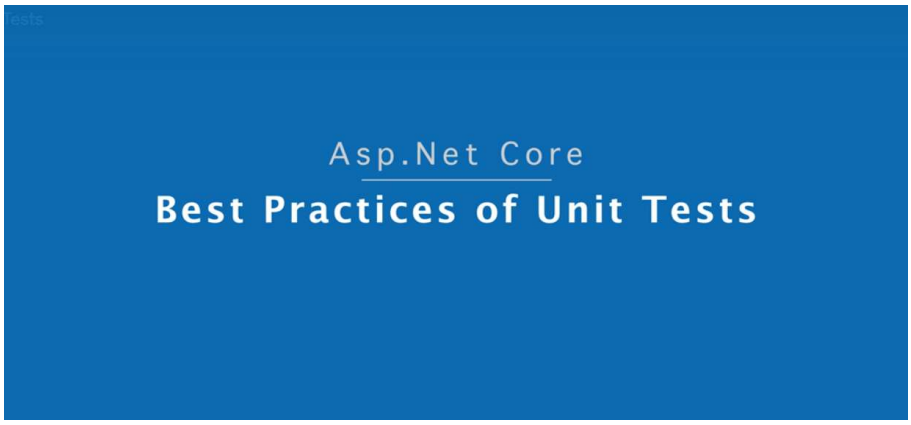
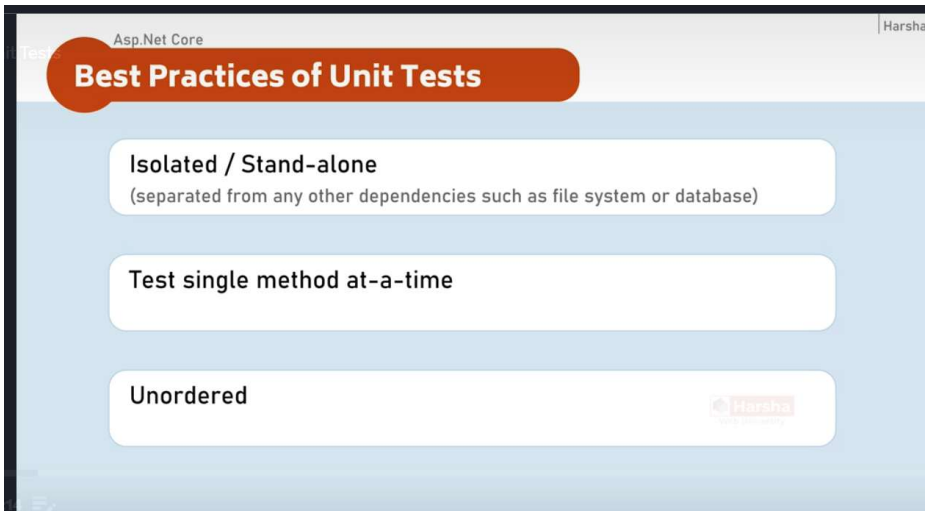
Section 19: Advanced Unit Testing - Moq & Repository Pattern [MVC and Web API]

Wednesday, February 12, 2025 10:56 AM





In this section, let us ensure that we are following all the best practices for **unit test cases**.
So, what are they?
To make your unit test cases follow **best practices** and make them **highly reusable**, we need to follow certain guidelines.

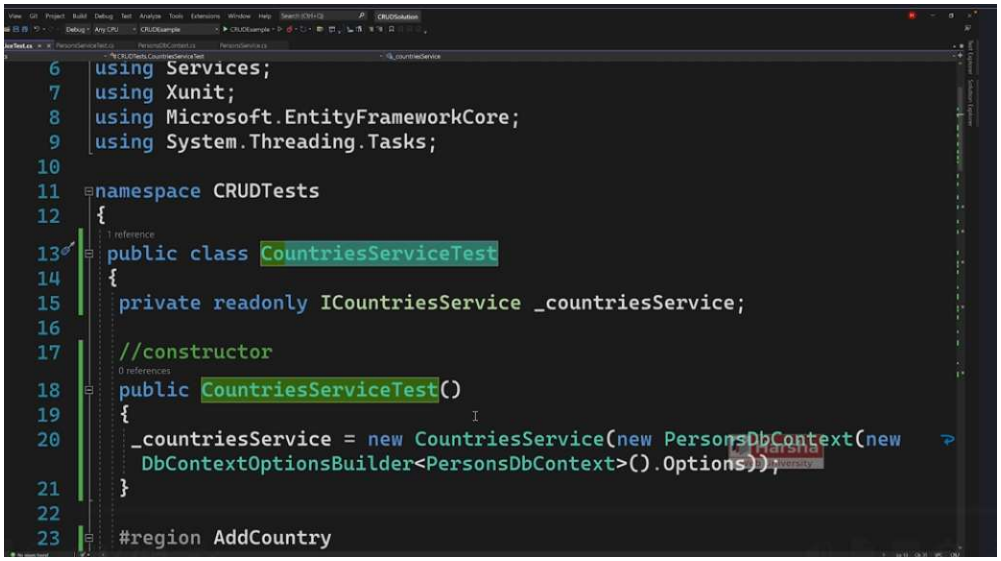
- We must design our unit test cases **completely isolated** from other classes, services, or any storage systems such as files or databases.
- Even by mistake, your unit test case should **not** hit the **file system**, **database**, or **any other additional classes**, except for the class that is being tested.

For example, let's say…

Our **CountryServiceTest** is testing the **CountryService** class.
So, except for **CountryService**, no other file system or database system should be accessed.
But currently, our unit test case tries to access the **database** using **Entity Framework**, which is **not a good practice** for unit testing.
The goal of a **unit test case** is to **test an individual class**, so it should **not** test anything beyond that.

In this section, we will ensure that we are **testing only a single class** in our unit tests, without any touchpoints with the **file system** or **databases**.

Additionally, we will **test one method at a time**.
Your unit test case should **not** access more than one method in the same test case, except for the one you are specifically trying to test.

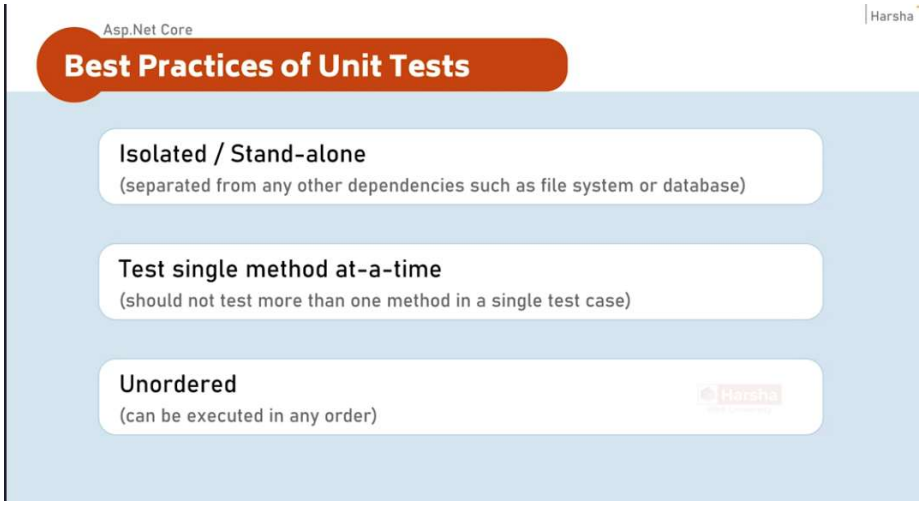


For example, let's say we have a unit test case called **GetAllCountries**.
If you look into the code, our goal is to **test only** the **GetAllCountries** method.
So, except for that method, we **should not** call any other methods.
But in this case, we are calling the **AddCountry** method a few times, which is **not** a good practice.
We should **limit** the test case to calling only **GetAllCountries**, like this.
But **practically**, how is it possible to get the **countries** without adding them?

Because every unit test cases executes independently. It is possible to solve this problem with mocking and we will try to ensure that we practice in this section.

```
List<CountryResponse> countries_list_from_add_country = new
 List<CountryResponse>();

foreach (CountryAddRequest country_request in country_request_list)
{
 countries_list_from_add_country.Add(await
  _countriesService.AddCountry(country_request));
}

List<CountryResponse> actualCountryResponseList = await
 _countriesService.GetAllCountries();

//read each element from countries_list_from_add_country
```

We should **not** test or call more than **one method at a time** in the same unit test case.
Our **unit test case** should also be **completely independent** of storage devices such as **files or databases**.
Apart from these two important best practices, we should also ensure that **unit test cases are unordered**.
This means unit test cases **do not** need to be executed in the same order as they are written. They should be able to run in **any order** as needed.
Even if we execute unit test cases **alphabetically** or in **any other sequence**, they should **still pass**.
Currently, we have designed them in this way.



Best Practices for Writing Unit Test Cases

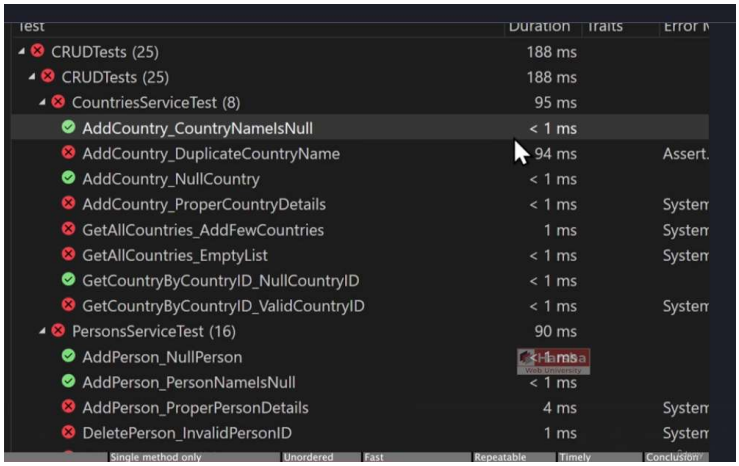
1. **Unit Tests Should Execute Quickly**
   - Each test should take only a few **milliseconds** to run.
   - For example, some tests may execute in **less than 1 millisecond**, while others take around **94 milliseconds**.
   - **None of the test cases should exceed 1000 milliseconds** (1 second).
   - If a test **takes longer than expected**, it may be **accessing an external storage device** (like a file system or database).
2. **Tests Should Be Repeatable**
   - Running the **same test multiple times** with the **same inputs** should always produce the **same results**.
   - If results **vary** between executions, there may be a **flaw in the test design** that needs to be **rechecked**.
3. **Tests Should Run in Any Order**
   - Unit tests should not have dependencies on execution order.
   - Whether executed **alphabetically, in parallel, or sequentially**, the results **must remain consistent**.
4. **Writing Tests Should Not Take Longer Than Writing Code**

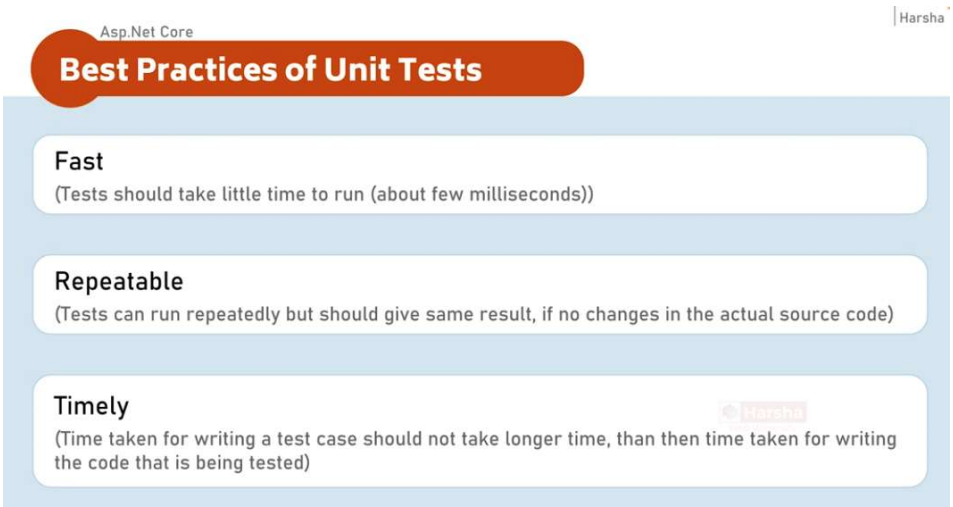
- If a service method (e.g., **GetAllCountries** or **AddPerson**) takes **5 minutes** to implement, its unit test **should not take longer** than that.
- If writing a test takes **significantly more time**, it may indicate a **flawed design**, and the test should be **simplified**.

5. **Avoid Testing Multiple Methods in One Test Case**
   - Each test should focus **only on one method** at a time.
   - Calling **multiple methods** in the **same test** violates best practices.

6. **Avoid Dependencies on Databases or Files**
   - Unit tests **should not** interact with databases, file systems, or external services.
   - The solution for this is **mocking the DB context**, which will be demonstrated in the next section.

These principles ensure that our **unit test cases are efficient, reliable, and easy to maintain**.





Understanding the Need for Mocking in Unit Tests
Issue with Current Unit Tests

- Many unit test cases **fail** due to the following error:
  InvalidOperationException: No database provider has been configured for this DB context.
- This happens because **Entity Framework Core** requires a **database provider** (such as SQL Server) when creating a DbContext instance.
- In our tests, we have **not provided** a database provider, which causes this error.

Why We Shouldn't Use a Real Database in Unit Tests

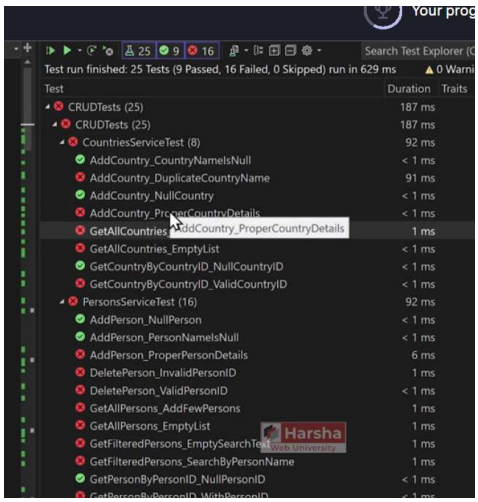
- **Unit tests should not depend on external systems** (like databases).
- Providing **SQL Server as a database provider** in tests **violates best practices**.
- **Database interactions make tests slow, unreliable, and difficult to maintain.**

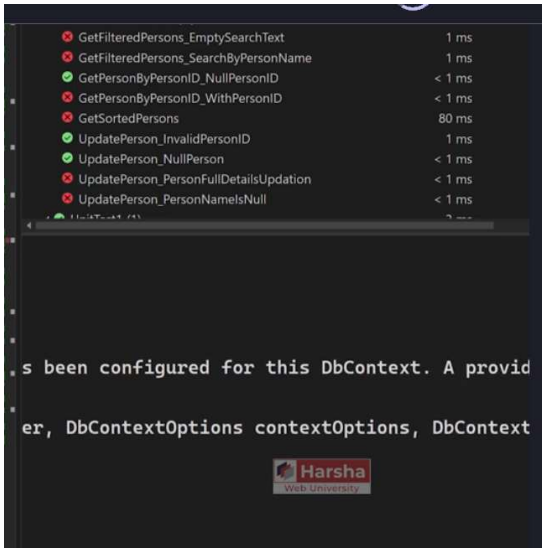
Solution: Mocking the DB Context

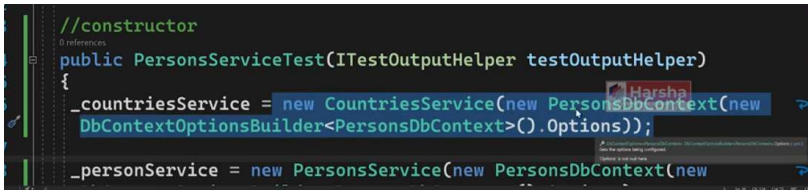
- Instead of using a **real database**, we can use a **mocked DbContext**.
- **Mocking** means creating a **fake** version of a class or dependency that behaves like the real one but doesn't require actual database operations.
- Using **mocking frameworks** (like **Moq** in .NET), we can simulate database behavior without connecting to an actual database.

