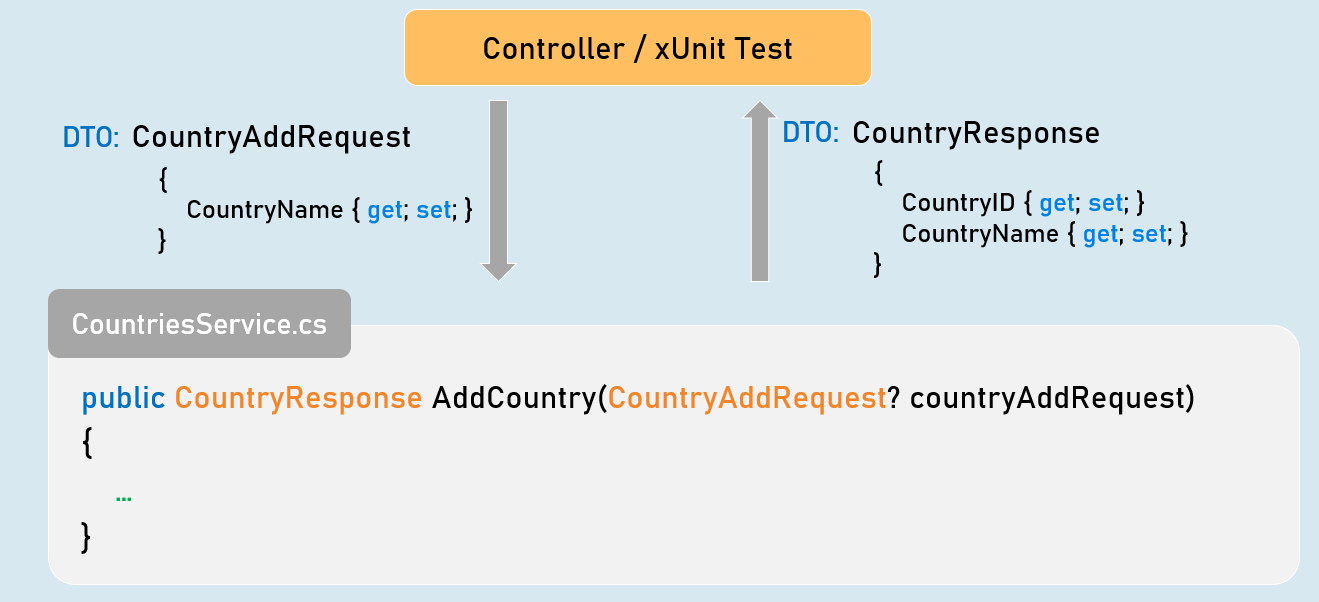
Introduction to xUnit

xUnit is the free, open source unit testing tool for .NET Framework.

Easy and extensible.

Best to use with a mocking framework called "Moq".

Add Country - xUnit Test



public CountryResponse AddCountry(CountryAddRequest? countryAddRequest)

{

//Check if "countryAddRequest" is not null.

//Validate all properties of "countryAddRequest"

//Convert "countryAddRequest" from "CountryAddRequest" type to "Country".

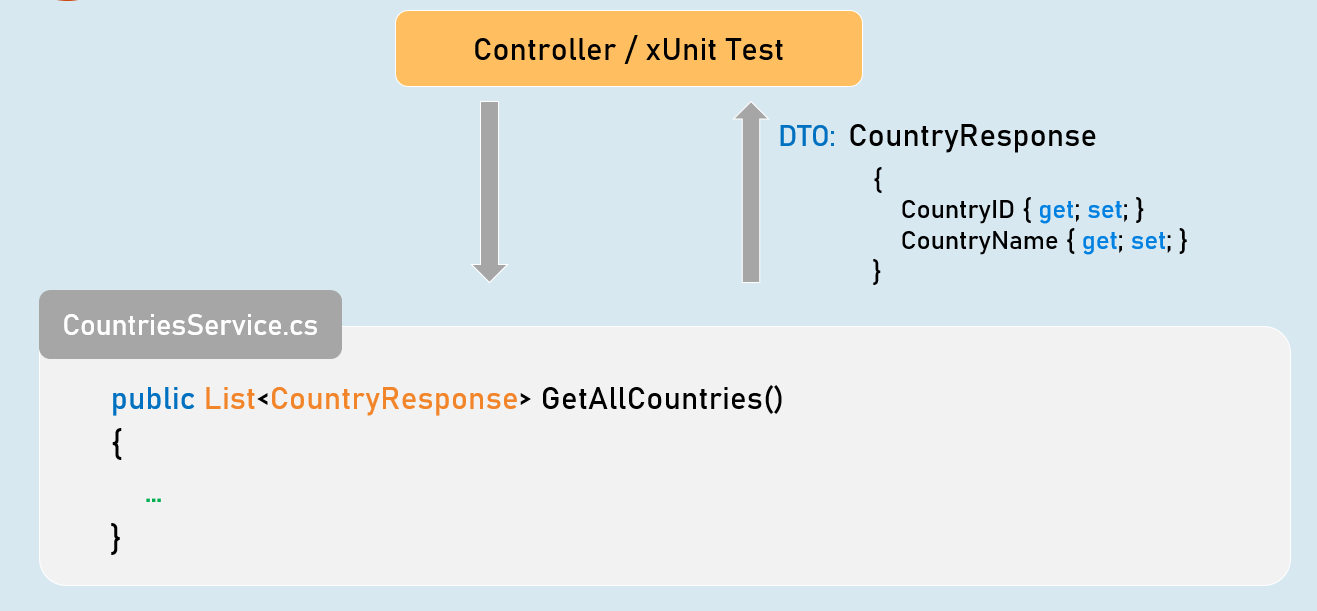
//Generate a new CountryID

//Then add it into List<Country>

//Return CountryResponse object with generated CountryID

}

Get All Countries - xUnit Test



public List<CountryResponse> GetAllCountries()

