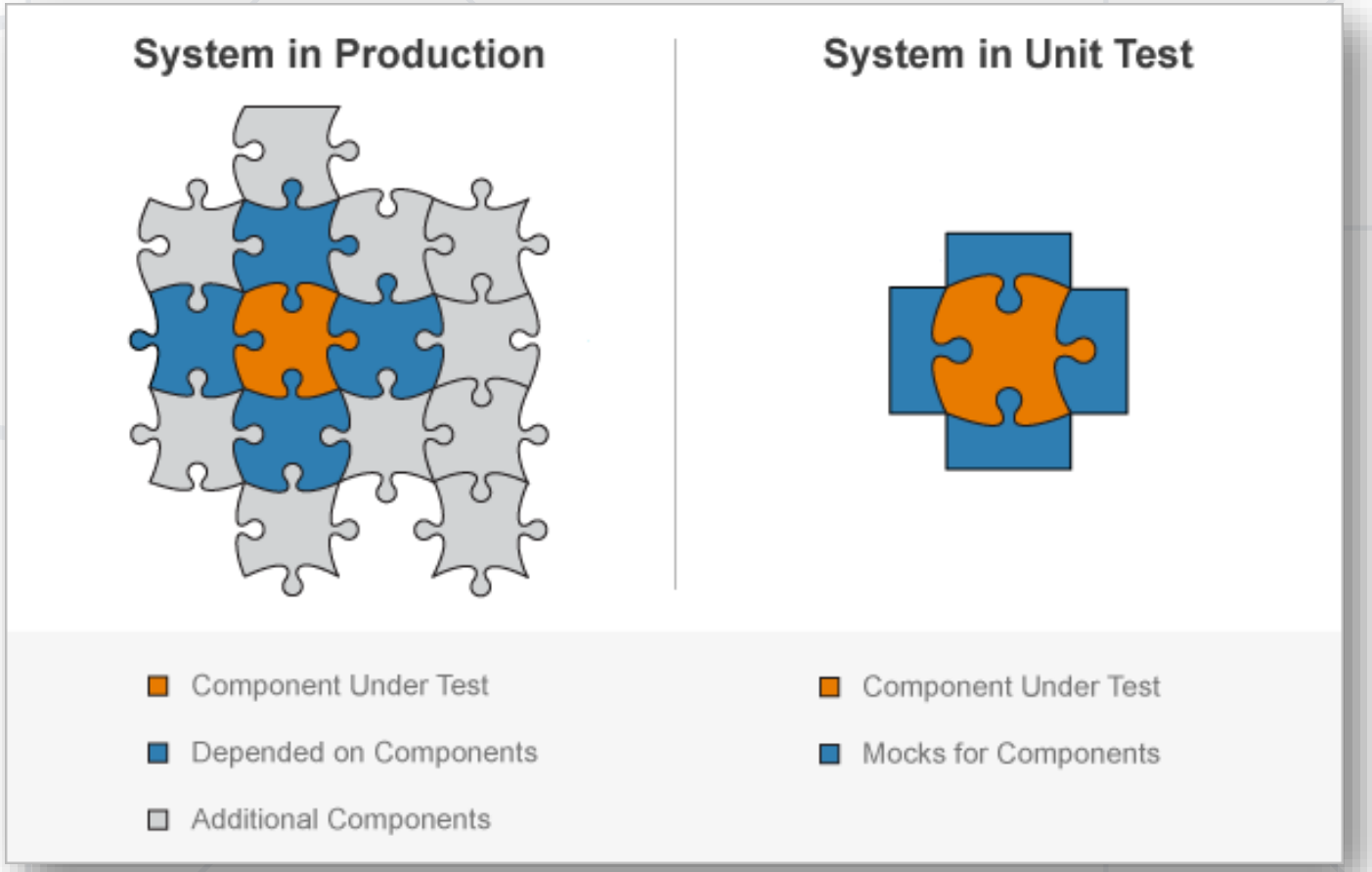
Simulating External Dependencies in Unit Tests

What is Mocking?