Next Steps

- In the next section, we will learn how to **mock the DbContext** using Moq.
- This will ensure that our unit tests are **isolated, fast, and reliable**.



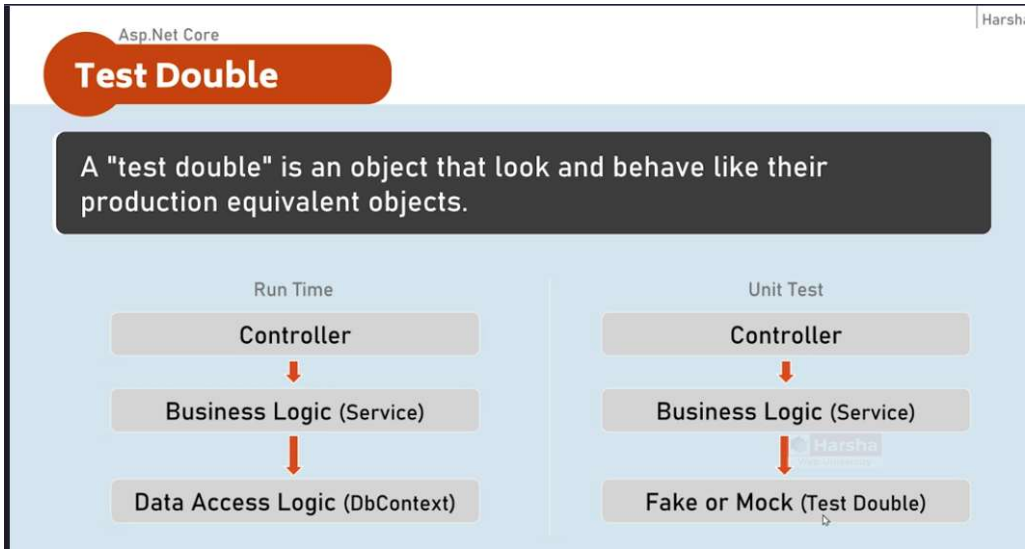




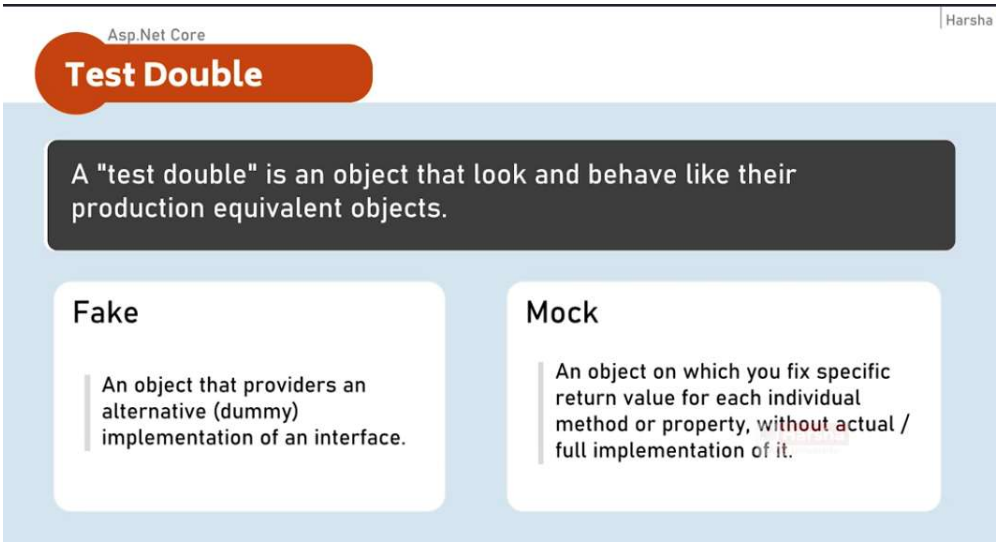
There are two things: mock or fake.
Both of these are called test doubles.
A test double is the common terminology for mocking or faking, where you create a mock or fake object for the unit test case so that the service can call it.

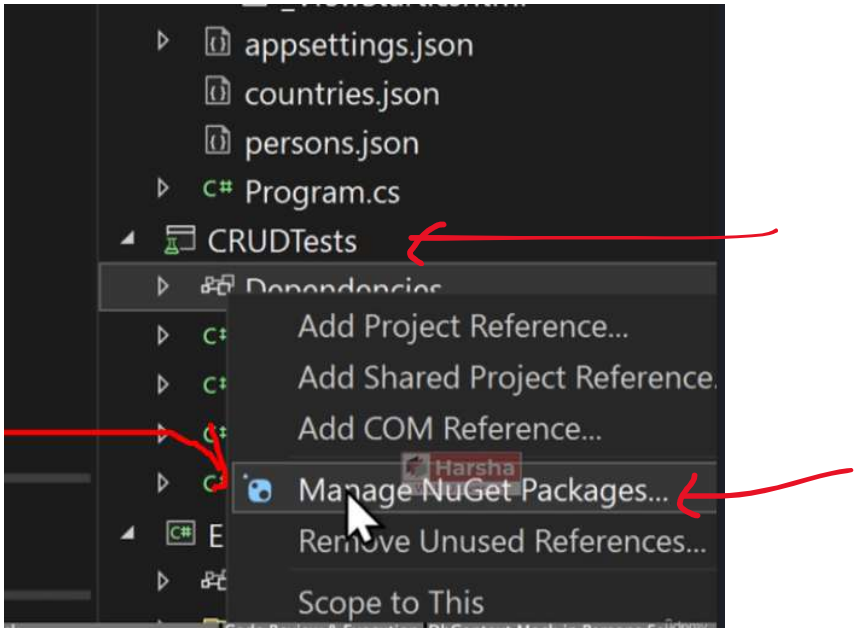
At normal runtime, meaning in the production or development environment, the business logic can call the data access logic, i.e., the DB context, directly.

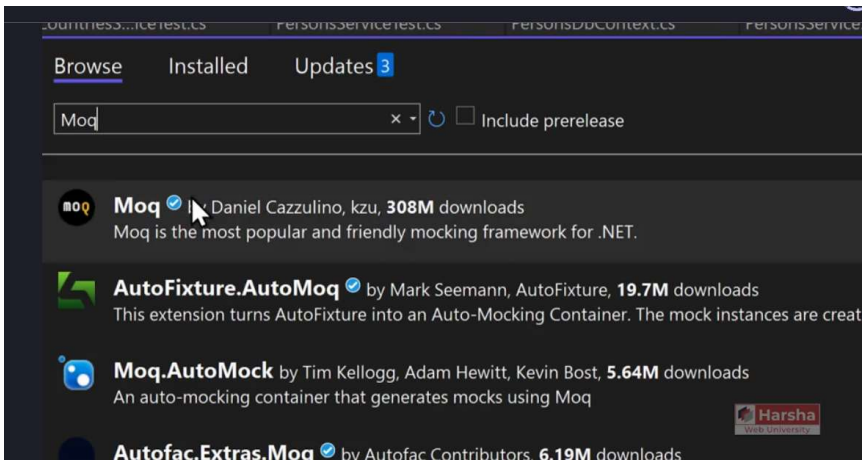
The DB context is the regular DB context that accesses the database. However, in unit test cases, we should not use any external sources like databases, right?
So, as an alternative, we create a mock or fake DB context, called a test double DB context.
So, what's the difference between a mock and a fake?



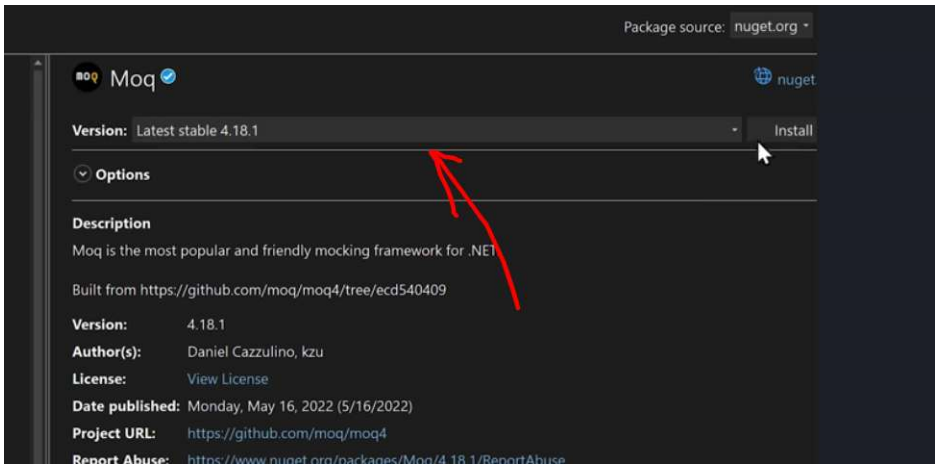
So, what's the difference between mock and fake?
A fake means the full alternative implementation of a particular class or service. For example, there is the regular DB context. Imagine someone has created an alternative fake DB context, which doesn't offer the exact same functionality but implements the same interface. The Entity Framework in-memory provider is an example of a fake DB context because it creates an alternate DB context that does not access the real SQL Server database. However, it contains all the properties and methods of the original DB context.
On the other hand, mocking is something we have to do ourselves. The difference between mocking and faking is that in the case of a mock, you create an alternative implementation for a particular property or method, as opposed to creating all the properties and methods of the class entirely. For example, if a service has 10 methods, and you implement all 10 methods in the fake class, that is called a fake class. But if you provide an alternative implementation for just one property or method, then that is called mocking.
In the case of mocking, you don't create the complete object, but instead create part of it. This offers the subset functionality needed for unit test cases. Especially for individual unit test cases, mocking is the better, preferred, and most commonly used approach.



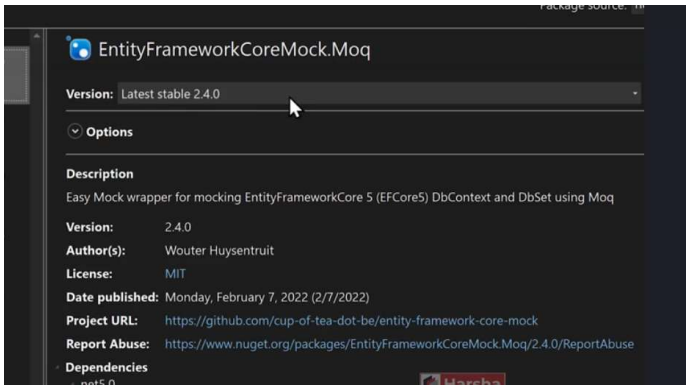


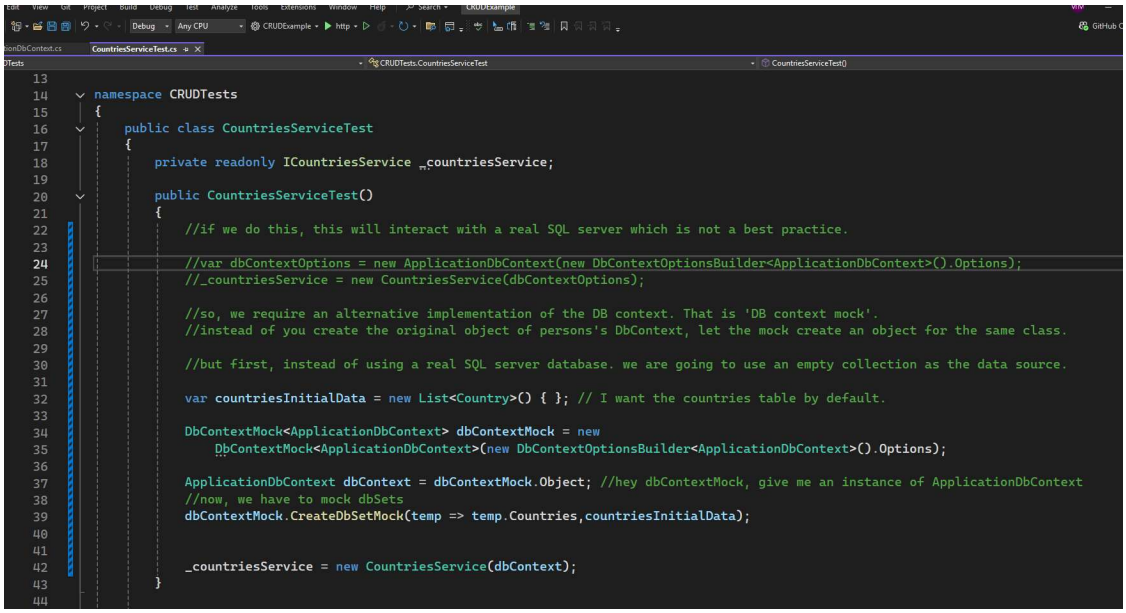


Actually mocking the DBContext yourself is a bit complicated and error prone.



To help us easily mock the DB context, we require another package



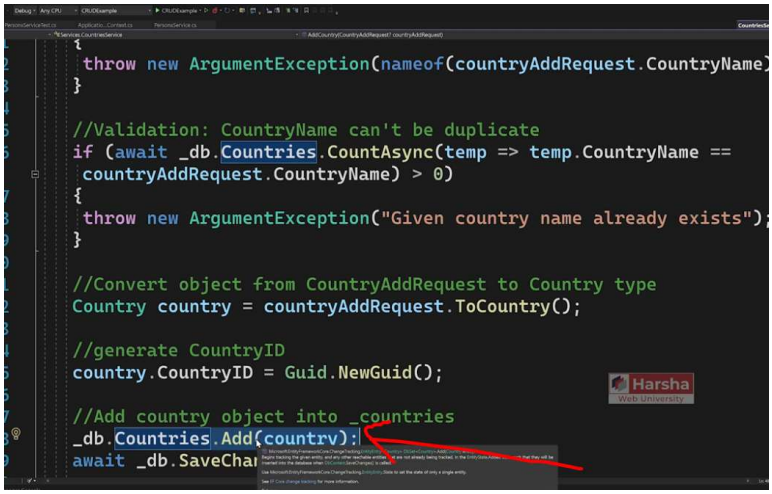


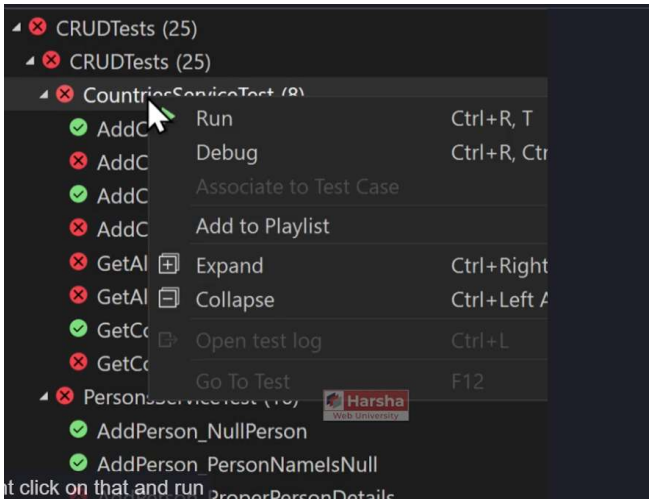
```
namespace CRUDTests
{
    public class CountriesServiceTest
    {
        private readonly ICountriesService _countriesService;

        public CountriesServiceTest()
        {
            //if we do this, this will interact with a real SQL server which is not a best practice.

            //var dbContextOptions = new ApplicationDbContext(new DbContextOptionsBuilder<ApplicationDbContext>().Options);
            //_countriesService = new CountriesService(dbContextOptions);

            //so, we require an alternative implementation of the DB context. That is 'DB context mock'.
            //instead of you create the original object of persons's DbContext, let the mock create an object for the same class.

            //but first, instead of using a real SQL server database. we are going to use an empty collection as the data source.

            var countriesInitialData = new List<Country>() { }; // I want the countries table by default.

            DbContextMock<ApplicationDbContext> dbContextMock = new
                DbContextMock<ApplicationDbContext>(new DbContextOptionsBuilder<ApplicationDbContext>().Options);

            ApplicationDbContext dbContext = dbContextMock.Object; //hey dbContextMock, give me an instance of ApplicationDbContext
            //now, we have to mock dbSets
            dbContextMock.CreateDbSetMock(temp => temp.Countries,countriesInitialData);


            _countriesService = new CountriesService(dbContext);
        }
```