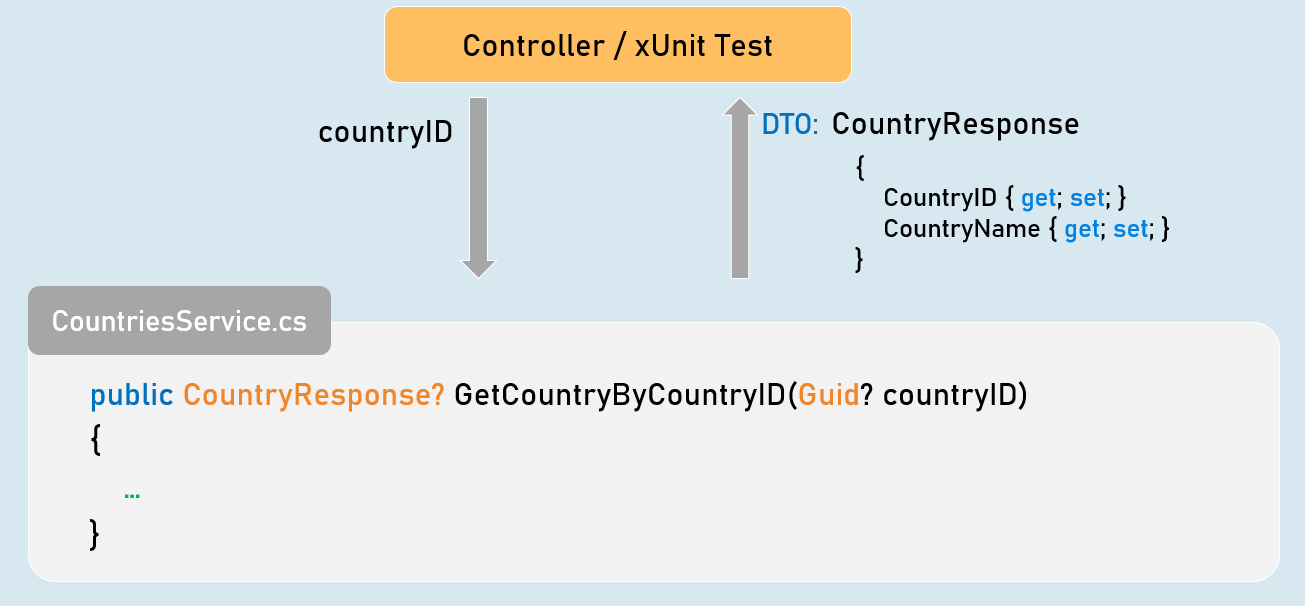
{

//Convert all countries from "Country" type to "CountryResponse" type.

//Return all CountryResponse objects

}

Get Country by Country ID - xUnit Test



public CountryResponse? GetCountryByCountryID(Guid? countryID)

{

//Check if "countryID" is not null.

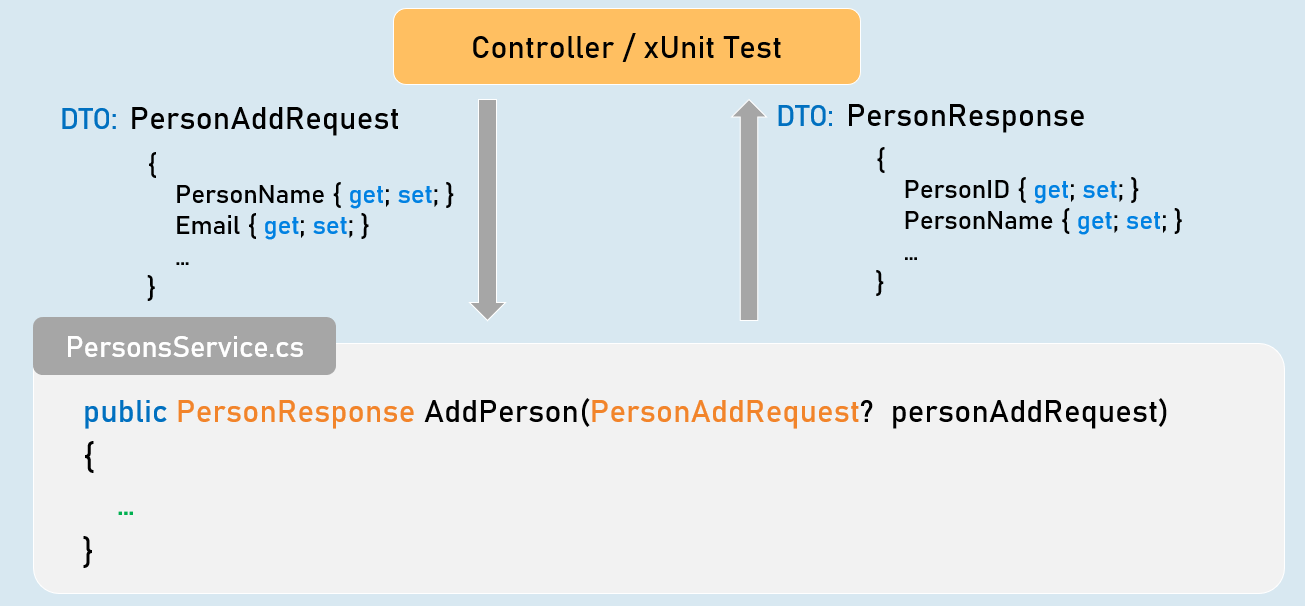
//Get matching country from List<Country> based countryID.

//Convert matching country object from "Country" to "CountryResponse" type.

//Return CountryResponse object

}

Add Person - xUnit Test



public PersonResponse AddPerson(PersonAddRequest? personAddRequest)

{

//Check if "personAddRequest" is not null.

//Validate all properties of "personAddRequest".

//Convert "personAddRequest" from "PersonAddRequest" type to "Person".

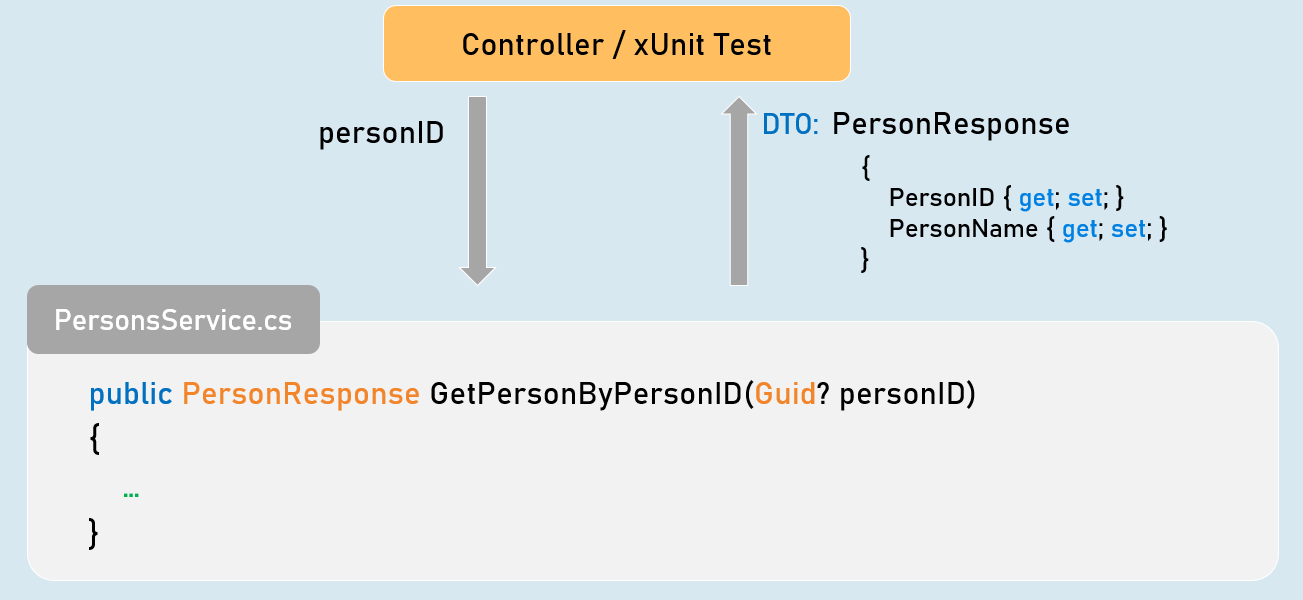
//Generate a new PersonID.