- **Mocking** – something made as an imitation
- **Mocking** is a software practice, primarily used in **Unit Testing**
  - An object under test may have **dependencies** on other objects
  - To **isolate** the behavior, the other objects are replaced
    - The replacements are **mocked objects**
    - The mocked objects **simulate** the behavior of the **real objects**
  - Useful if the real objects are **impractical**/**incorporate** to the unit test
- Basically, **Mocking** is creating objects that **simulate behavior**
- In .NET, we can use the **Moq** library to create mock objects

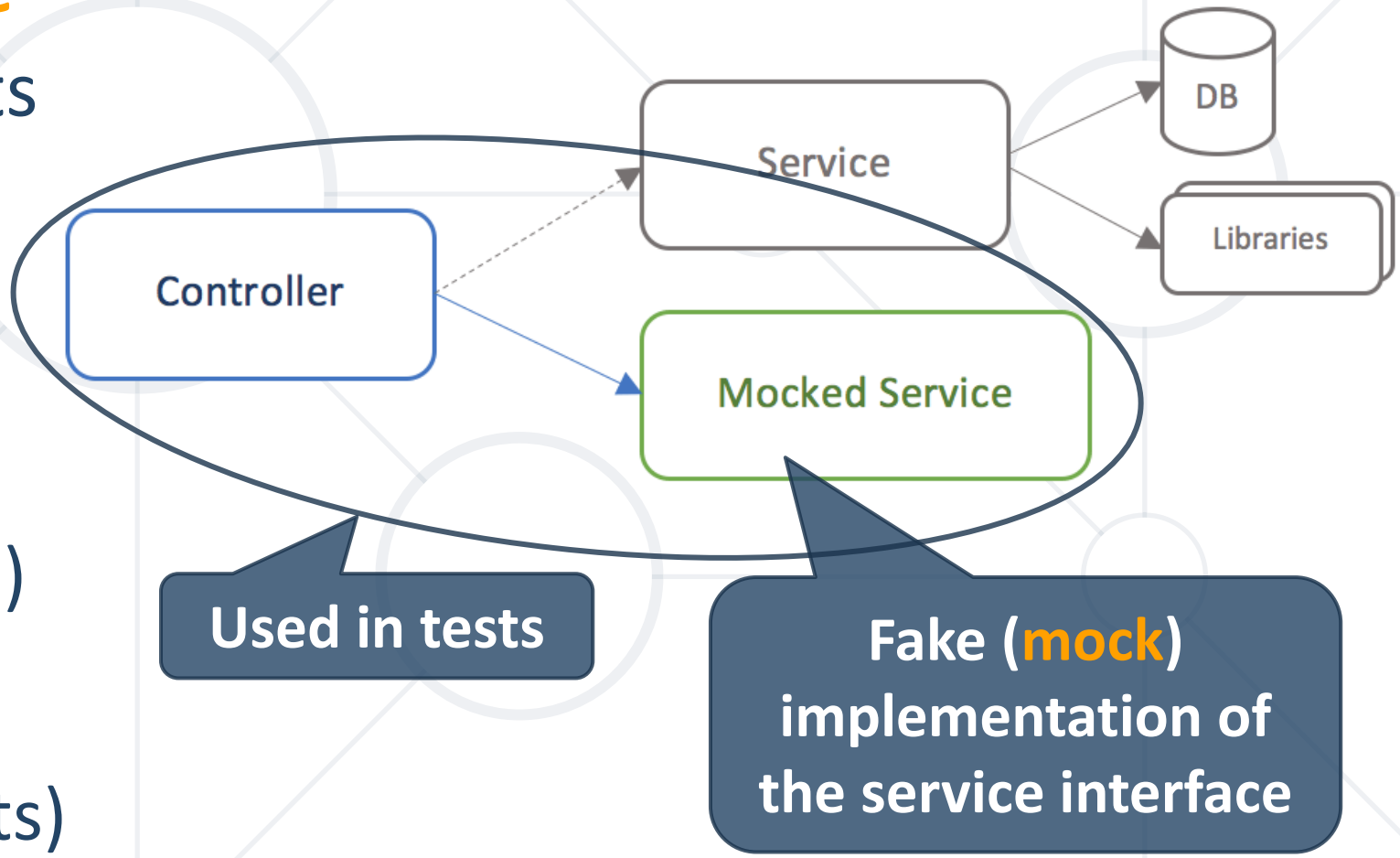
Why Mocking?