Now, let's go to this test method

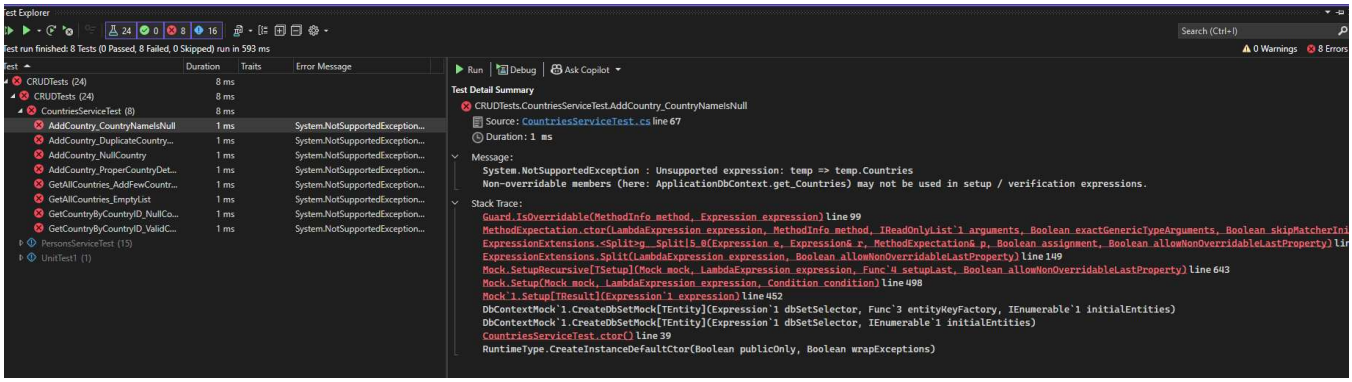
```
    }

    // When you supply proper country name, it should insert (add) the country to the existing list of countries.
    [Fact]
    public async Task AddCountry_ProperCountryDetails()
    {
        //Arrange
        CountryAddRequest? request = new CountryAddRequest()
        {
            CountryName = "Japan"
        };

        //Act
        CountryResponse response = await _countriesService.AddCountry(request);
        List<CountryResponse> countries_from_GetAllCountries = await _countriesService.GetAllCountries();


        //Assert
        Assert.True(response.CountryID != Guid.Empty);
        Assert.Contains(response,countries_from_GetAllCountries);
    }
    #endregion
```

Add() method now does not perform the real database insertion but instead, but instead it calls the mock functionality of the Add() method. Same goes to SaveChangesAsync(0 method.

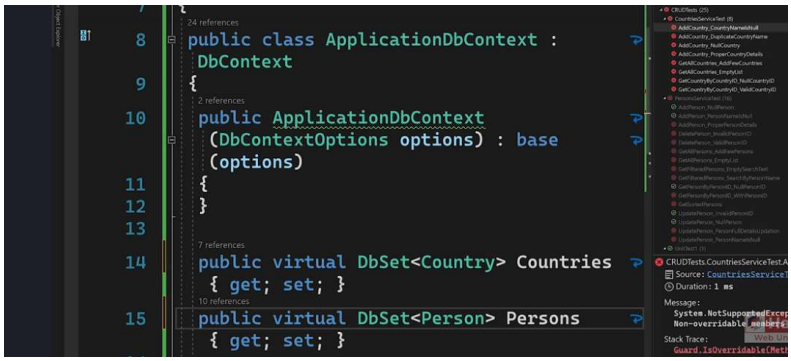


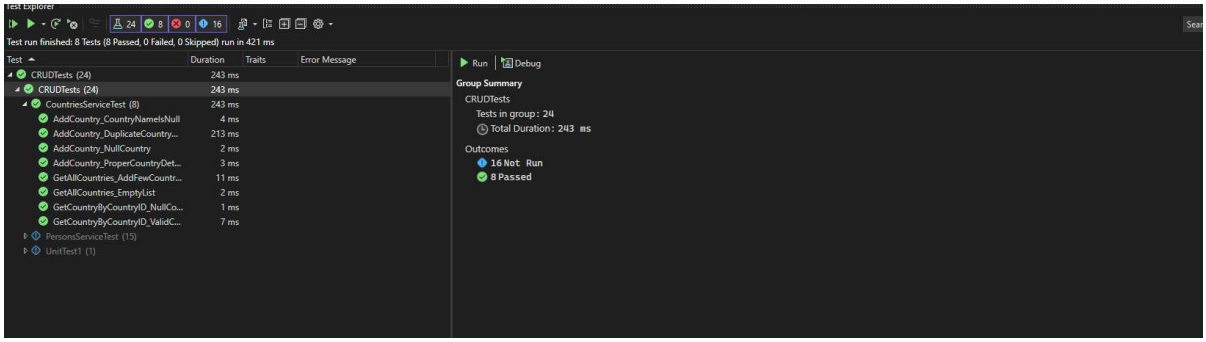


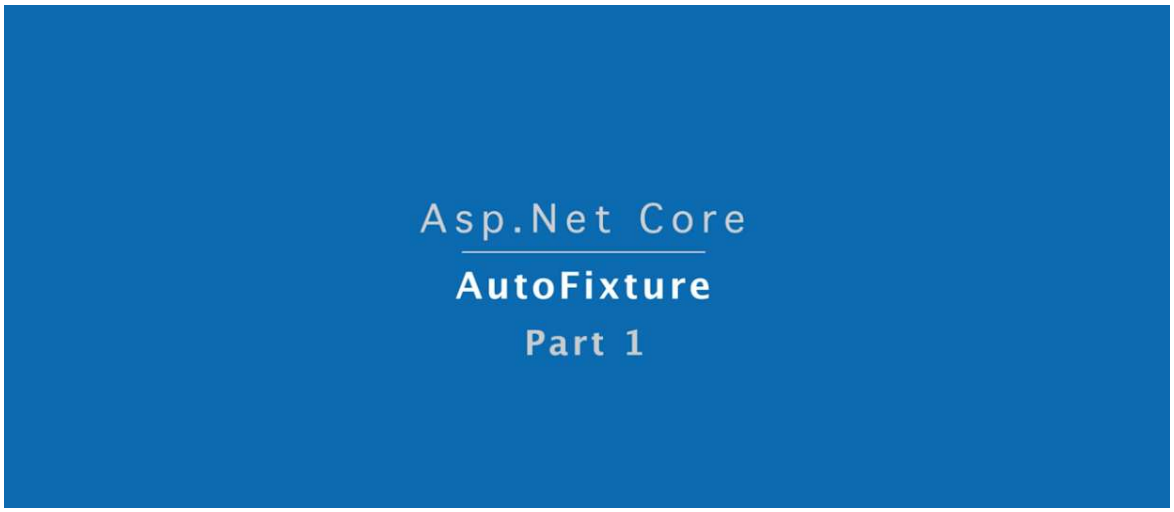
We are getting this error.



It allows child classes/mocking child classes to override these DbSets.

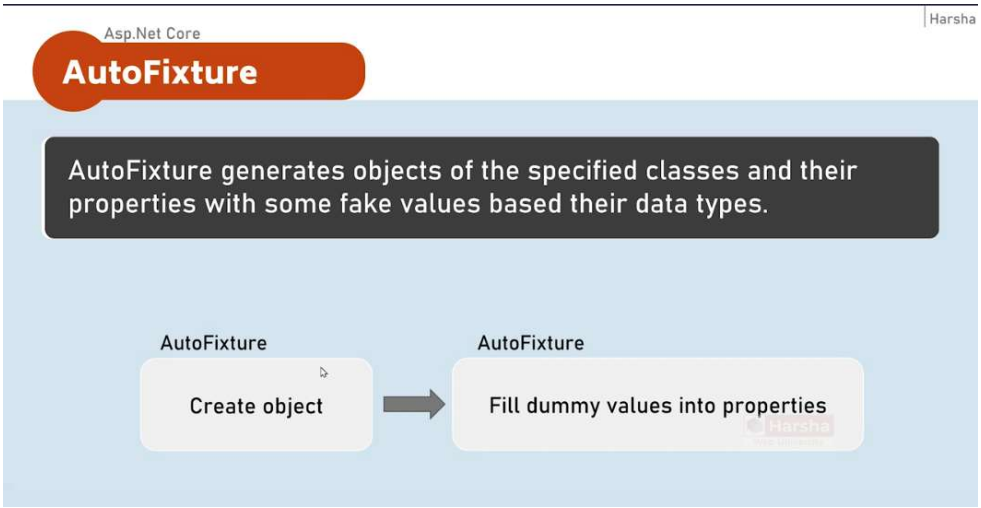






If you notice any unit test code, for example, when we open "add person," in almost every unit test case, we are trying to do one thing as common: we are trying to create a dummy model object with some fake data, like this.
So, in order to minimize the overhead of creating your own model objects, if there is a library that creates the model objects automatically with some fake or dummy data, that would be better, right?
And that is exactly what a fixture does.

```
80
81      //When we supply proper person details, it should insert the
         person into the persons list; and it should return an object
          of PersonResponse, which includes with the newly generated
         person id
82      [Fact]
        0 references
83      public async Task AddPerson_ProperPersonDetails()
84      {
85          //Arrange
86          PersonAddRequest? personAddRequest = new PersonAddRequest()
             { PersonName = "Person name...", Email =
            "person@example.com", Address = "sample address", CountryID
             = Guid.NewGuid(), Gender = GenderOptions.Male, DateOfBirth
             = DateTime.Parse("2000-01-01"), ReceiveNewsLetters =
            true };
87
88          //Act
```



AutoFixture is one of the third-party libraries or packages that makes it easy to create model objects and fill them with fake data.
What exactly it does is that it automatically creates a new model object and initializes it with some default or dummy values.
So instead of creating the model object manually like this, you will be using AutoFixture.

Asp.Net Core

AutoFixture

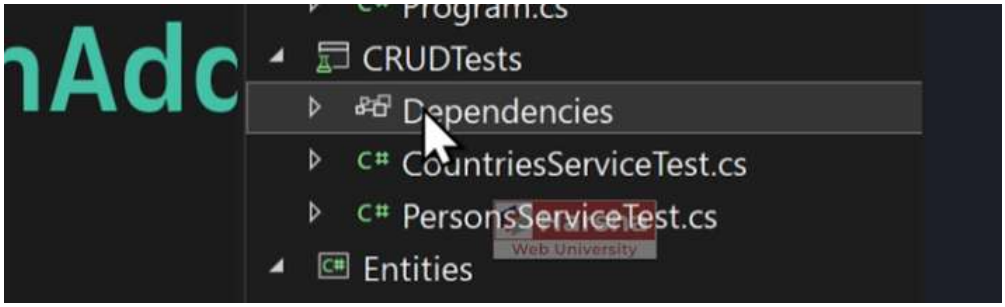
AutoFixture generates objects of the specified classes and their properties with some fake values based their data types.

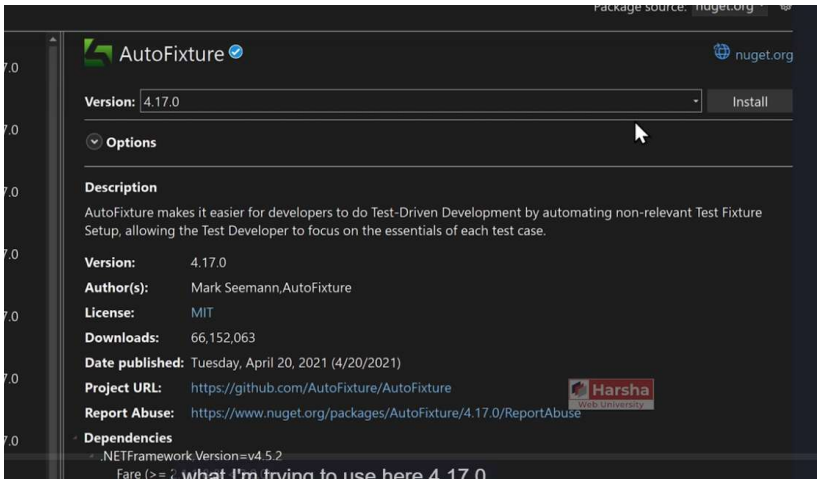
Normal object creation

```
new ModelClass() {
    Property1 = value,
    Property2 = value }
```

With AutoFixture

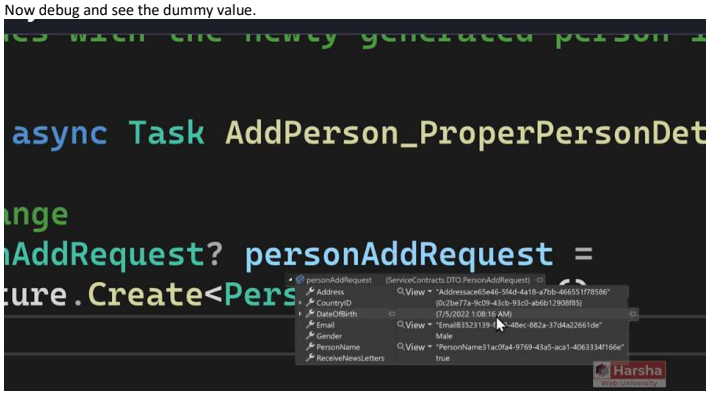
```
Fixture.Create<ModelClass>();
//initializes all properties of the specified
model class with dummy values
```

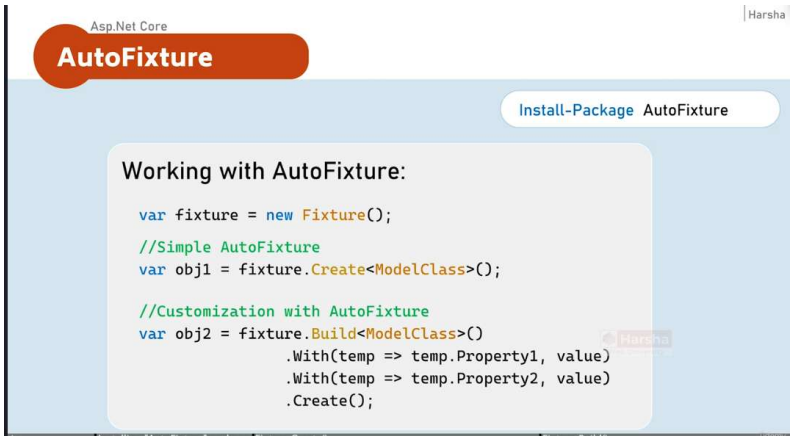




```
//which includes with the newly genarated person id
[Fact]
public async Task AddPerson_ProperPersonDetails()
{
    //Arrange
    //PersonAddRequest? personAddRequest = new PersonAddRequest()
    //{
    //    PersonName = "Person Name...",
    //    Email = "person@example.com",
    //    Address = "sample address",
    //    CountryID = Guid.NewGuid(),
    //    Gender = GenderOptions.Male,
    //    DateOfBirth = DateTime.Parse("2000-01-01"),
    //    ReceiveNewsLetters = true,
    //};

    PersonAddRequest? personAddRequest = _fixture.Create<PersonAddRequest>();

    //Act
```

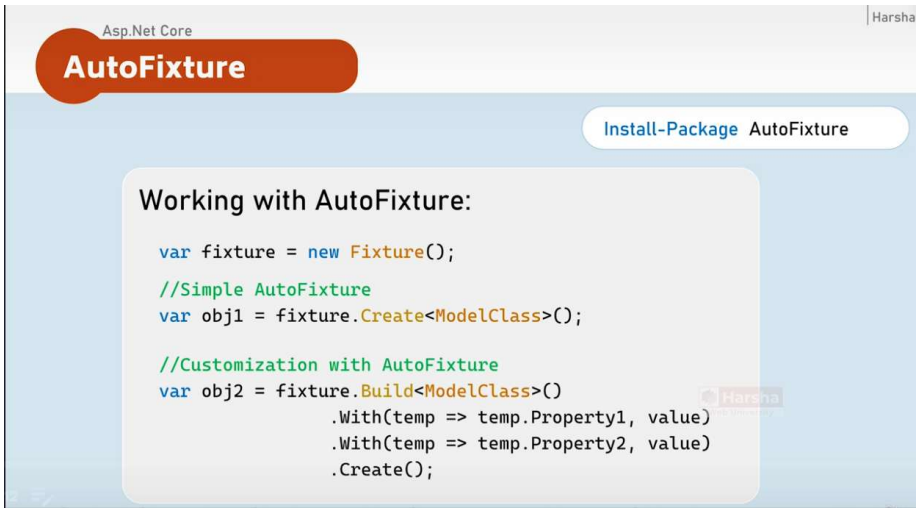
Now debug and see the dummy value.