//Then add it into List<Person>.

//Return PersonResponse object with generated PersonID.

}

Get Person by Person ID - xUnit Test



public PersonResponse GetPersonByPersonID(Guid? personID)

{

//Check if "personID" is not null.

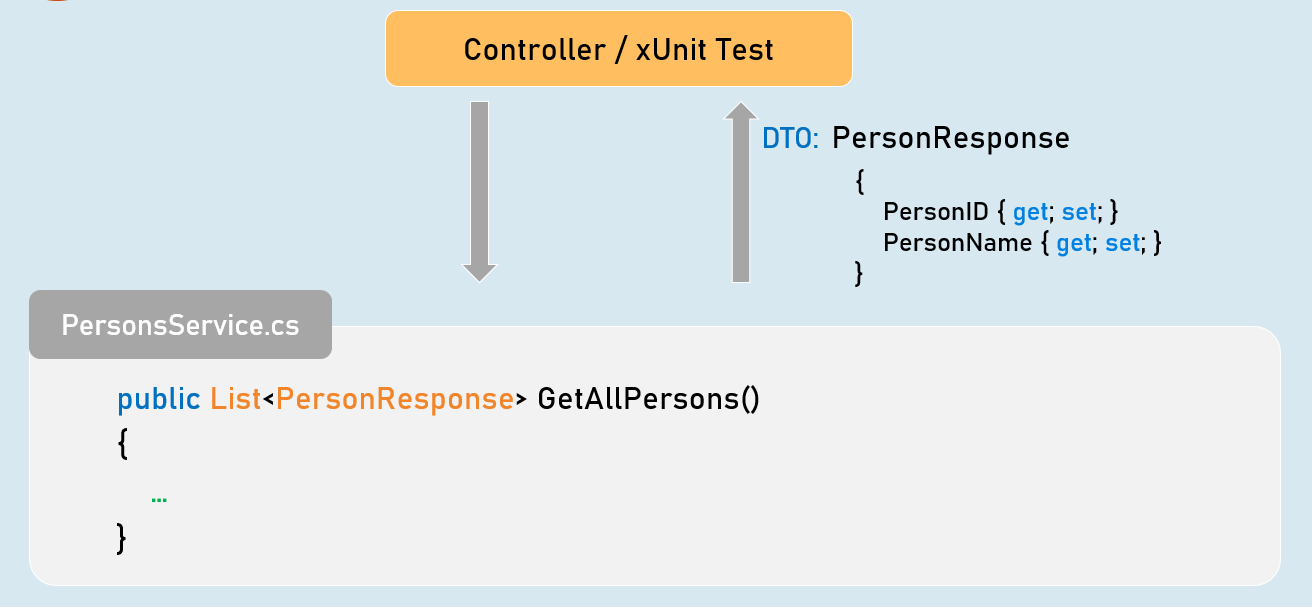
//Get matching person from List<Person> based personID.

//Convert matching person object from "Person" to "PersonResponse" type.

//Return PersonResponse object

}

Get All Persons - xUnit Test



public List<PersonResponse> GetAllPersons()

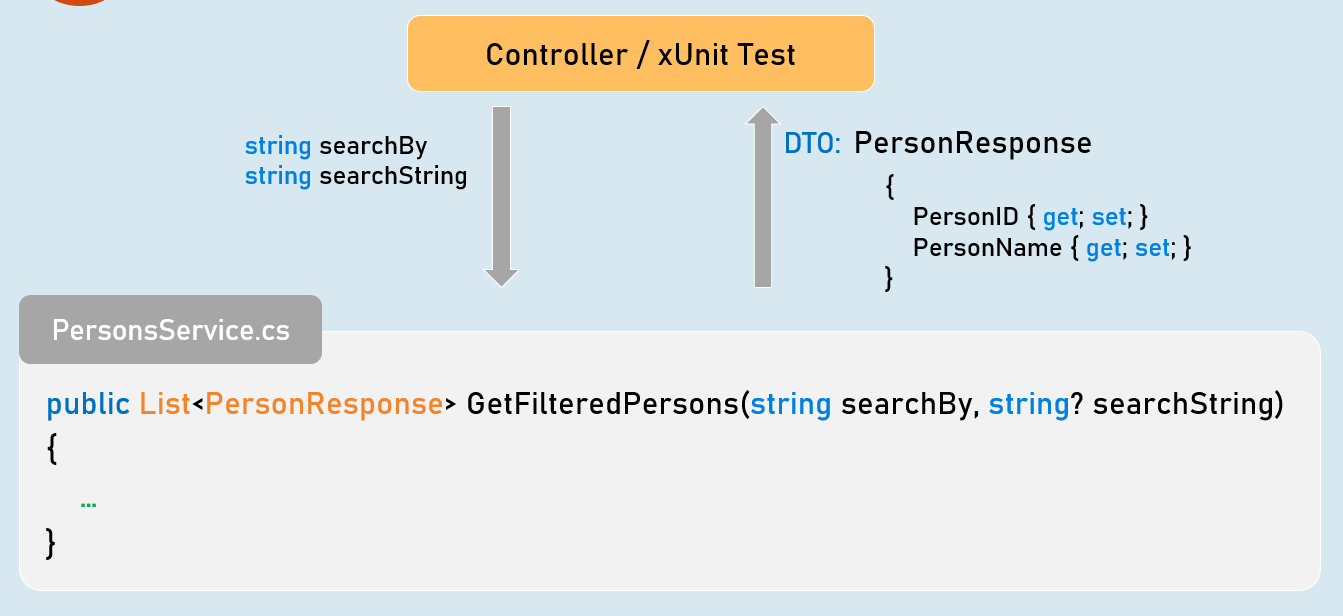
{

//Convert all persons from "Person" type to "PersonResponse" type.

//Return all PersonResponse objects

}

Get Filtered Persons - xUnit Test



public List<PersonResponse> GetFilteredPersons(string searchBy, string? searchString)

{

//Check if "searchBy" is not null.