- **Unit testing** should **test a single component**
  - Isolated from all the others
- **External dependencies** should be **mocked** (faked, simulated)



Mocking

- **Mocking simulates the behavior** of real objects
  - Usually done through interfaces
  - Real implementation (e.g., with a database)
  - Fake implementation (used for the unit tests)



Testing a Service with Mocking

Software University

```
public interface ICreditDecisionService
{
    string GetDecision(int creditScore);
}
```

```
public class CreditDecision
{
    ICreditDecisionService service;
    public CreditDecision(ICreditDecisionService service)
        => this.service = service;

    public string MakeCreditDecision(int creditScore)
            => service.GetDecision(creditScore);
}
```

```
public class CreditDecisionService
        : ICreditDecisionService
{
    public string GetDecision
        (int creditScore)
    {
        if (creditScore < 550)
            return "Declined";
        else if (creditScore < 675)
            return "Maybe";
        else
            return "Absolutely";
    }
}
```

Testing a Service with Mocking

```
[TestFixture]
public class CreditDecisionTests
{
    [Test]
    public void MakeCreditDecision_ShouldReturnCorrectResult()
    {
        var mockCreditDecisionService = new Mock<ICreditDecisionService>();
        mockCreditDecisionService
            .Setup(p => p.GetDecision(100))
            .Returns("Declined");

        var controller = new CreditDecision(mockCreditDecisionService.Object);
        var result = controller.MakeCreditDecision(100);

        Assert.That(result, Is.EqualTo("Declined"));

        mockCreditDecisionService.VerifyAll();
    }
}
```

NOTE: You will need the following NuGet package installed: **Moq**

Mock the service

Use the mocked service

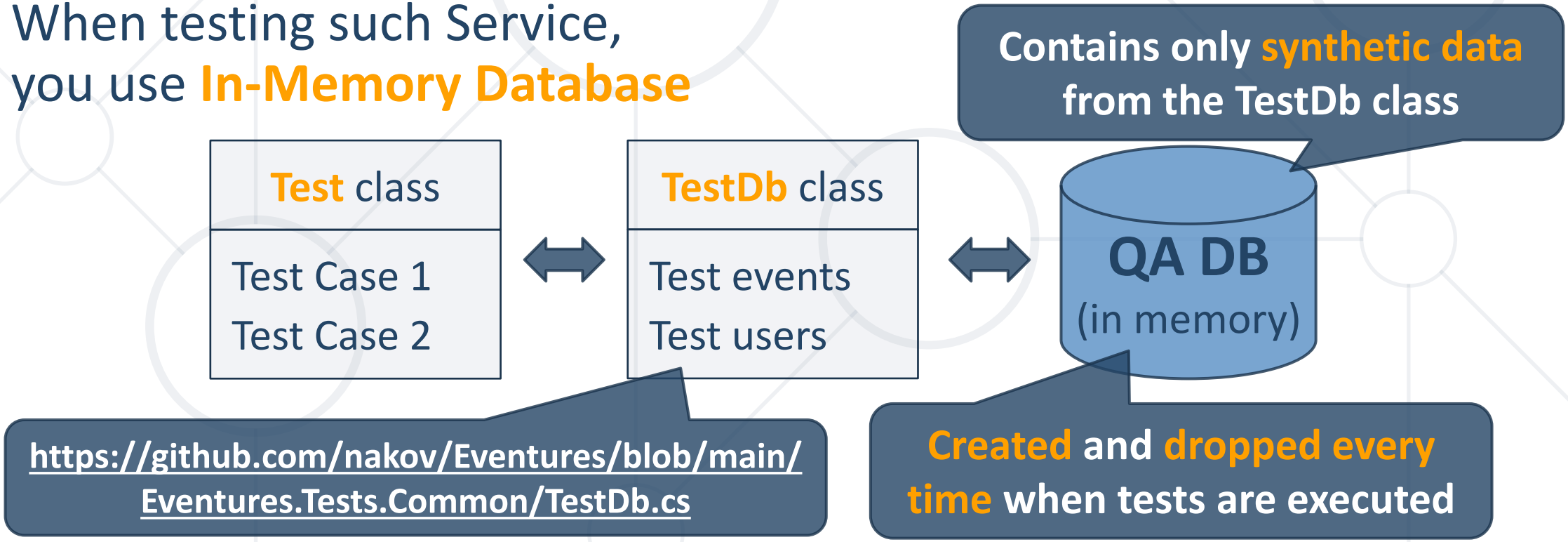
You can look at a more complex example here