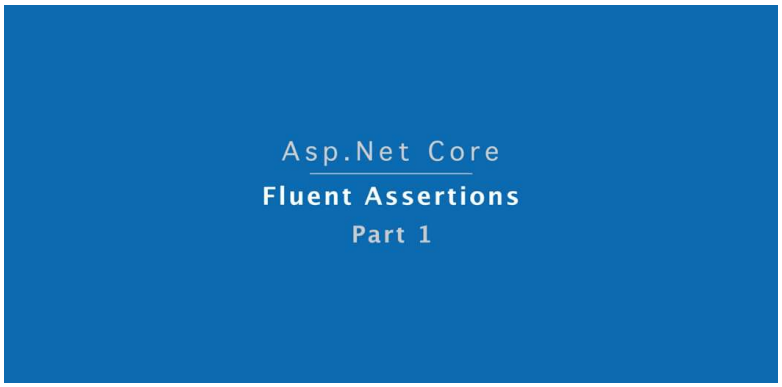


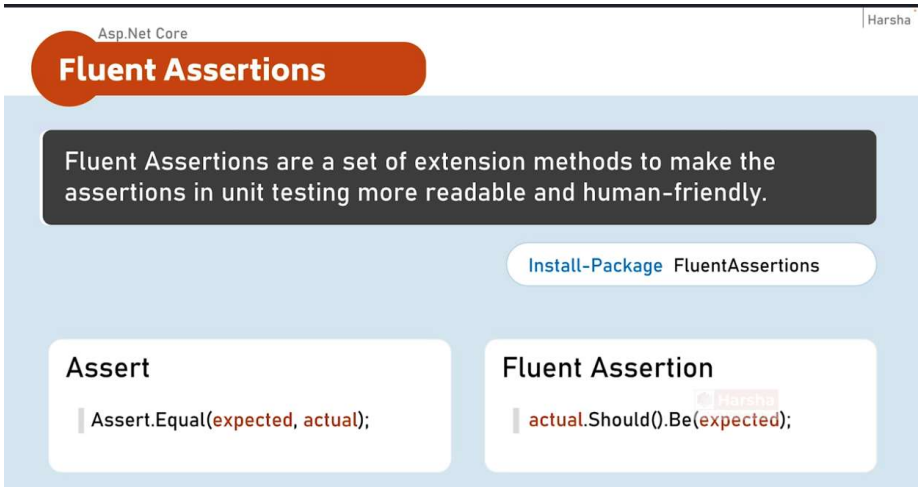


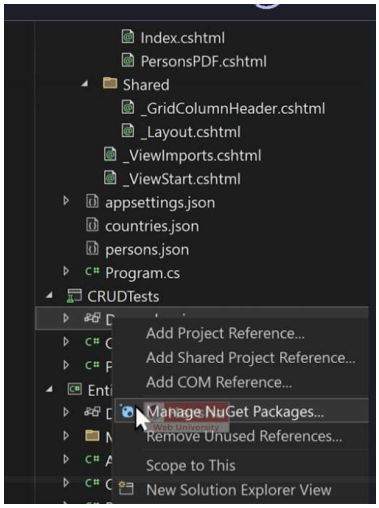
In AutoFixture, we have two methods: create and build.

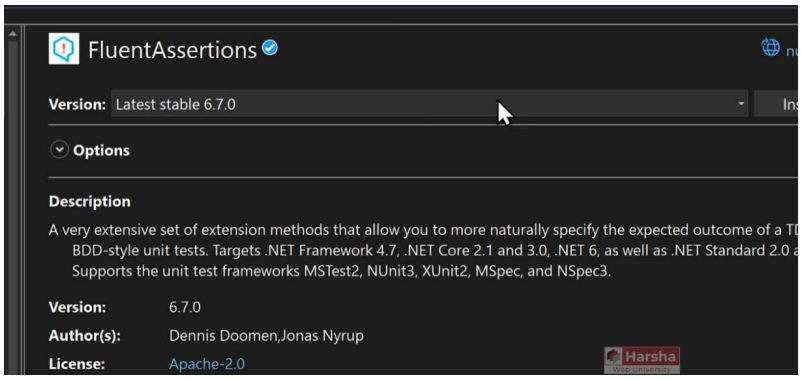
If you want to initialize all properties with default values based on their respective data types, you can use create.
However, if you want to customize some property values, you can use build, followed by create.
In the last lecture, we used AutoFixture in only one test case.
In this lecture, let's try to apply AutoFixture to all the remaining unit test cases.

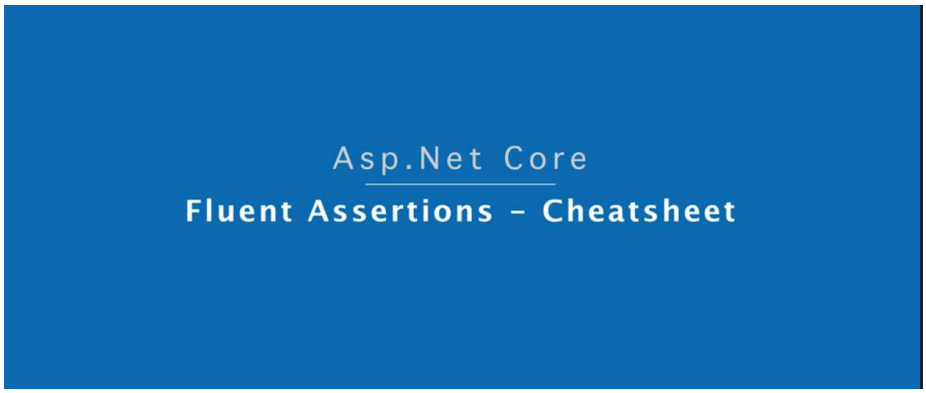












Now, let us see the predefined methods of the fluid assertion.
We are trying to compare the general asset and flow into assertion so that you can easily understand when to use which method.

Harsha

Asp.Net Core

Fluent Assertions

[Part 1]

Assert:

```
//Equal
Assert.Equal(expected, actual);

//Not Equal
Assert.NotEqual(expected, actual);

//Null
Assert.Null(actual);

//Not Null
Assert.NotNull(actual);
```

Fluent Assertion:

```
//Equal
actual.Should().Be(expected);

//Not Equal
actual.Should().NotBe(expected);

//Null
actual.Should().BeNull();

//Not Null
actual.Should().NotBeNull();
```

Harsha

Asp.Net Core

Fluent Assertions

[Part 2]

Assert:

```
//True
Assert.True(actual);

//False
Assert.False(actual);

//Empty
Assert.Empty(actual);

//Not Empty
Assert.NotEmpty(actual);
```

Fluent Assertion:

```
//True
actual.Should().BeTrue();

//False
actual.Should().BeFalse();

//Empty
actual.Should().BeEmpty();

//Not Empty
actual.Should().NotBeEmpty();
```

Harsha

Asp.Net Core

Fluent Assertions

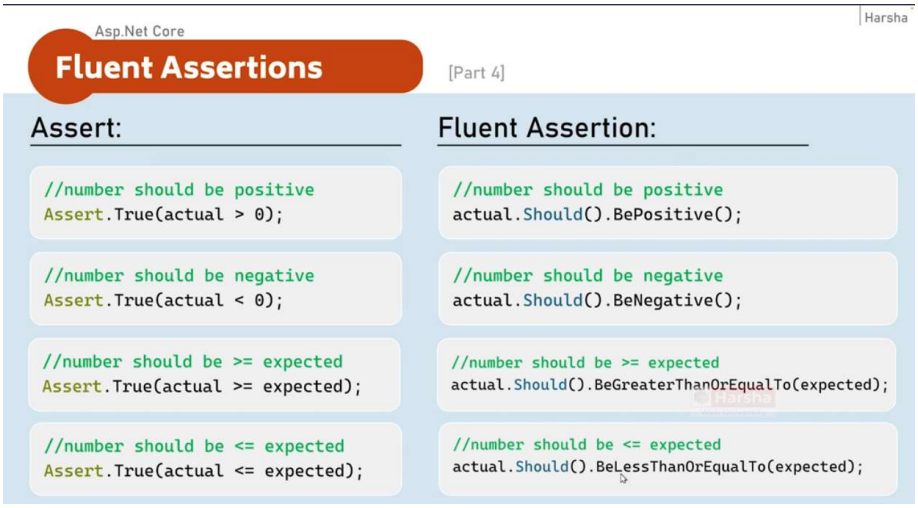
[Part 3]

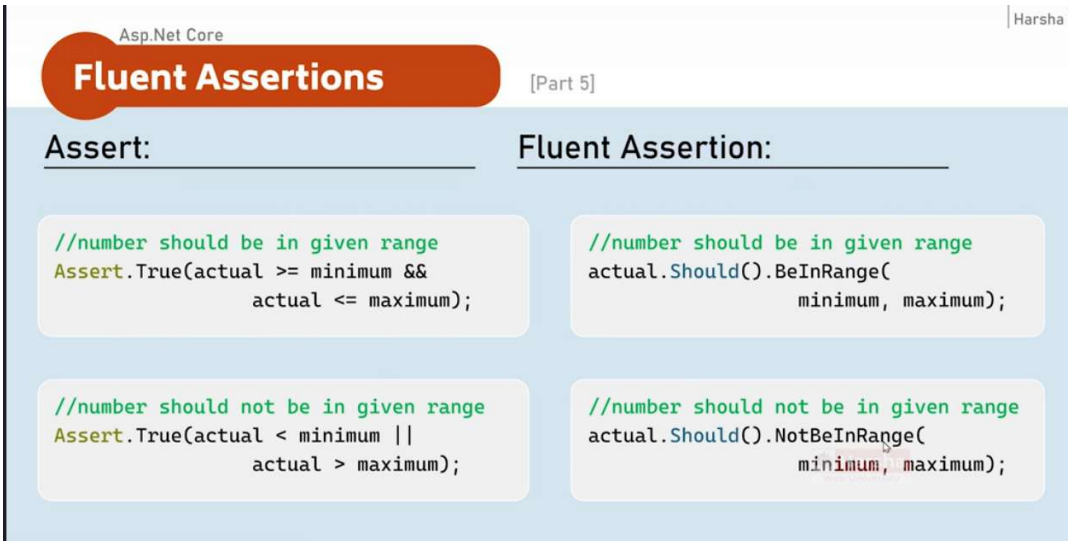
Assert:

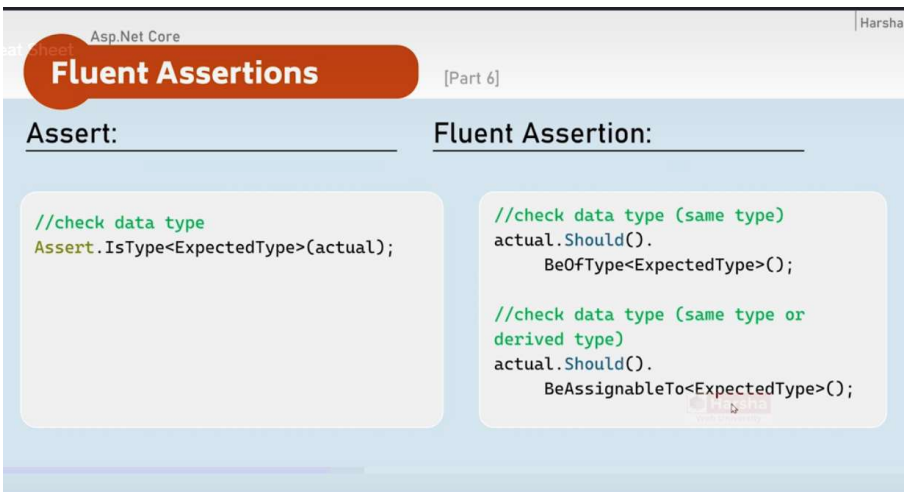
```
//Null or empty
Assert.True(string.IsNullOrEmpty(
                actual)); //string

Assert.True(actual == null ||
    actual.Length == 0); //collection

//Should not be null or empty
Assert.False
(string.IsNullOrEmpty(actual)); //string

Assert.False(actual == null ||
    actual.Length == 0); //collection
```

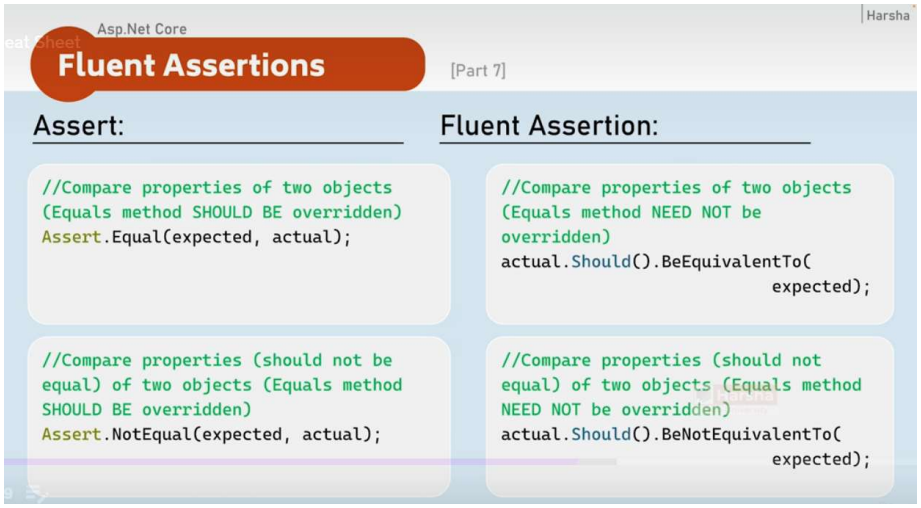
Fluent Assertion:

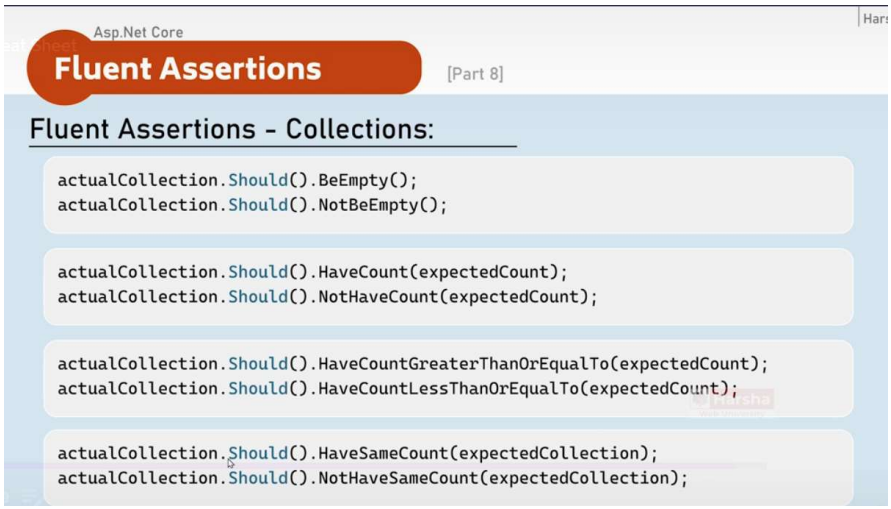
```
//Null or empty
actual.Should().BeNullOrEmpty();

//Should not be null or empty
actual.Should().NotBeNullOrEmpty();
```