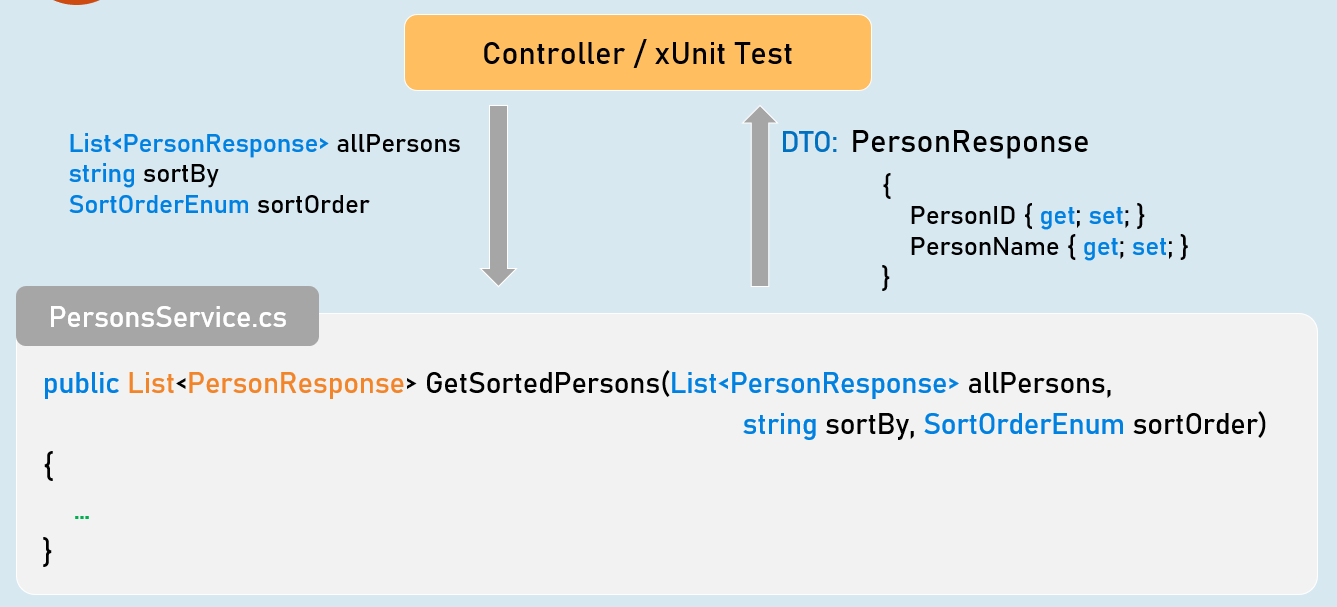
//Get matching persons from List<Person> based on given searchBy and searchString.

//Convert the matching persons from "Person" type to "PersonResponse" type.

//Return all matching PersonResponse objects

}

Get Sorted Persons - xUnit Test



public List<PersonResponse> GetSortedPersons(List<PersonResponse> allPersons,

string sortBy, SortOrderEnum sortOrder)

{

//Check if "sortBy" is not null.

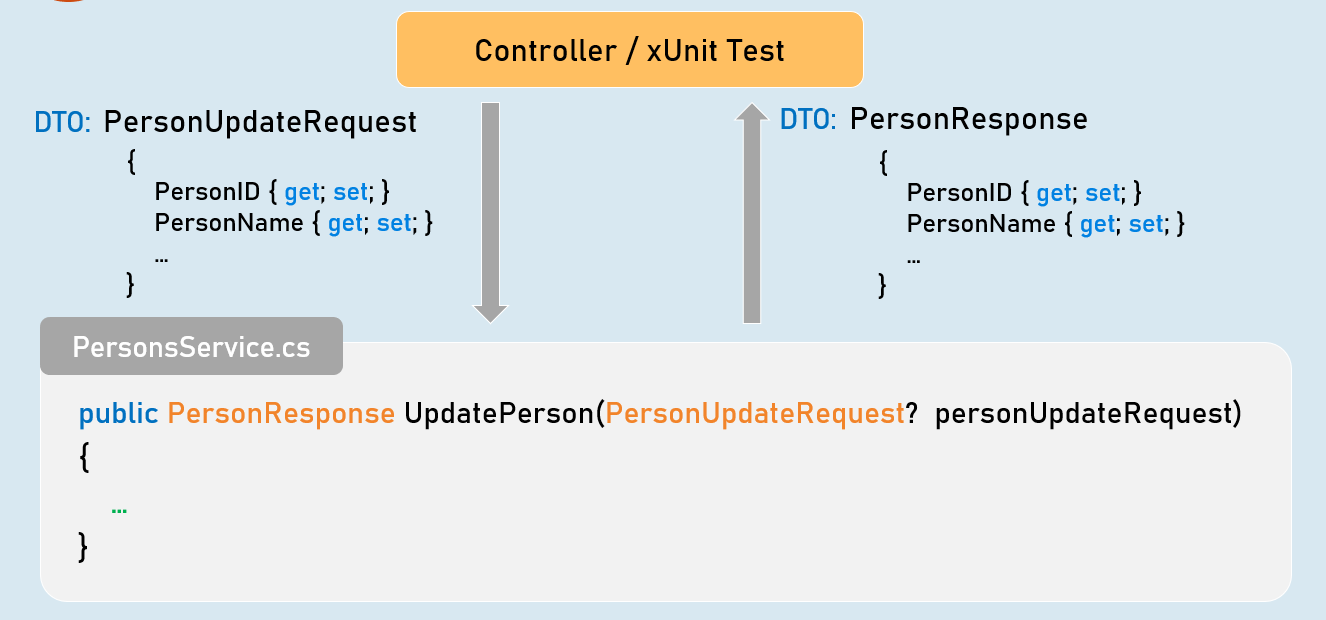
//Get sorted persons from "allPersons" based on given "sortBy" and "sortOrder".

//Convert the sorted persons from "Person" type to "PersonResponse" type.

//Return all sorted PersonResponse objects

}

Update Person - xUnit Test



public PersonResponse UpdatePerson(PersonUpdateRequest? personUpdateRequest)

{

//Check if "personUpdateRequest" is not null.

//Validate all properties of "personUpdateRequest"

//Get the matching "Person" object from List<Person> based on PersonID.