In-Memory Database

Data Storage in Memory

In-Memory Database

- **Services**, accessing a **Database**, can (and should) also be tested
  - When testing such Services, you don't create a new Database
    - This would, otherwise, overload the Database Server
  - When testing such Service, you use **In-Memory Database**



EF Core In-Memory Database

- **EF Core** provides an **In-Memory Database**
  - Included with the `Microsoft.EntityFrameworkCore.InMemory` package
  - Its general purpose is to be a **testing database**

```
public class CreditDecision
{
    public int Id { get; set; }
    public int Score { get; set; }
    public string Decision { get; set; }
}
```

Create a **db context** as usual

```
public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options)
        : base(options) {}

    public DbSet<CreditDecision> CreditDecisions { get; set; }
}
```

Testing a Service with In-memory Database

Software University

```
public interface ICreditDecisionService
{
    CreditDecision GetById(int id);
}
```

```
public class CreditDecisionService: ICreditDecisionService
{
    private readonly AppDbContext data;

    public CreditDecisionService(AppDbContext data)
      => this.data = data;

    public CreditDecision GetById(int id)
      => this.data.CreditDecisions.Find(id);
}
```

Inject the db context to the service

Testing a Service with In-memory Database

Software University

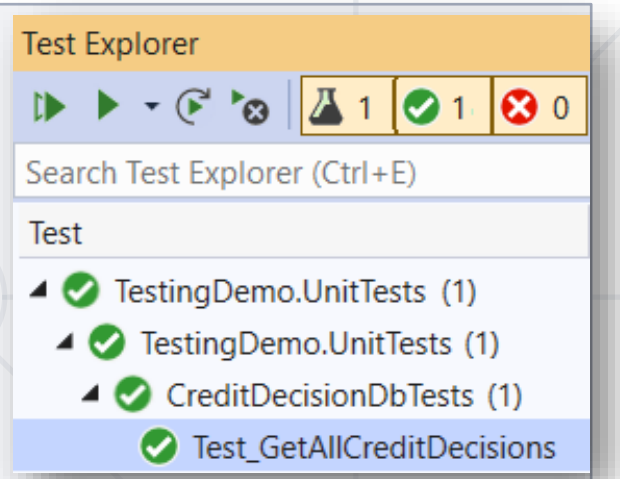
```
[TestFixture]
public class CreditDecisionDbTests
{
    private IEnumerable<CreditDecision> decisions;
    private AppDbContext dbContext;

    [SetUp] // Method is executed before tests
    public void TestInitialize()
    {
        this.decisions = new List<CreditDecision>() {
            new CreditDecision(){ Id = 1, Score = 100, Decision = "Declined" },
            new CreditDecision(){ Id = 2, Score = 600, Decision = "Maybe"  },
            new CreditDecision(){ Id = 3, Score = 800, Decision = "Absolutely" }};

        var options = new DbContextOptionsBuilder<AppDbContext>()
            .UseInMemoryDatabase(databaseName: "CreditsInMemoryDb") // Use an in-memory DB
            .Options;

        this.dbContext = new AppDbContext(options);
        this.dbContext.AddRange(this.decisions); // Add data to the DB
        this.dbContext.SaveChanges();
    }

    ...
}
```