Harsha
Asp.Net Core
Fluent Assertions
[Part 4]
Assert:
Fluent Assertion:
//number should be positive
Assert.True(actual > 0);
//number should be positive
actual.Should().BePositive();
//number should be negative
Assert.True(actual < 0);
//number should be negative
actual.Should().BeNegative();
//number should be >= expected
Assert.True(actual >= expected);
//number should be >= expected
actual.Should().BeGreaterThanOrEqualTo(expected);
//number should be <= expected
Assert.True(actual <= expected);
//number should be <= expected
actual.Should().BeLessThanOrEqualTo(expected);

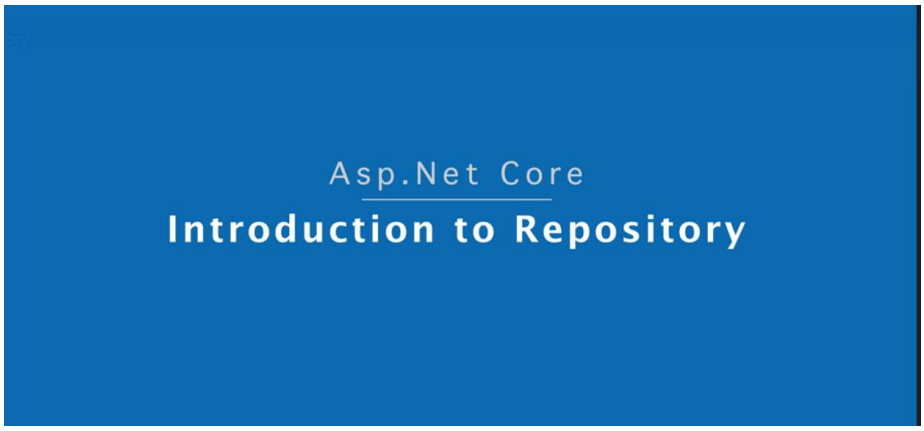

Harsha
Asp.Net Core
Fluent Assertions
[Part 5]
Assert:
Fluent Assertion:
//number should be in given range
Assert.True(actual >= minimum &&
actual <= maximum);
//number should be in given range
actual.Should().BeInRange(
minimum, maximum);
//number should not be in given range
Assert.True(actual < minimum ||
actual > maximum);
//number should not be in given range
actual.Should().NotBeInRange(
minimum, maximum);

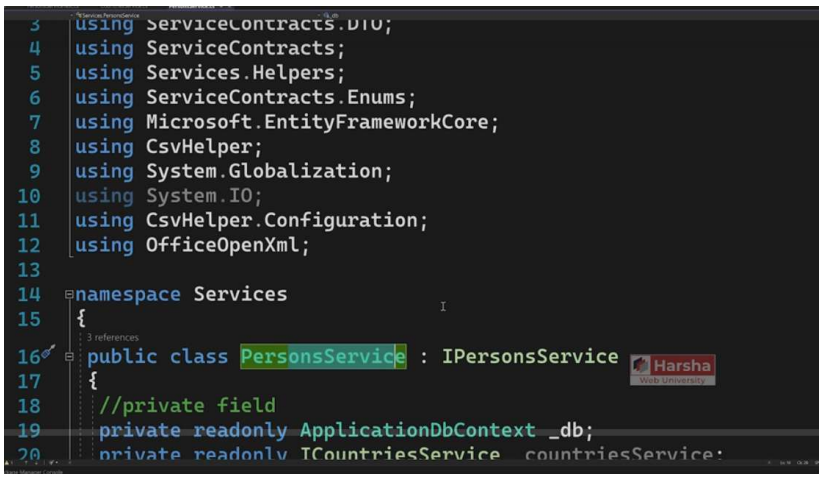

Harsha
Asp.Net Core
Fluent Assertions
[Part 6]
Assert:
Fluent Assertion:
//check data type
Assert.IsType<ExpectedType>(actual);
//check data type (same type)
actual.Should().
BeOfType<ExpectedType>();
//check data type (same type or
derived type)
actual.Should().
BeAssignableTo<ExpectedType>();


Harsha
Asp.Net Core
Fluent Assertions
[Part 7]
Assert:
Fluent Assertion:
//Compare properties of two objects (Equals method SHOULD BE overridden)
Assert.Equal(expected, actual);
//Compare properties of two objects (Equals method NEED NOT be overridden)
actual.Should().BeEquivalentTo(expected);
//Compare properties (should not be equal) of two objects (Equals method SHOULD BE overridden)
Assert.NotEqual(expected, actual);
//Compare properties (should not equal) of two objects (Equals method NEED NOT be overridden)
actual.Should().BeNotEquivalentTo(expected);


Asp.Net Core
Fluent Assertions
[Part 8]
Fluent Assertions - Collections:
actualCollection.Should().BeEmpty();
actualCollection.Should().NotBeEmpty();
actualCollection.Should().HaveCount(expectedCount);
actualCollection.Should().NotHaveCount(expectedCount);
actualCollection.Should().HaveCountGreaterThanOrEqualTo(expectedCount);
actualCollection.Should().HaveCountLessThanOrEqualTo(expectedCount);
actualCollection.Should().HaveSameCount(expectedCollection);
actualCollection.Should().NotHaveSameCount(expectedCollection);


Harsha
Asp.Net Core
Fluent Assertions
[Part 9]
Fluent Assertions - Collections:
actualCollection.Should().BeEquivalentTo(expectedCollection);
actualCollection.Should().NotBeEquivalentTo(expectedCollection);
actualCollection.Should().ContainInOrder(expectedCollection);
actualCollection.Should().NotContainInOrder(expectedCollection);
actualCollection.Should().OnlyHaveUniqueItems(expectedCount);
actualCollection.Should().OnlyContain(temp => condition);
actualCollection.Should().BeInAscendingOrder(temp => temp.Property);
actualCollection.Should().BeInDescendingOrder(temp => temp.Property);





There is a design flaw in this application.
The business logic, which is the service logic, is tightly coupled with the data store.
This means that as part of the service logic, we are writing the data access logic, directly accessing the DB context.

For example, consider the AddPerson method in the PersonService. After essential validations, we are writing code to access the data store, meaning we are directly using the DB context.

The concept of DB context is specific to Entity Framework. If, in the future, your company or team decides to migrate to another data store—such as an in-memory collection, Dapper, Azure, PostgreSQL, or any other database—you would need to modify the service logic.

Every time you make changes in the service, it leads to regression testing, requiring repeated checks of the same logic, even for small changes.

This tight coupling of business logic with data access logic is not a good practice for larger applications. While it may be acceptable for small-scale applications where migration is not a concern, real-world applications generally require a clear separation between business logic and data access logic.

This separation can be achieved using the **Repository** pattern.

```
    throw new ArgumentNullException(nameof(personAddRequest));
}

//Model validation
ValidationHelper.ModelValidation(personAddRequest);

//convert personAddRequest into Person type
Person person = personAddRequest.ToPerson();

//generate PersonID
person.PersonID = Guid.NewGuid();

//add person object to persons list
_db.Persons.Add(person);
await _db.SaveChangesAsync();
//_db.sp_InsertPerson(person);
```

I'm not saying that everyone should use the repository pattern, but in my experience, the majority of companies prefer it.

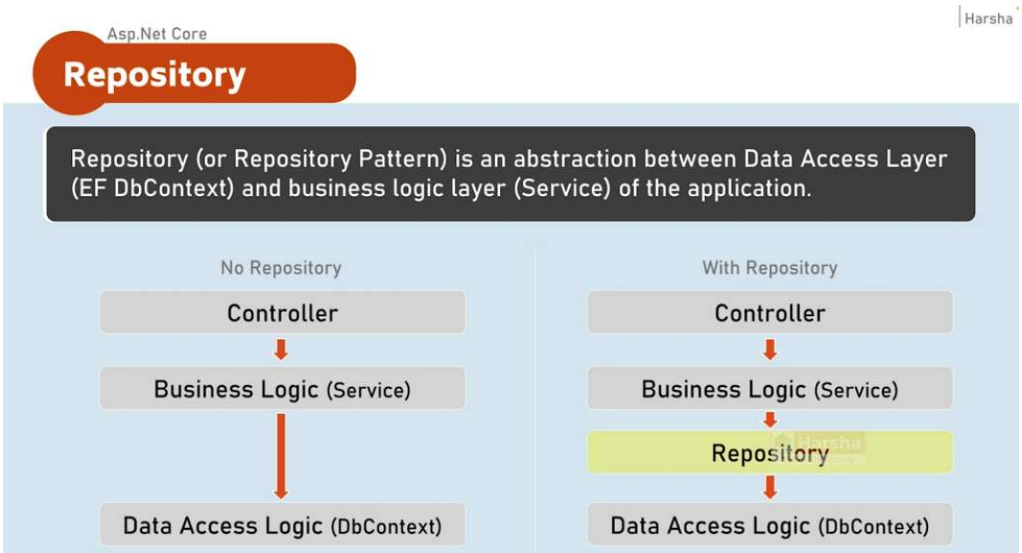
What is a Repository?

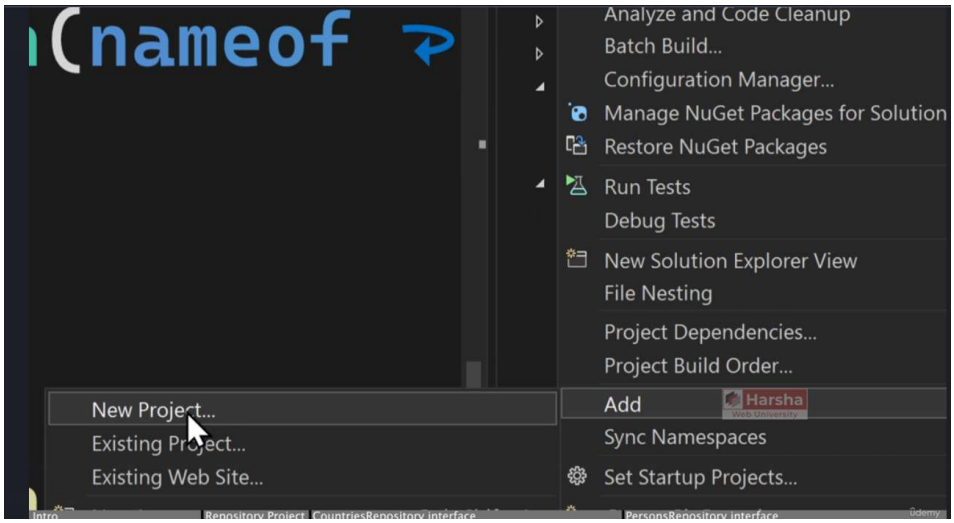
A repository is an abstraction layer between business logic and data access logic.
This means the repository is tightly coupled with the data access logic—it accesses the database using the **DB context**—but the **business logic is not**.
The business logic remains completely separate from data access logic and depends only on the method signatures of the repository interface.

For example, consider a method called GetCountries.

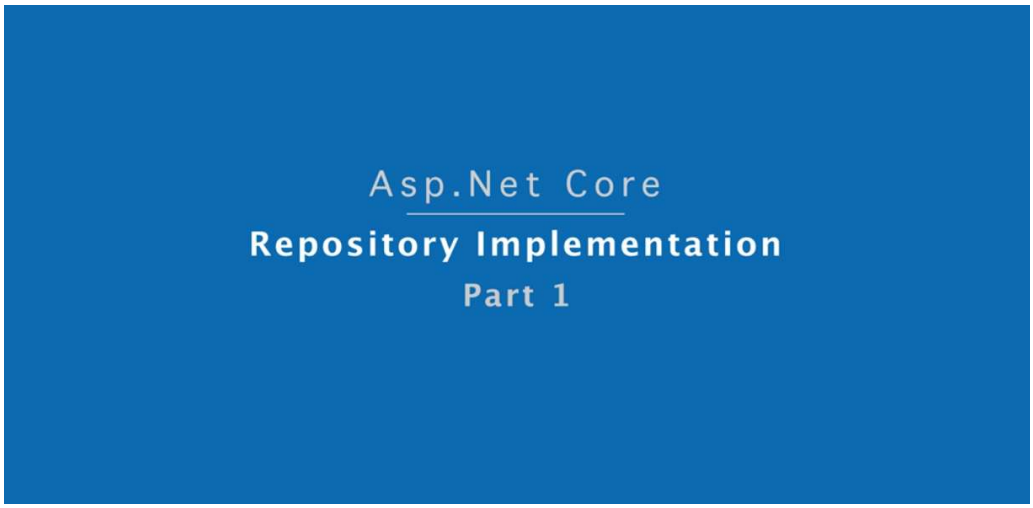
- The service simply calls the repository without worrying about how it is implemented or what the actual data store is.
- The actual data source logic is handled within the repository itself.

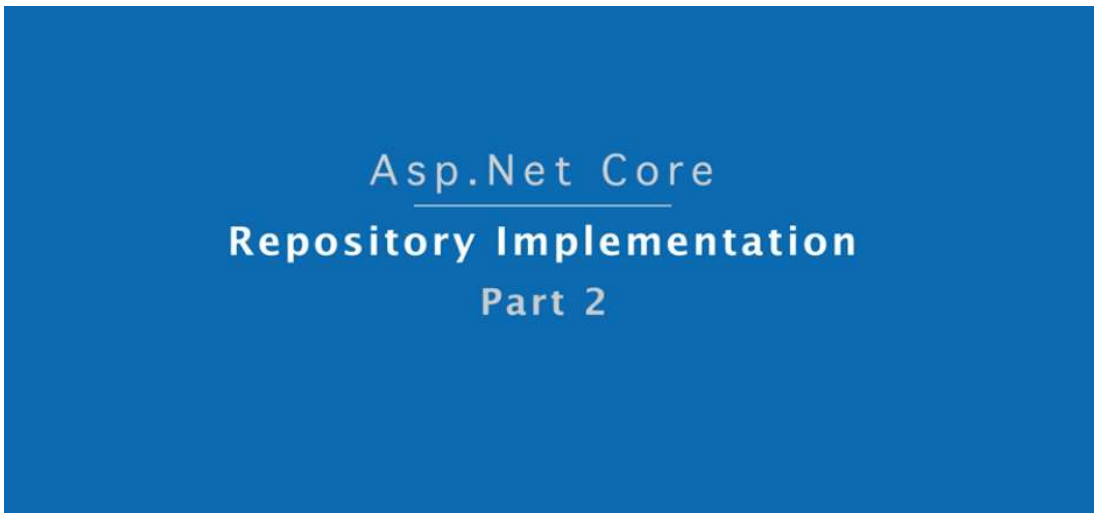
If, in the future, you need to change the data store—say, migrating to **Azure**—you can create an alternative repository without modifying existing service logic.
This leads to **horizontal scaling** instead of modifying existing code, which is one of the biggest advantages of the repository pattern.