//Check if matching "Person" object is not null

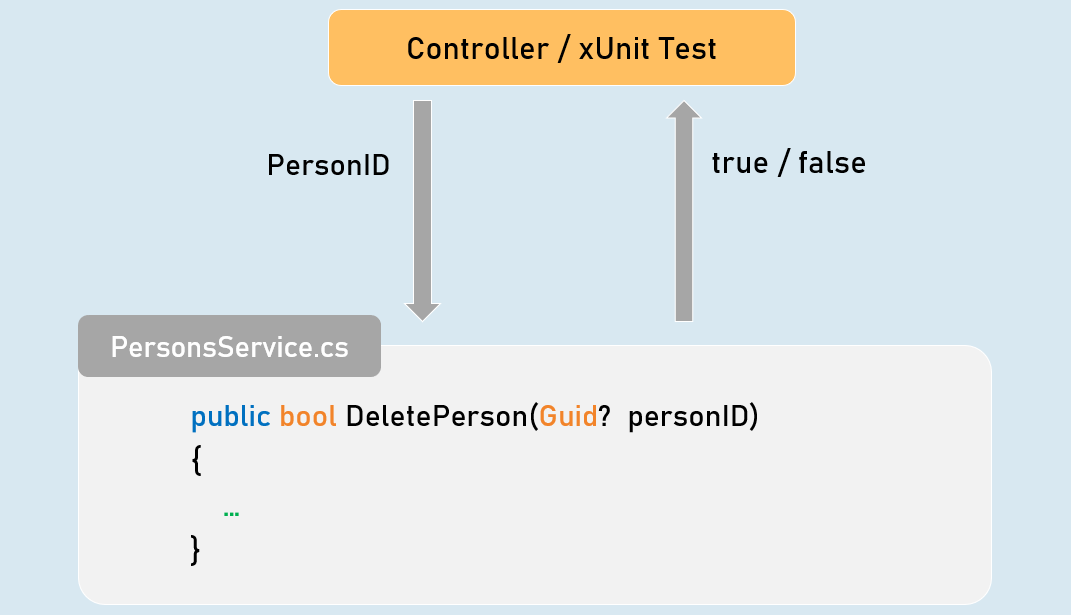
//Update all details from "PersonUpdateRequest" object to "Person" object

//Convert the person object from "Person" to "PersonResponse" type

//Return PersonResponse object with updated details

}

Delete Person - xUnit Test



public bool DeletePerson(Guid? personID)

{

//Check if "personID" is not null.

//Get the matching "Person" object from List<Person> based on PersonID.

//Check if matching "Person" object is not null

//Delete the matching "Person" object from List<Person>

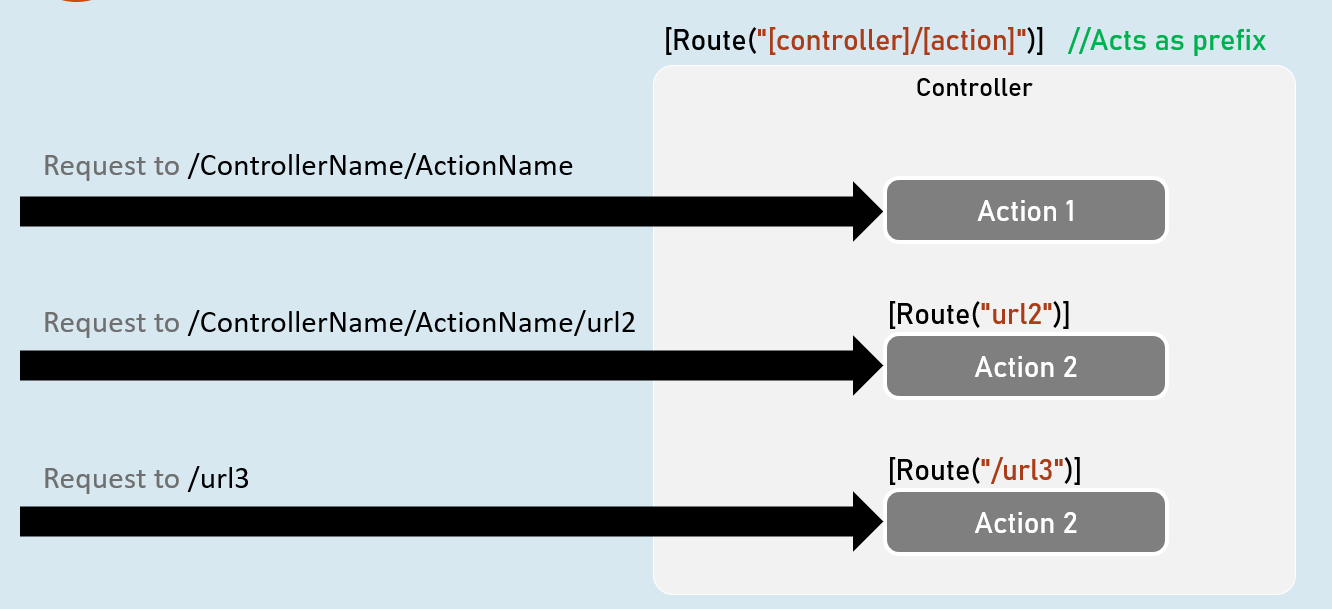
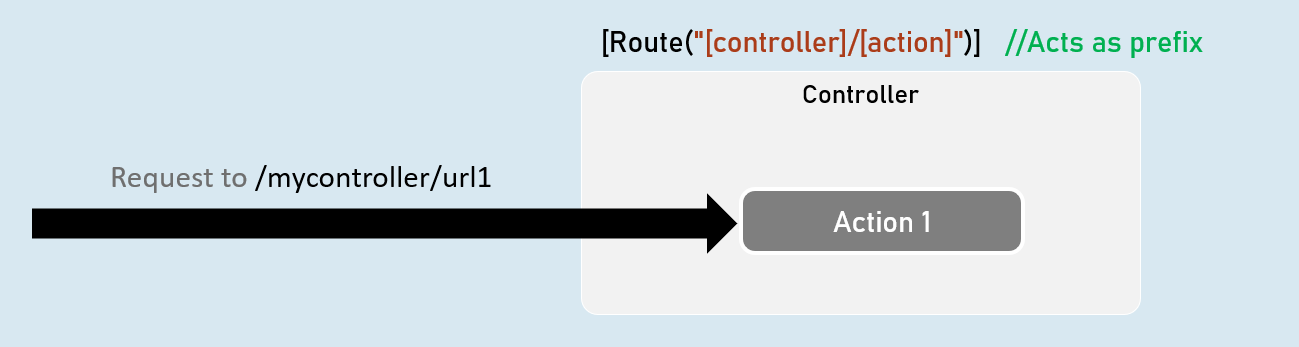
//Return Boolean value indicating whether person object was deleted or not

}

Route Tokens

The route tokens [controller], [action] can be used to apply common-patterned routes for all action methods.