Create data for the in-memory db

Testing a Service with In-memory Database

Software University

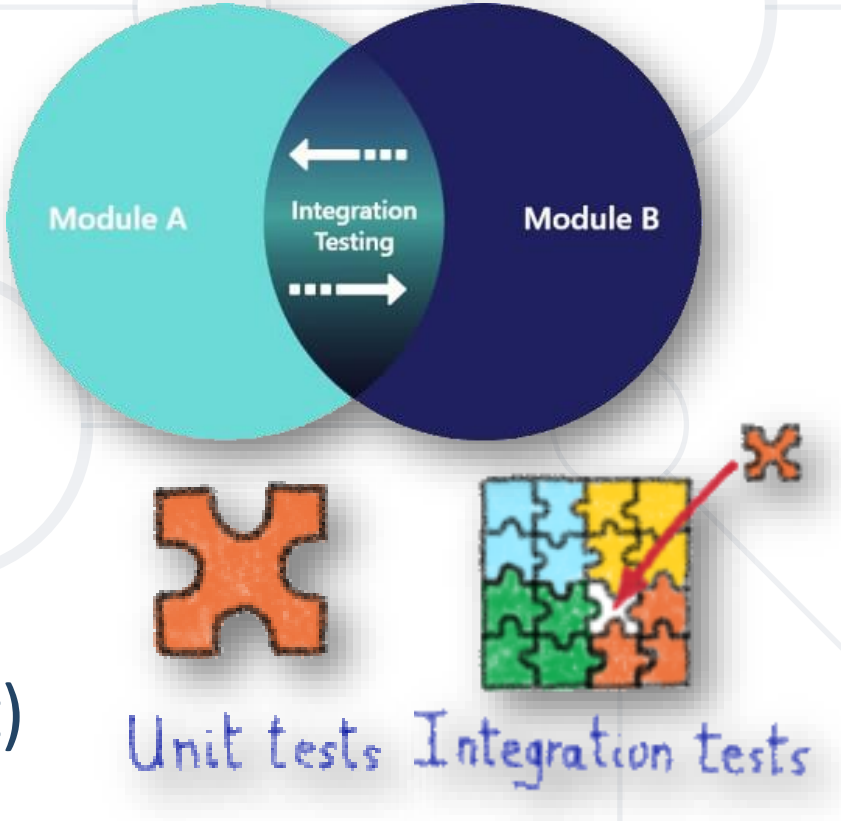
```
[TestFixture]
public class CreditDecisionDbTests
{
    ...

    [Test]
    public void Test_GetAllCreditDescisions()
    {
        var decisionId = 1;

        ICreditDecisionService service =
            new CreditDecisionService(this.dbContext); // Pass the in-memory db to the service
        var decision = service.GetById(decisionId); // Use service methods

        var dbDecision = this.decisions
            .ToList()
            .Find(d => d.Id == decisionId);

        // Compare and assert retuned CreditDecision from DB and from the service method
        Assert.True(decision != null);
        Assert.True(decision.Score == dbDecision.Score);
        Assert.True(decision.Decision == dbDecision.Decision);
    }
}
```



Integration Testing

Integration Testing

- **Integration testing** tests several units (components) together
  - Aims to expose faults in the **interaction between integrated units**
  - Example: test user registration + data access services + database storage (if the new user is stored in the DB)
- **Unit testing** vs. **integration testing**
  - Integration testing tests the interaction between several units
  - Unit testing tests a single unit (component)



Integration Testing

- **ASP.NET Core** supports **Integration Testing** using a **Unit Test framework**
  - The framework has an integrated web host and in-memory test server
- **Integration Tests** follow a sequence of events
  - The app's web host must be configured
  - A test server client is created to submit requests to the app
  - [**Arrange**] The test app prepares a request
  - [**Act**] The client submits the request and receives a response
  - [**Assert**] The actual response is validated based on expected result
  - After all tests have run, the results are reported