Select class library

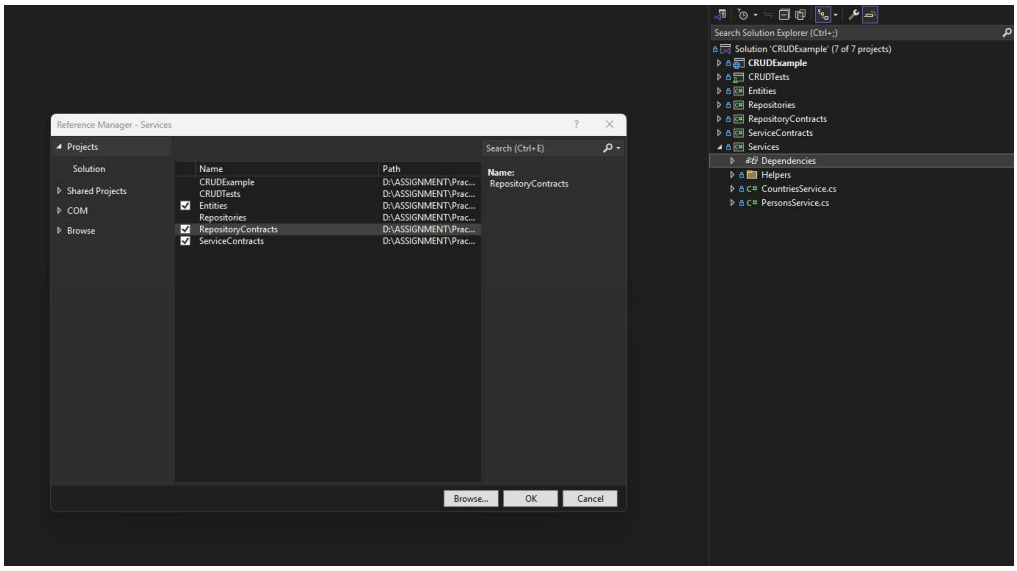


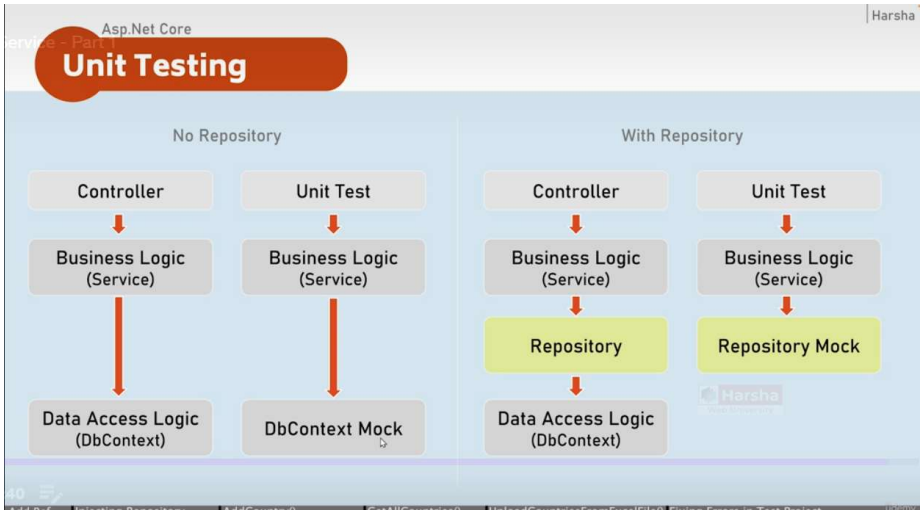


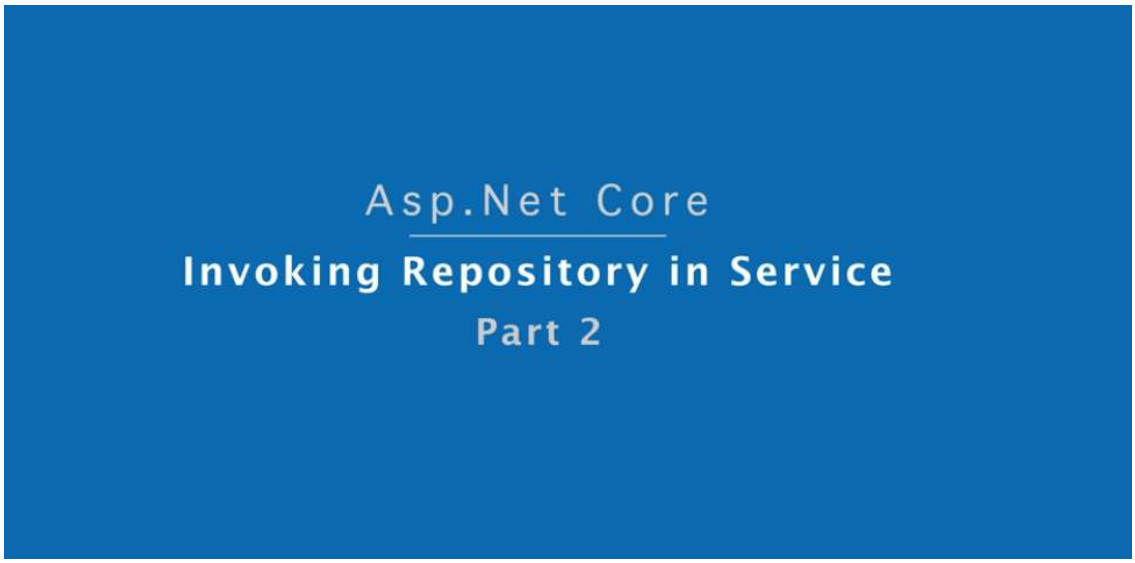
Asp.Net Core

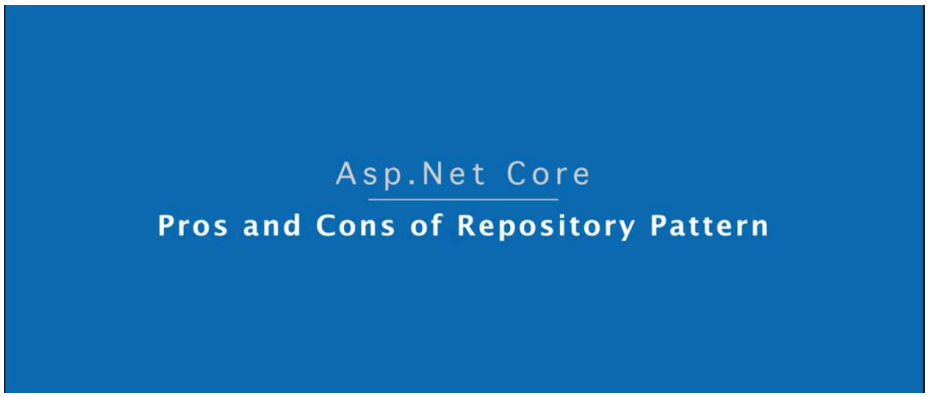
Invoking Repository in Service

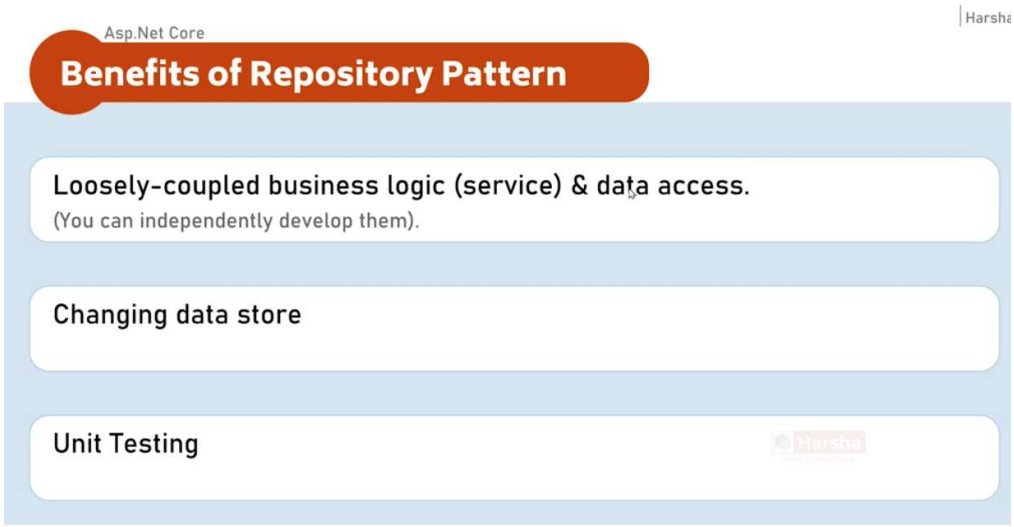
Part 1











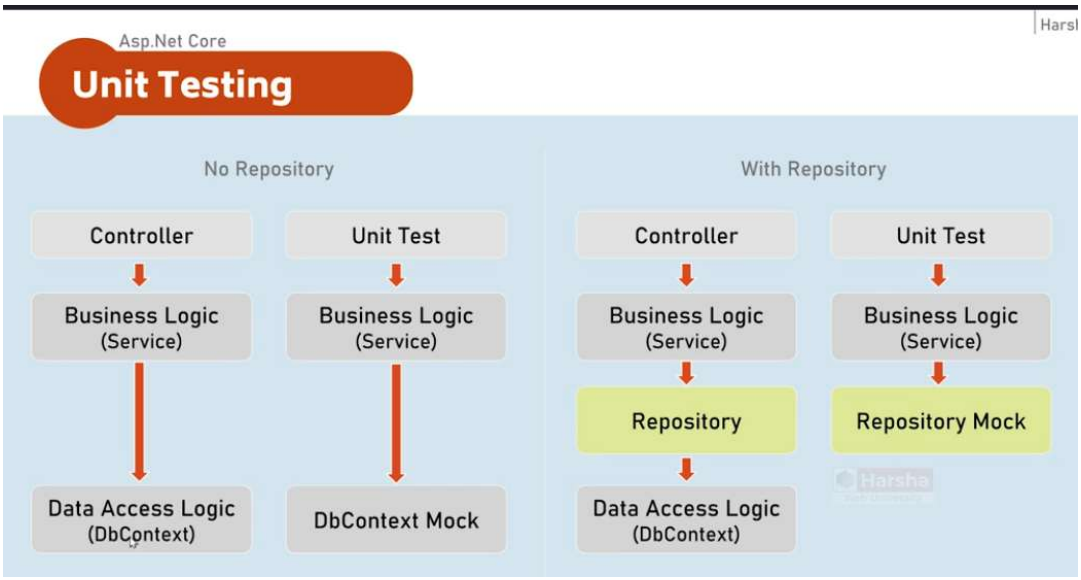
If you can access the **DB context** and interact with the database directly within the **service layer**, why use the **repository pattern**?
Here are three key benefits:

1. **Loosely Coupled Business Logic and Data Access**
   - The repository pattern separates business logic from data access logic.
   - The service layer depends only on the **repository interface**, not the actual database implementation.
2. **Easier Maintainability and Scalability**
   - If you need to switch to a different database (e.g., moving from **Entity Framework** to **Dapper** or **Azure Cosmos DB**), you only need to modify the repository layer, **not the service layer**.
   - This avoids unnecessary modifications and reduces **regression testing**.

3 Better Unit Testing Support

- Since the service depends on an **interface**, you can easily **mock** the repository during unit testing.
- This allows you to test business logic independently, without needing a real database connection.

Even though direct DB context usage works for **small applications**, for **larger and scalable** systems, the **repository pattern** helps keep things modular and easier to manage.



In traditional code, we often access the **DB context** directly within the **business logic**, creating **tight coupling** between business and data access logic.

Why is this a problem?

- If we need to modify the **data access logic**, we also have to update the **service layer**.
- This leads to **regression testing**, meaning we must **retest the entire business logic**, even for small changes in the data layer.

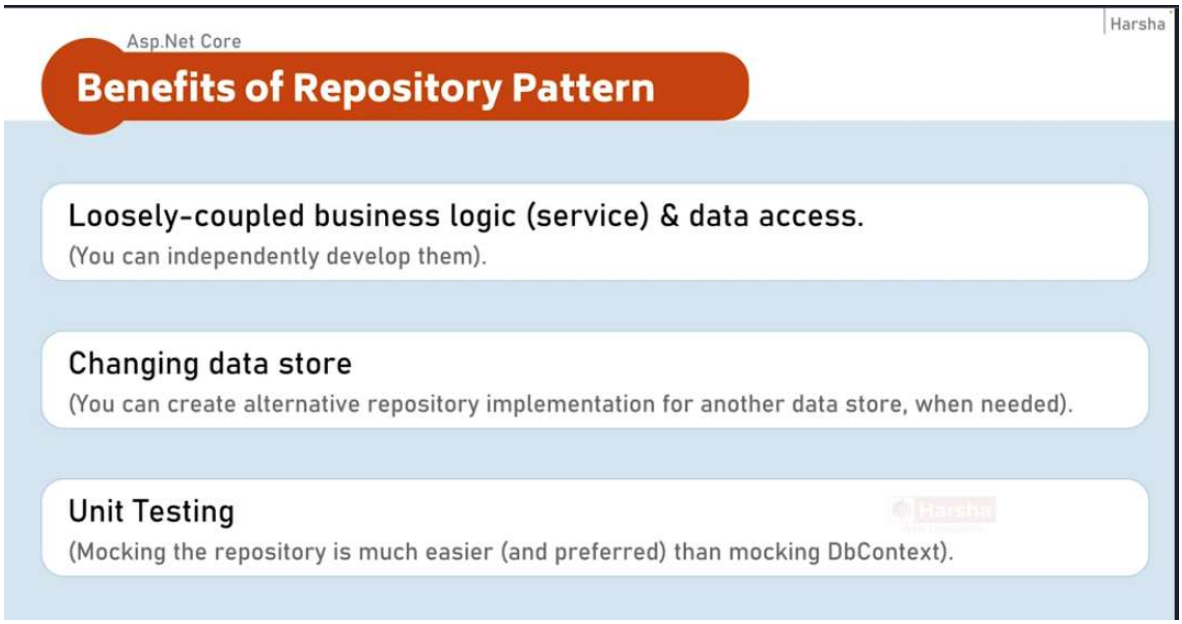
How does the repository pattern help?

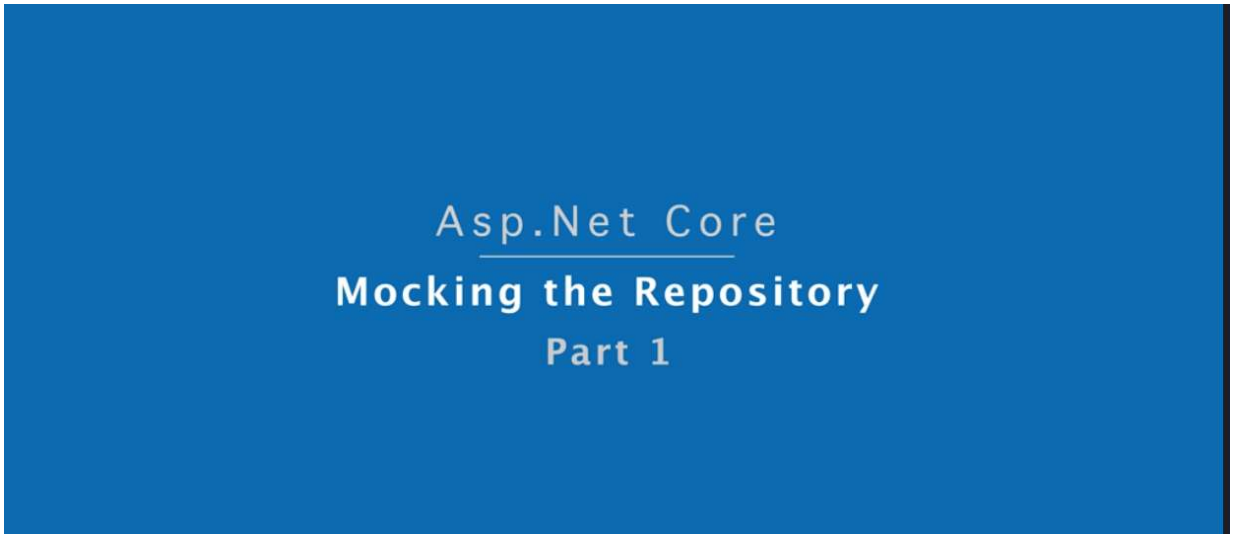
- With a **repository**, changes in the **data access logic** (e.g., modifying a LINQ query) only require **testing the repository layer**.
- There's **no need to retest the business logic** or upper layers, making maintenance **easier and safer**.

What about database migration?

- If you decide to switch from **SQL Server to PostgreSQL**, or move to **Azure** or another cloud provider, you don't need to modify your **service layer**.
- Instead, you just create a **new repository implementation** that follows the **same repository interface**, keeping the rest of your application untouched.