The route of controller acts as a prefix for the route of actions

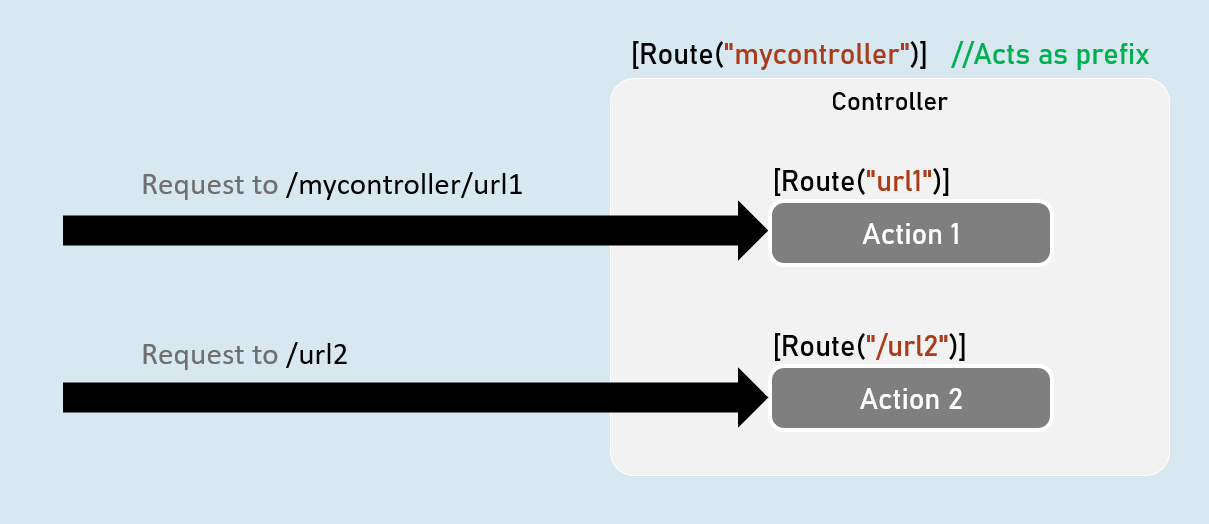
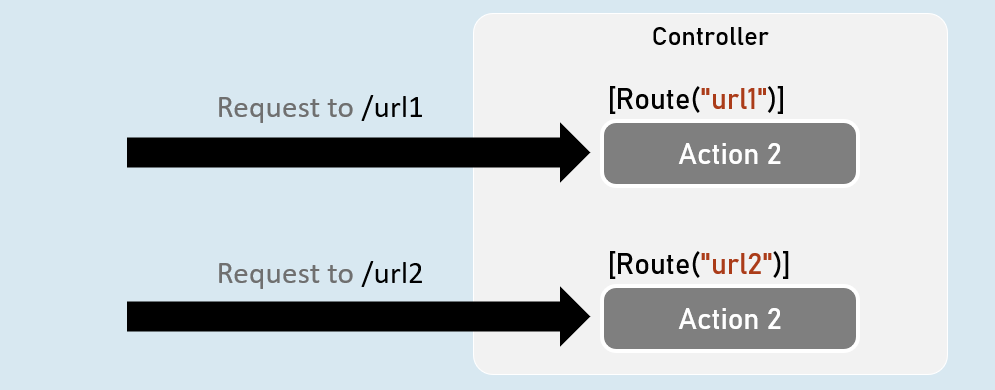


Attribute Routing

[Route] attribute specifies route for an action method or controller.

The route of controller acts as a prefix for the route of actions.

**[Route] - Attribute Routing**



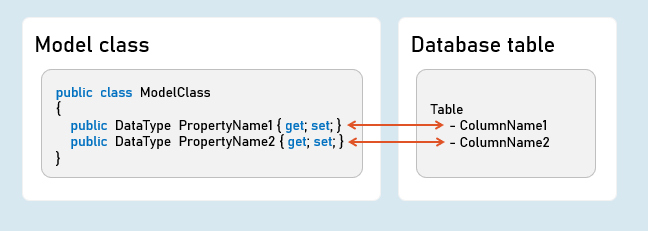
Introduction to EntityFrameworkCore

EntityFrameworkCore is light-weight, extensible and cross-platform framework for accessing databases in .NET applications.

It is the most-used database framework for Asp.Net Core Apps.



**EFCore Models**



**Pros & Cons of EntityFrameworkCore**

**Shorter Code**

The CRUD operations / calling stored procedures are done with shorter amount of code than ADO.NET.

**Performance**

EFCore performs slower than ADO.NET.

So ADO.NET or its alternatives (such as Dapper) are recommended for larger & high-traffic applications.

**Strongly-Typed**

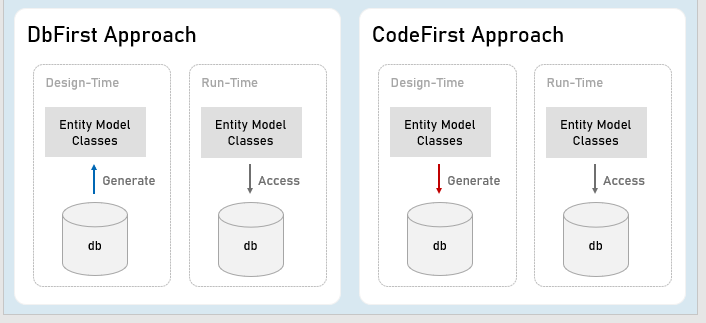
The columns as created as properties in model class.

So the Intellisense offers columns of the table as properties, while writing the code.

Plus, the developer need not convert data types of values; it's automatically done by EFCore itself.

Approaches in Entity Framework Core

**EFCore Approaches**



Pros and Cons of EFCore Approaches

**CodeFirst Approach**

Suitable for newer databases.

Manual changes to DB will be most probably lost because your code defines the database.

Stored procedures are to be written as a part of C# code.

Suitable for smaller applications or prototype-level applications only; but not for larger or high data-intense applications.

**DbFirst Approach**

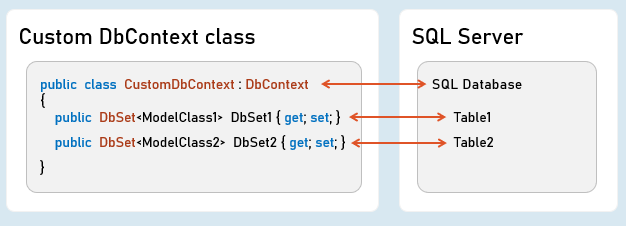
Suitable if you have an existing database or DB designed by DBAs, developed separately.

Manual changes to DB can be done independently.

Stored procedures, indexes, triggers etc., can be created with T-SQL independently.

Suitable for larger applications and high data-intense applications.

DbContext and DbSet



**DbContext**

An instance of DbContext is responsible to hold a set of DbSets' and represent a connection with database.

**DbSet**

Represents a single database table; each column is represented as a model property.