Integration Test for the TaskBoard App

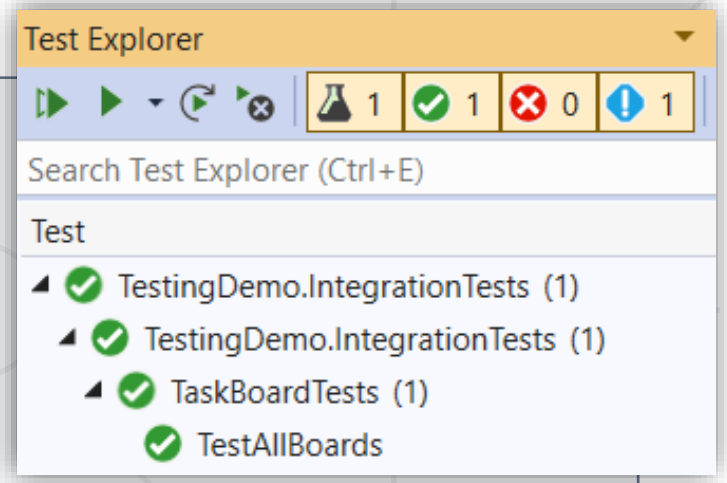
Software University

```
[Test]
public async Task TestAllBoards()
{
    // Arrange
    var httpClient = new HttpClient();

    // Act
    var response = await httpClient
        .GetAsync("https://taskboard.nakov.repl.co/boards");

    // Assert
    Assert.AreEqual(HttpStatusCode.OK, response.StatusCode);
}
```

Send a GET request

Sends HTTP request and receives HTTP responses

Make assertions



Selenium

Selenium

- **Selenium** is a portable **testing** framework for web applications
  - Provides a playback tool for authoring tests (Selenium IDE)
  - Provides a test domain-specific language (Selenese)
  - Provides "browser driving" natively (Selenium WebDriver)
- "**Selenium automates browsers. That's it!**" Selenium docs
  - Automates web applications for test purposes
  - Useful for integration testing SPA apps

Install and Set-up Selenium

- Install latest stable Node.js: https://nodejs.org/en/
  - Install the npm package: `npm install -g selenium-standalone`
- Install latest Java: https://www.java.com/en/download/
- Download Selenium Standalone Server JAR
  - https://www.seleniumhq.org/download/
- Download ChromeDriver (to match your Chrome version)
  - http://chromedriver.chromium.org/downloads
  - Extract the file in the same folder as Selenium Standalone Server
- Start selenium server `java -jar .\selenium-server-standalone-3.141.59.jar`

Selenium and .NET

- Let's get ready Selenium for ASP.NET Core App Testing
  - Install NuGet packages:
    - `Selenium.Support`
    - `Selenium.WebDriver`

MyTested.AspNetCore.Mvc

MyTested.AspNetCore.Mvc

Software University

- **MyTested.AspNetCore.Mvc** is a **powerful** testing library
  - Automatic resolving of test dependencies
  - Fluent API with strongly-typed extensions
  - Unit tests, integration tests, route tests, everything covered
  - Built-in mocks for every ASP.NET Core scenario
    - Authentication, DbContext, HTTP, Session, Caching, and many more…
  - Examine at https://github.com/ivaylokenov/MyTested.AspNetCore.Mvc
  - Latest available version:

```
Install-Package MyTested.AspNetCore.Mvc.Universe -Version 3.1.2
```

MyTested.AspNetCore.Mvc

Software University

```
// Instantiates controller with the registered global services,
// and mocks authenticated user,
// and tests for valid model state,
// and tests for added by the action view bag entry,
// and tests for view result and model with specific assertions
MyController<MyMvcController>
    .Instance(instance => instance
        .WithUser(user => user.WithUsername("MyUserName")))
    .Calling(c => c.MyAction(myRequestModel))
    .ShouldHave()
    .ValidModelState()
    .AndAlso().ShouldHave()
    .ViewBag(viewBag => viewBag.ContainingEntry("MyViewBagProperty", "MyViewBagValue"))
    .AndAlso().ShouldReturn()
    .View(result => result
        .WithModelOfType<MyResponseModel>()
        .Passing(model =>
        {
            Assert.AreEqual(1, model.Id);
            Assert.AreEqual("My property value", model.MyProperty);
        }));
```

Summary

- **Testing**
- **Unit Testing** – testing a single unit
  - **NUnit** – a unit-testing framework
  - **Mocking** – simulating external dependencies
  - **In-memory** database
- **Integration Testing** – testing a combination of units
- **Selenium** – automated tests in browser
- **MyTested.AspNetCore.Mvc**

Questions?

SoftUni

Software University

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

Finance Academy

SoftUni Diamond Partners

Software University

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, softuni.org
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – https://softuni.org
- © Software University – https://softuni.bg