This separation makes your application **more flexible, testable, and scalable**. 🚀



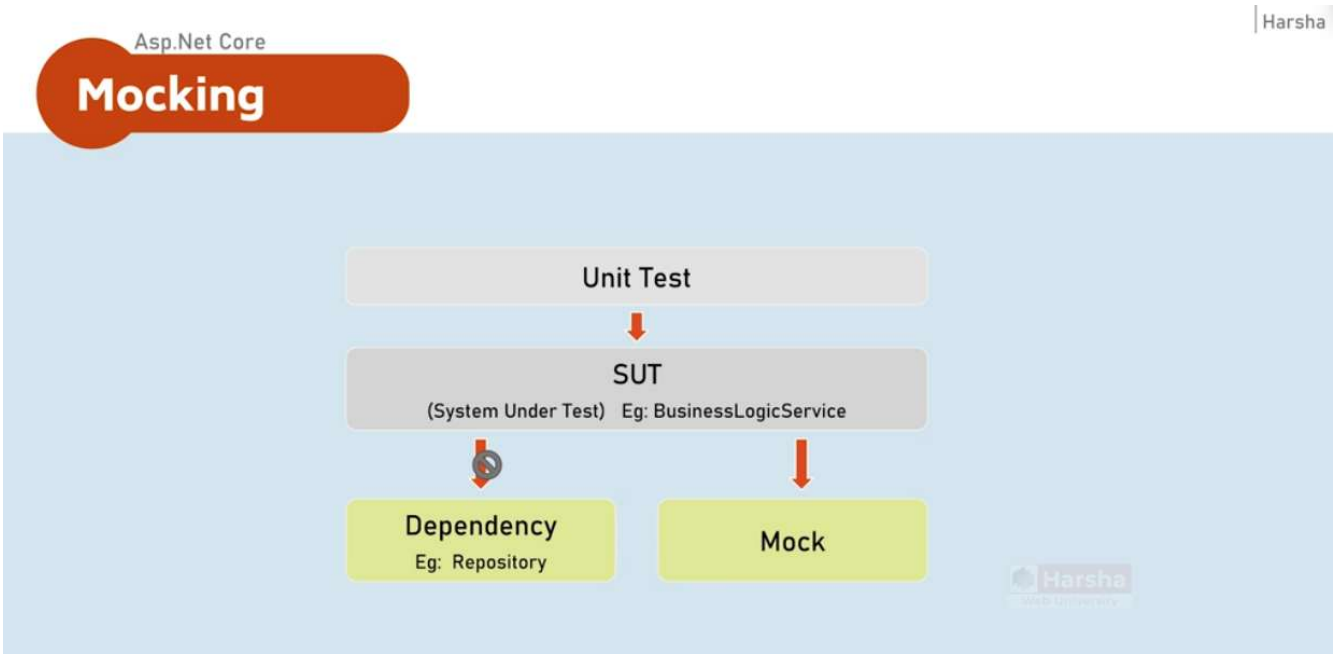


You will **truly experience the power of mocking** in this lecture! 🚀
Why?
In the **previous lecture**, when we mocked the **DB context**, the actual mocking of its properties and methods was handled by a **predefined package or library**.
But to be honest, **that's not a real example of mocking**.

What does real mocking mean?

- **Mocking** actually means **defining a return value for a method without executing** or **re-implementing** it.
- This allows us to **simulate dependencies** without relying on a real database, making our tests **faster and more reliable**.

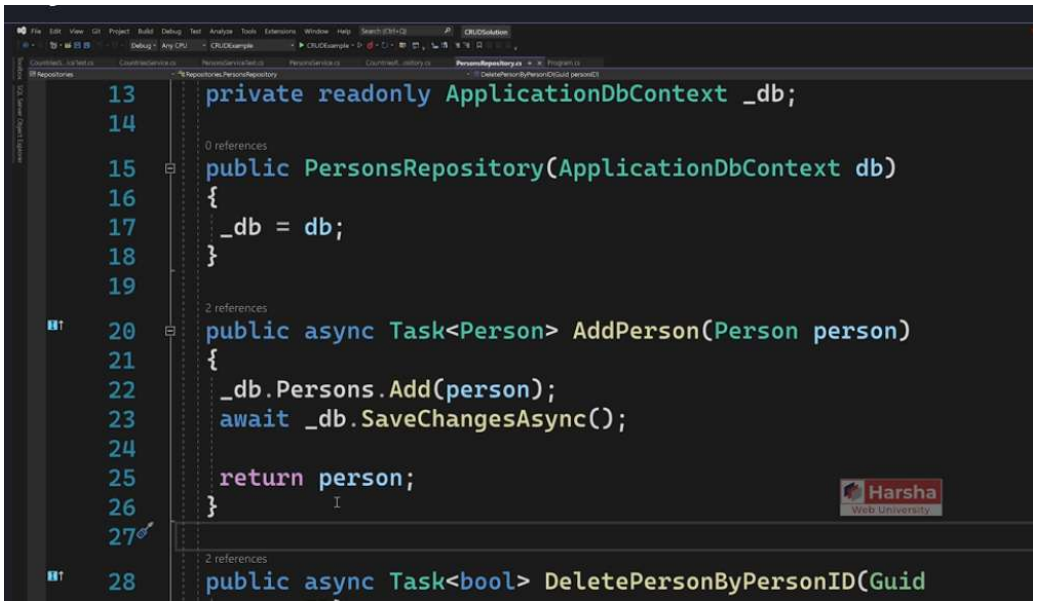
So, in this lecture, you'll see **real mocking in action**! 🔥



Example of Real Mocking
Let's say we have a method called **AddPerson** in the **PersonRepository**.
With **mocking**, we don't actually execute this method. Instead, we:
☑ **Define a return value** without running the real implementation.
☑ **Return a dummy value** when the method is called.
How does this work?

- When someone calls **AddPerson**, instead of executing its actual logic, the mock returns a **predefined response**.
- This helps in **isolating tests**, ensuring we're not depending on real database operations.

👉 **That's exactly what mocking is!** 🎭
And in this lecture, we're going to implement it in action! 🚀



Of course! Before we start mocking, we need to make sure we have installed the **Moq** package in the **CRUD Tests** project.

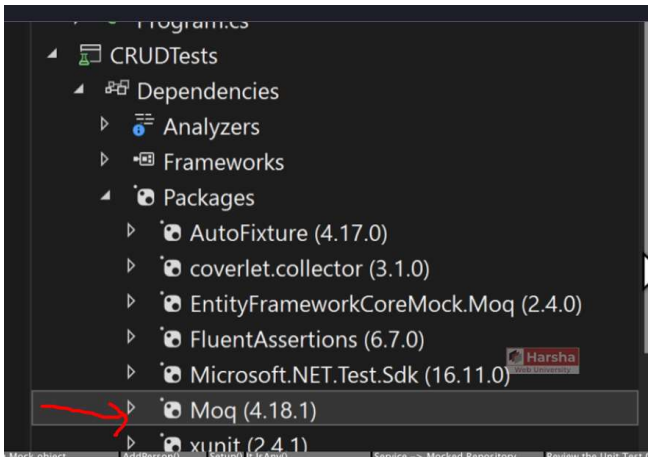
How to Check?

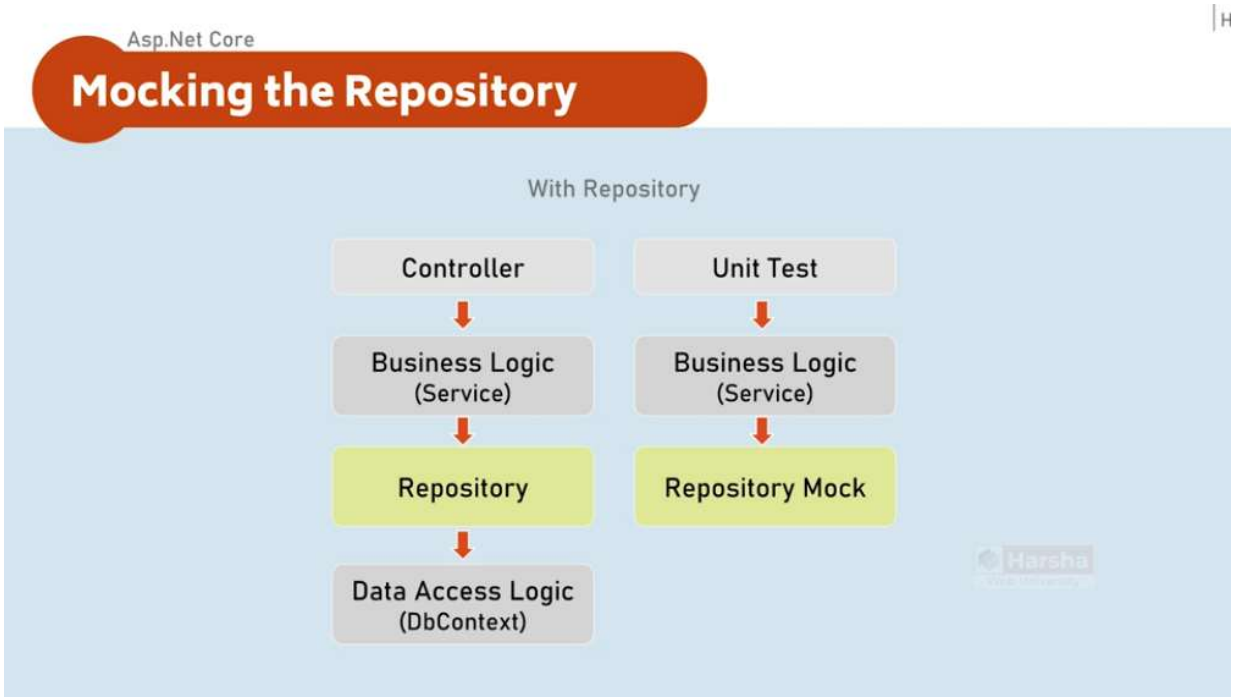
- In your **test project**, go to **Dependencies > Packages**.
- You should see a package called **Moq** listed there.

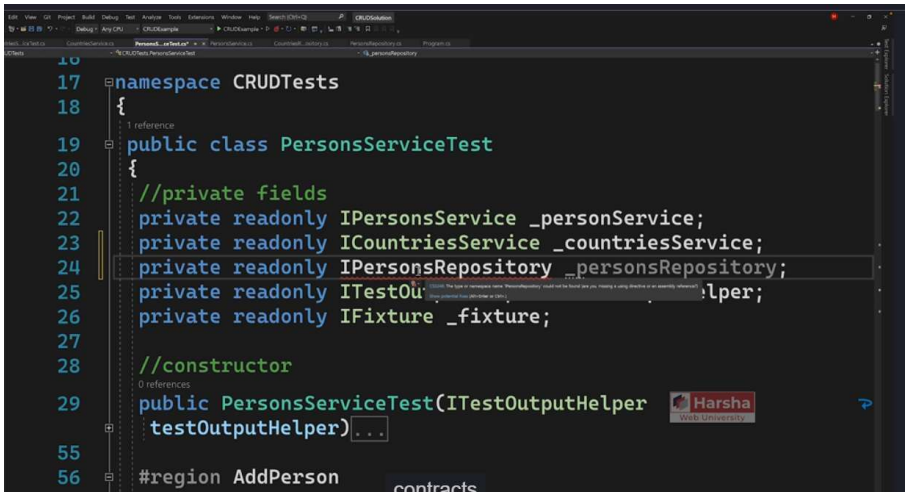
Why is this important?

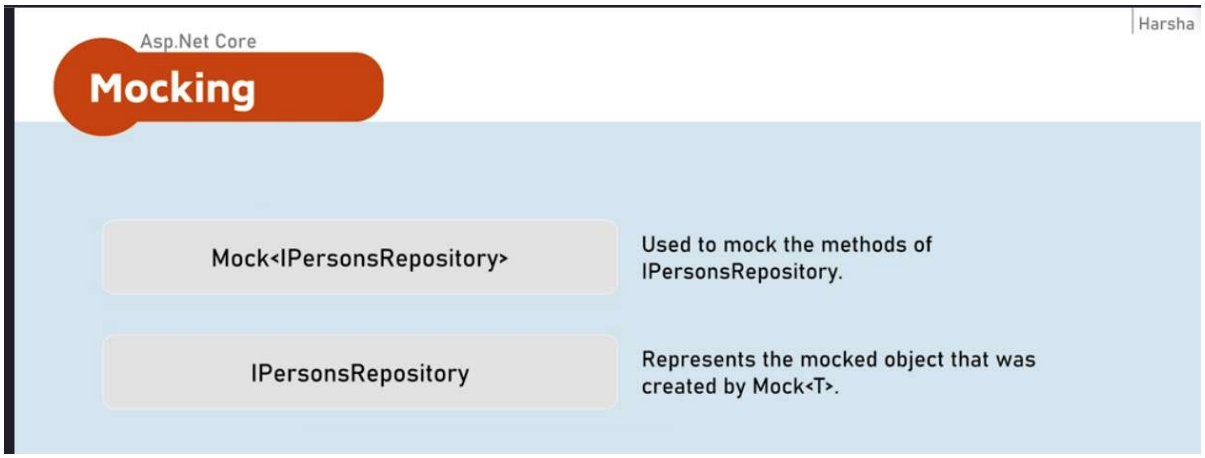
- **Moq** is the actual package that allows us to create **mock objects** for testing.
- Without it, we won't be able to mock dependencies like the **repository or DB context**.

Once it's installed, we're all set to implement **real mocking** in our tests! 🚀









Unit Testing Controllers in MVC Architecture

In **Model-View-Controller (MVC)** architecture, controllers are designed to be **loosely coupled** from the **view, service, or any other layers**.

☞ **Why?**

- This makes controllers **independently testable** without needing to rely on actual services or repositories.

Mocking in Different Layers

✓ **When testing services → We mock the repository**

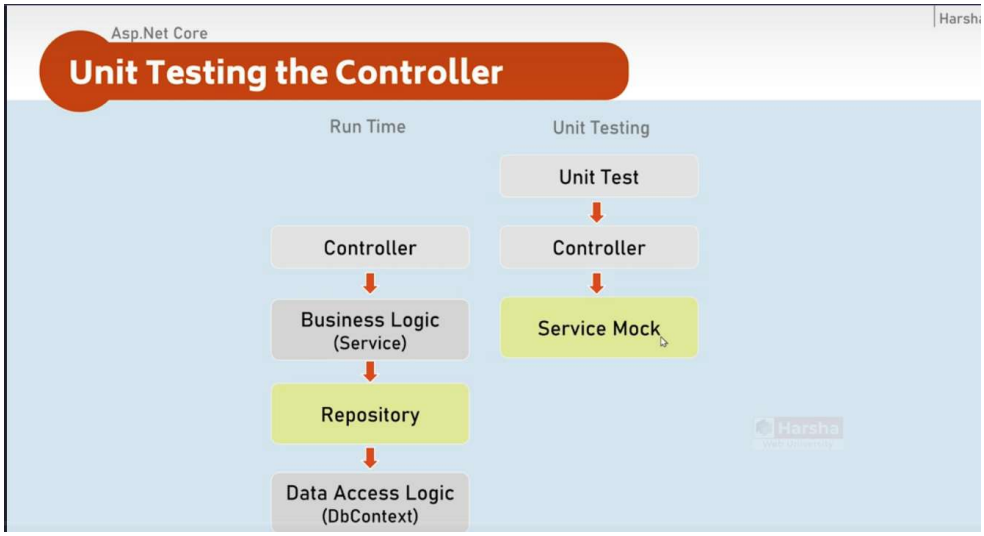
✓ **When testing controllers → We mock the service**

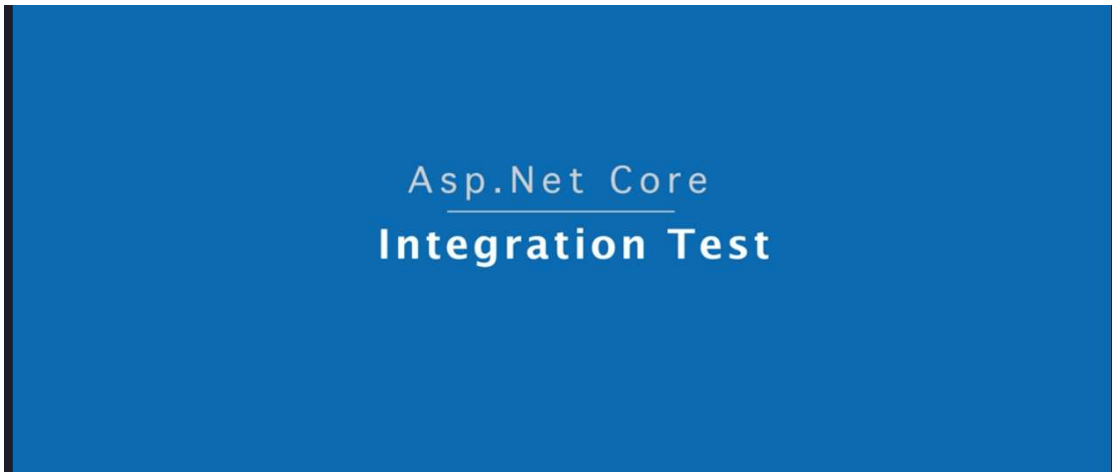
Why Mock the Service in Controller Tests?