**Add DbContext as Service in Program.cs:**

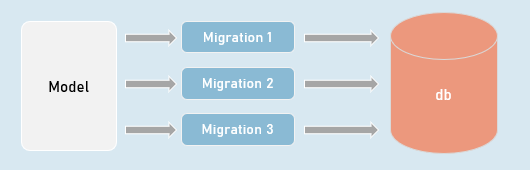
builder.Services.AddDbContext<DbContextClassName>( options => {

options.UseSqlServer();

}

);

Code-First Migrations



**Migrations**

Creates or updates database based on the changes made in the model.

in**Package Manager Console (PMC):**

Add-Migration MigrationName

//Adds a migration file that contains C# code to update the database

Update-Database -Verbose

//Executes the migration; the database will be created or table schema gets updated as a result.

Seed Data

in DbContext:

modelBuilder.Entity<ModelClass>().HasData(entityObject);

It adds initial data (initial rows) in tables, when the database is newly created.

EF CRUD Operations - Query

**SELECT - SQL**

SELECT Column1, Column2 FROM TableName

WHERE Column = value

ORDER BY Column

**LINQ Query:**

\_dbContext.DbSetName

.Where(item => item.Property == value)

.OrderBy(item => item.Property)

.Select(item => item);

//Specifies condition for where clause

//Specifies condition for 'order by' clause

//Expression to be executed for each row

EF CRUD Operations - Insert

**INSERT - SQL**

INSERT INTO TableName(Column1, Column2) VALUES (Value1, Value2)

**Add:**

\_dbContext.DbSetName.Add(entityObject);

//Adds the given model object (entity object) to the DbSet.

**SaveChanges()**

\_dbContext.SaveChanges();

//Generates the SQL INSERT statement based on the model object data and executes the same at database server.

EF CRUD Operations - Delete

**DELETE - SQL**

DELETE FROM TableName WHERE Condition

**Remove:**

\_dbContext.DbSetName.Remove(entityObject);

//Removes the specified model object (entity object) to the DbSet.

**SaveChanges()**

\_dbContext.SaveChanges();

//Generates the SQL DELETE statement based on the model object data and executes the same at database server.

EF CRUD Operations - Update

**UPDATE - SQL**

UPDATE TableName SET Column1 = Value1, Column2 = Value2 WHERE PrimaryKey = Value

**Update:**

entityObject.Property = value;

//Updates the specified value in the specific property of the model object (entity object) to the DbSet.

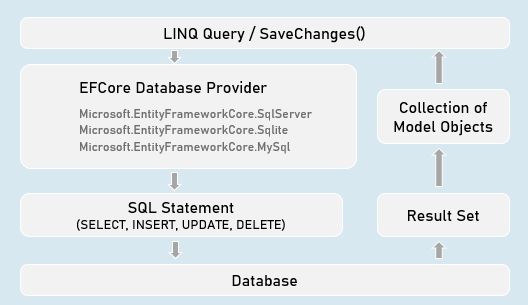
**SaveChanges()**

\_dbContext.SaveChanges();

//Generates the SQL UPDATE statement based on the model object data and executes the same at database server.

How EF Query Works?

**Workflow of Query Processing in EF**



EF - Calling Stored Procedures

**Stored Procedure for CUD (INSERT | UPDATE | DELETE):**

int DbContext.Database.ExecuteSqlRaw(

string sql,

params object[] parameters)

//Eg: "EXECUTE [dbo].[StoredProcName] @Param1 @Parm2

//A list of objects of SqlParameter type

**Stored Procedure for Retrieving (Select):**

IQueryable<Model> DbSetName.FromSqlRaw(

string sql,

paramsobject[] parameters)

//Eg: "EXECUTE [dbo].[StoredProcName] @Param1 @Parm2"

//A list of objects of SqlParameter type

**Creating Stored Procedure (SQL Server)**

CREATE PROCEDURE [schema].[procedure\_name]

(@parameter\_name data\_type, @parameter\_name data\_type)

AS BEGIN

statements

END

Advantages of Stored Procedure

**Single database call**

You can execute multiple / complex SQL statements with a single database call.

As a result, you'll get:

Better performance (as you reduce the number of database calls)

Complex database operations such as using temporary tables / cursors becomes easier.

**Maintainability**

The SQL statements can be changed easily WITHOUT modifying anything in the application source code (as long as inputs and outputs doesn't change)

[Column] Attribute

**Model class**

public class ModelClass

{

[Column("ColumnName", TypeName = "datatype")]

public DataType PropertyName { get; set; }

[Column("ColumnName", TypeName = "datatype")]

publicDataTypePropertyName { get; set; }

}

Specifies column name and data type of SQL Server table.

EF - Fluent API

**DbContext class**

public class CustomDbContext : DbContext

{

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

//Specify table name (and schema name optionally) to be mapped to the model class

modelBuilder.Entity<ModelClass>( ).ToTable("table\_name", schema: "schema\_name");

//Specify view name (and schema name optionally) to be mapped to the model class

modelBuilder.Entity<ModelClass>( ).ToView("view\_name", schema: "schema\_name");

//Specify default schema name applicable for all tables in the DbContext

modelBuilder.HasDefaultSchema("schema\_name");

}

}

public class CustomDbContext : DbContext

{

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<ModelClass>( ).Property(temp => temp.PropertyName)

.HasColumnName("column\_name") //Specifies column name in table

.HasColumnType("data\_type") //Specifies column data type in table

.HasDefaultValue("default\_value") //Specifies default value of the column

}

}

public class CustomDbContext : DbContext

{

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

//Adds database index for the specified column for faster searches

modelBuilder.Entity<ModelClass>( ).HasIndex("column\_name").IsUnique();

//Adds check constraint for the specified column - that executes for insert & update

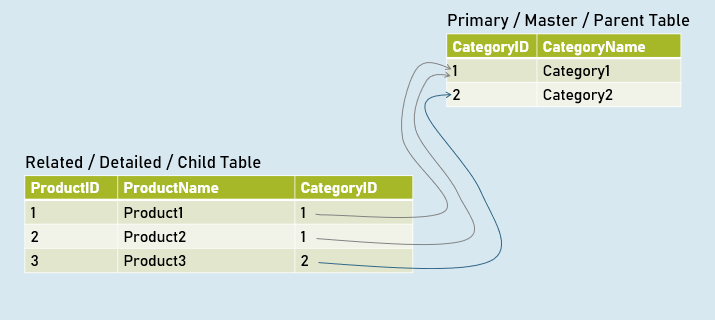
modelBuilder.Entity<ModelClass>( ).HasCheckConstraint("constraint\_name", "condition");