- Since controllers **call services directly**, we don't want real service execution in unit tests.
- Instead, we **mock the service layer**, ensuring that controller tests focus only on **controller logic**.

This approach ensures **isolation**, making our tests **faster, more reliable, and easier to maintain**! 🚀

From <https://chatgpt.com/c/67add8eb-ada8-800c-9f02-fa7d4618f139>





Unit Testing and Integration Testing

Both **unit testing** and **integration testing** are equally important in application testing.
The difference between them is:

- **Unit Testing** focuses on a **single component**, such as testing a **controller** or a **service**.
- It ensures that a component works **independently**, without worrying about **underlying or higher-level components**.
- For example, while testing a **controller**, you don't check whether the **service** or **repository** is working—you only verify the **controller's return value**.

On the other hand, **Integration Testing** focuses on the **overall functionality** of the application.

- Instead of making a **real browser request**, you send a request using an **HTTP client object**.
- This request **flows through all components**, meaning it covers the **entire stack** of application layers.

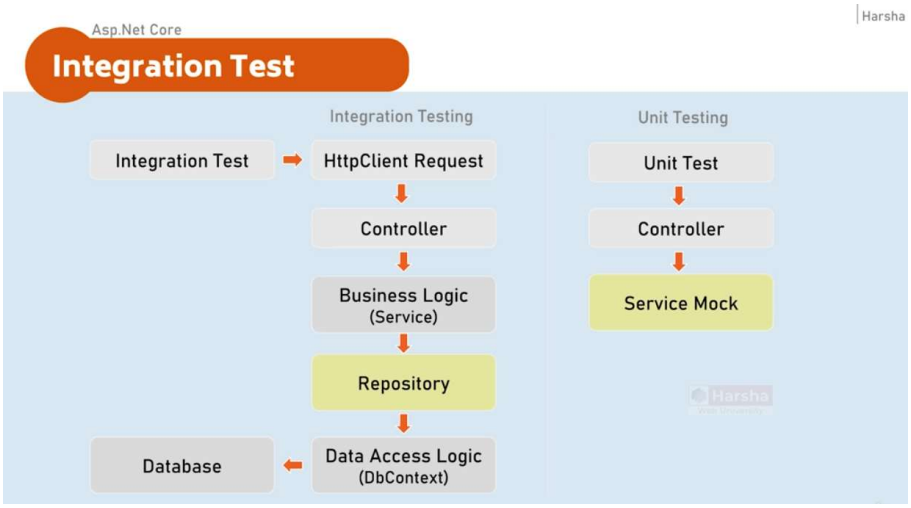
For example, a request typically follows this flow:

1. **Controller** receives the request.
2. Controller calls the **service**.
3. Service calls the **repository**.
4. Repository interacts with the **DB context**.

In **integration testing**, the same set of components execute, but instead of using a browser, we programmatically send HTTP requests using an **HTTP client object**.
The goal is to verify the **actual end result** from the controller.

- For example, if the **controller action method** returns a **ViewResult**, the test will check whether the **HTTP response** is **valid or invalid**.

Next: Creating an Integration Test in ASP.NET Core 🚀



Creating an Integration Test in ASP.NET Core

In this lecture, we will create a **simple integration test** in **ASP.NET Core**.

General Rule for Integration Testing

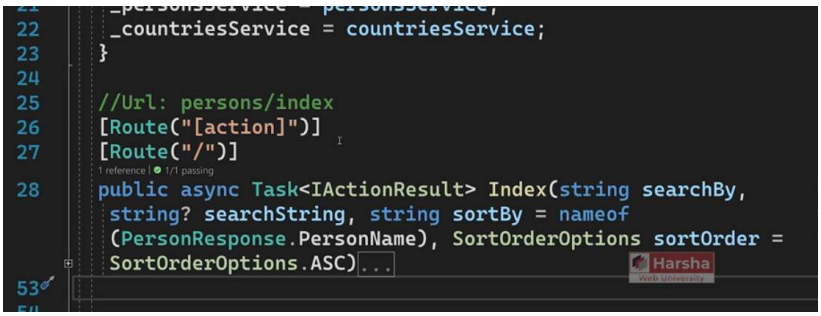
We need to create **one integration test per route** in the application.
For example, if we have a route:
/persons/index
We will write an integration test to ensure it **returns a valid response** successfully.

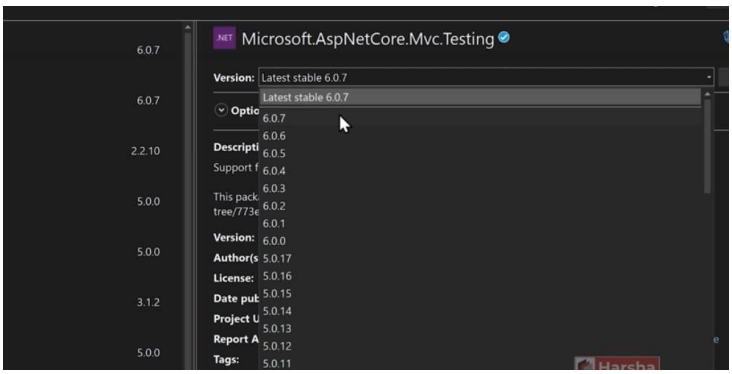
Steps to Create an Integration Test

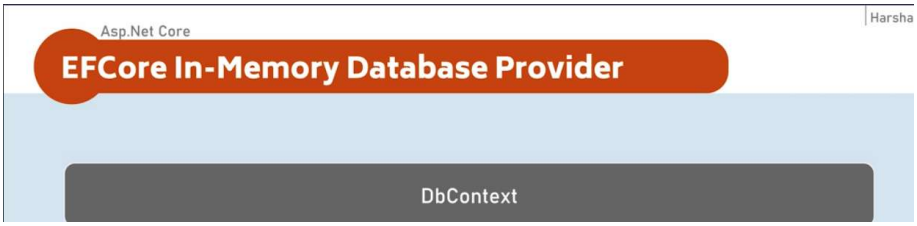
1. **Set up the test project** with the required dependencies.
2. **Send an HTTP request** to the /persons/index endpoint using an **HTTP client object**.
3. **Verify the response** to check if it returns a **valid status code** and expected data.

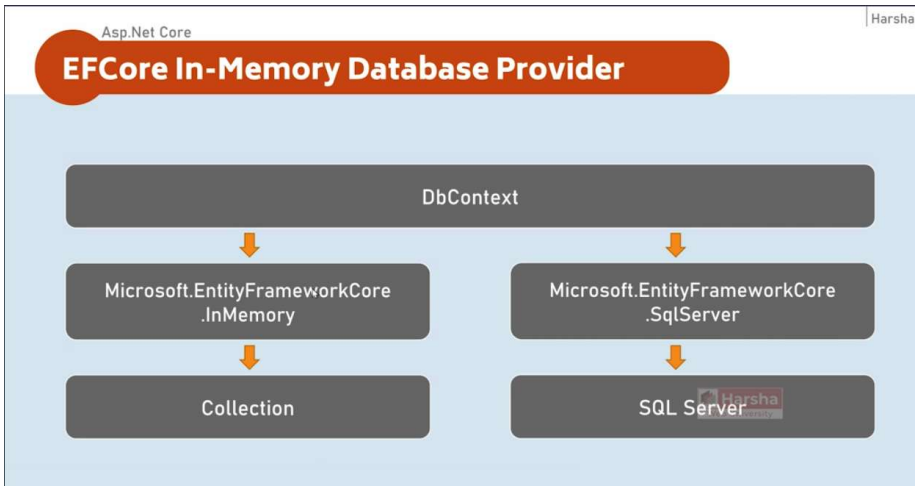
Now, let's implement it! 🚀

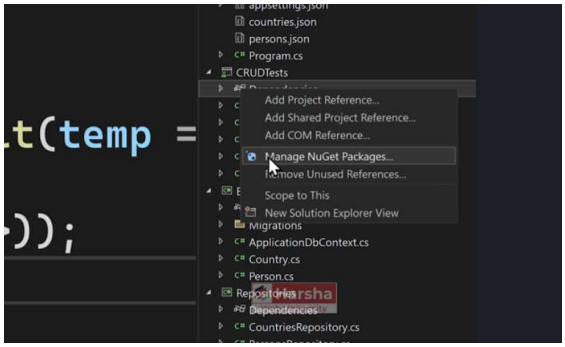
From <https://chatgpt.com/c/67add8eb-ada8-800c-9f02-fa7d4618f139>

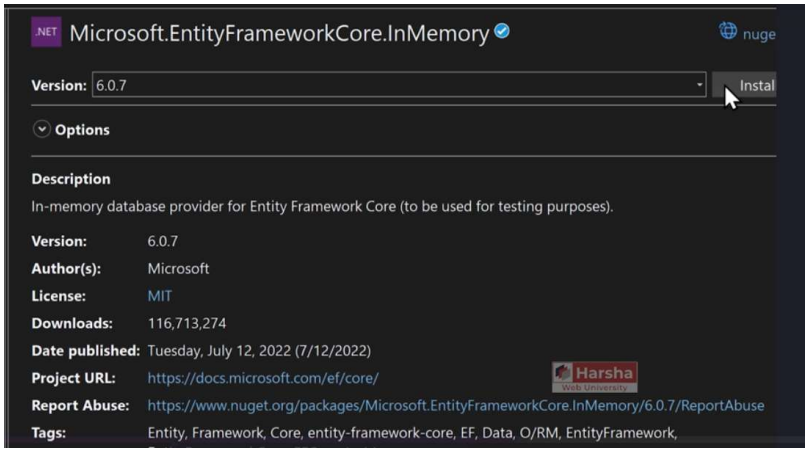




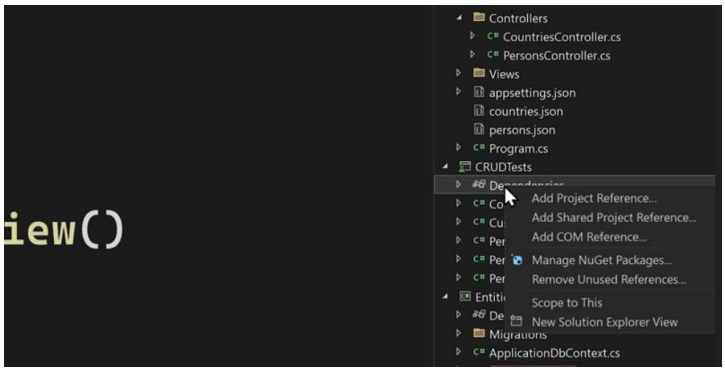


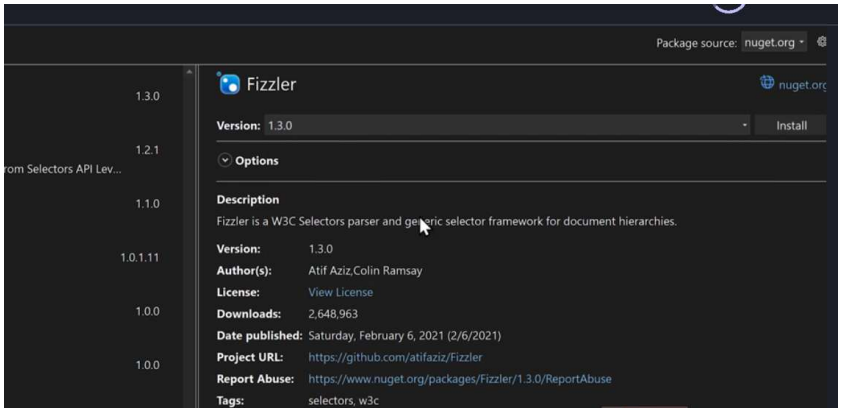

Asp.Net Core
Harsha
EFCore In-Memory Database Provider
DbContext
Microsoft.EntityFrameworkCore
.InMemory
Microsoft.EntityFrameworkCore
.SqlServer
Collection
SQL Server

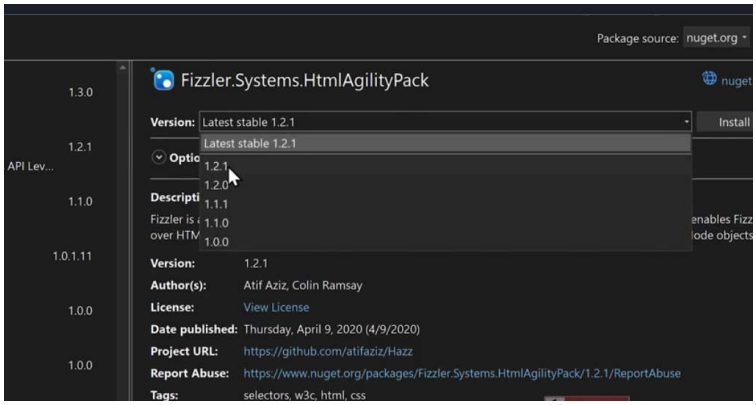

countries.json
persons.json
Program.cs
CRUDTests
Add Project Reference...
Add Shared Project Reference...
Add COM Reference...
Manage NuGet Packages...
Remove Unused References...
Scope to This
New Solution Explorer View
Migrations
ApplicationDbContext.cs
Country.cs
Person.cs
Dependencies
CountriesRepository.cs


Microsoft.EntityFrameworkCore.InMemory
nuge
Version: 6.0.7
Instal
Options
Description
In-memory database provider for Entity Framework Core (to be used for testing purposes).
Version: 6.0.7
Author(s): Microsoft
License: MIT
Downloads: 116,713,274
Date published: Tuesday, July 12, 2022 (7/12/2022)
Project URL: https://docs.microsoft.com/ef/core/
Report Abuse: https://www.nuget.org/packages/Microsoft.EntityFrameworkCore.InMemory/6.0.7/ReportAbuse
Tags: Entity, Framework, Core, entity-framework-core, EF, Data, O/RM, EntityFramework,
Harsha


Asp.Net Core
Integration Testing with Response Body


iew()
Controllers
CountriesController.cs
PersonsController.cs
Views
appsettings.json
countries.json
persons.json
Program.cs
CRUDTests
Add Project Reference...
Add Shared Project Reference...
Add COM Reference...
Manage NuGet Packages...
Remove Unused References...
Scope to This
New Solution Explorer View
ApplicationDbContext.cs


Package source: nuget.org
Fizzler
Version: 1.3.0
Install
Options
Description
Fizzler is a W3C Selectors parser and generic selector framework for document hierarchies.
Version: 1.3.0
Author(s): Atif Aziz,Colin Ramsay
License: View License
Downloads: 2,648,963
Date published: Saturday, February 6, 2021 (2/6/2021)
Project URL: https://github.com/atifaziz/Fizzler
Report Abuse: https://www.nuget.org/packages/Fizzler/1.3.0/ReportAbuse
Tags: selectors, w3c


Package source: nuget.org
Fizzler.Systems.HtmlAgilityPack
Version: Latest stable 1.2.1
Install
Latest stable 1.2.1
1.2.1
1.2.0
1.1.1
1.1.0
1.0.0
Version: 1.2.1
Author(s): Atif Aziz, Colin Ramsay
License: View License
Date published: Thursday, April 9, 2020 (4/9/2020)
Project URL: https://github.com/atifaziz/Hazz
Report Abuse: https://www.nuget.org/packages/Fizzler.Systems.HtmlAgilityPack/1.2.1/ReportAbuse
Tags: selectors, w3c, html, css