}

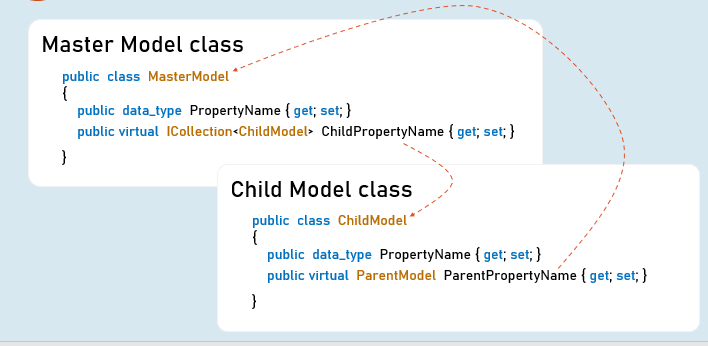
}

EF - Table Relations with Fluent API

**Table Relations**



**EF - Table Relations with Navigation Properties**



**EF - Table Relations with Fluent API**

**DbContext class**

public class CustomDbContext : DbContext

{

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

//Specifies relation between primary key and foreign key among two tables

modelBuilder.Entity<ChildModel>( )

.HasOne<ParentModel>(parent => parent.ParentReferencePropertyInChildModel)

.WithMany(child => child.ChildReferencePropertyInParentModel) //optional

.HasForeignKey(child => child.ForeignKeyPropertyInChildModel)

}

}

EF - Async Operations

**async**

The method is awaitable.

Can execute I/O bound code or CPU-bound code

**await**

Waits for the I/O bound or CPU-bound code execution gets completed.

After completion, it returns the return value.

Generate PDF Files



**Rotativa.AspNetCore:**

using Rotativa.AspNetCore;

using Rotativa.AspNetCore.Options;

return new ViewAsPdf("ViewName", ModelObject, ViewData)

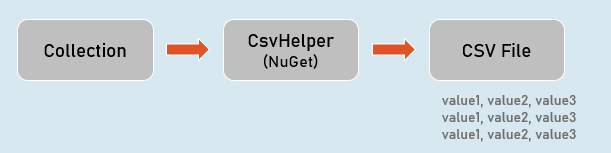
{

PageMargins = new Margins() { Top = 1, Right = 2, Bottom = 3, Left = 4 },

PageOrientation = Orientation.Landscape

}

Generate CSV Files (CSVHelper)



CsvWriter:

**WriteRecords(records)**

Writes all objects in the given collection.

Eg:

1,abc

2,def

**WriteHeader<ModelClass>( )**

Writes all property names as headings.

Eg:

Id, Name

**WriteRecord(record)**

Writes the given object as a row.

Eg:

1, abc

**WriteField( value )**

Writes given value.

**NextRecord( )**

Moves to the next line.

**Flush( )**

Writes the current data to the stream.

Generate Excel Files (EPPlus)



ExcelWorksheet

["cell\_address"].Value

Sets or gets value at the specified cell.

["cell\_address"].Style

Sets or gets formatting style of the specific cell.

Best Practices of Unit Tests

**Isolated / Stand-alone**

(separated from any other dependencies such as file system or database)

**Test single method at-a-time**

(should not test more than one method in a single test case)

**Unordered**

(can be executed in any order)

**Fast**

(Tests should take little time to run (about few milliseconds))

**Repeatable**

(Tests can run repeatedly but should give same result, if no changes in the actual source code)

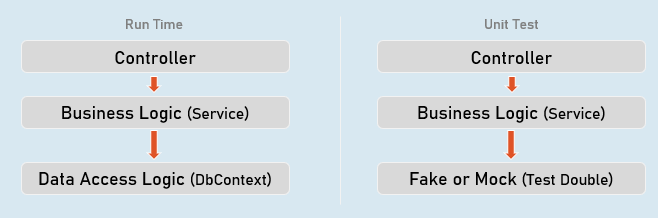
**Timely**

(Time taken for writing a test case should not take longer time, than then time taken for writing the code that is being tested)

Mocking the DbContext

**Test Double**

A "test double" is an object that look and behave like their production equivalent objects.



A "test double" is an object that look and behave like their production equivalent objects.

**Fake**

An object that providers an alternative (dummy) implementation of an interface

**Mock**

An object on which you fix specific return value for each individual method or property, without actual / full implementation of it.

**Mocking the DbContext**

Install-Package Moq

Install-Package EntityFrameworkCoreMock.Moq

**Mocking the DbContext:**

var dbContextOptions = new DbContextOptionsBuilder<DbContextClassName>().Options;

//mock the DbContext

DbContextMock<DbContextClass> dbContextMock = new DbContextMock<DbContextClass>(dbContextOptions);

var initialData = new List<ModelClass>() { … };

//mock the DbSet

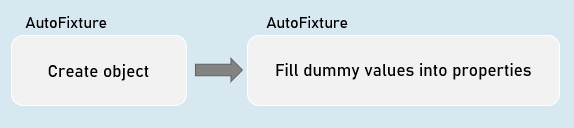
var dbSetMock = dbContextMock.CreateDbSetMock(temp => temp.DbSetName, initialData);

//create service instance with mocked DbContext

var service = newServiceClass(dbContextMock.Object);

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

**AutoFixture**

Install-Package AutoFixture

**Working with AutoFixture:**

var fixture = new Fixture();

//Simple AutoFixture

var obj1 = fixture.Create<ModelClass>();

//Customization with AutoFixture

var obj2 = fixture.Build<ModelClass>()

.With(temp => temp.Property1, value)

.With(temp => temp.Property2, value)

.Create();

Fluent Assertions

Fluent Assertions are a set of extension methods to make the assertions in unit testing more readable and human-friendly.

Install-Package FluentAssertions

**Assert**

//Equal

Assert.Equal(expected, actual);

//Not Equal

Assert.NotEqual(expected, actual);

//Null

Assert.Null(actual);

//Not Null

Assert.NotNull(actual);

//True

Assert.True(actual);

//False

Assert.False(actual);

//Empty

Assert.Empty(actual);

//Not Empty

Assert.NotEmpty(actual);

//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

//number should be positive

Assert.True(actual > 0);

//number should be negative

Assert.True(actual < 0);

//number should be >= expected

Assert.True(actual >= expected);

//number should be <= expected

Assert.True(actual <= expected);

//number should be in given range

Assert.True(actual >= minimum && actual <= maximum);

//number should not be in given range

Assert.True(actual < minimum || actual > maximum);

//check data type

Assert.IsType<ExpectedType>(actual);

//Compare properties of two objects (Equals method SHOULD BE overridden)

Assert.Equal(expected, actual);

//Compare properties (should not be equal) of two objects (Equals method SHOULD BE overridden)

Assert.NotEqual(expected, actual);

**Fluent Assertion**

//Equal

actual.Should().Be(expected);

//Not Equal

actual.Should().NotBe(expected);

//Null

actual.Should().BeNull();

//Not Null

actual.Should().NotBeNull();

//True

actual.Should().BeTrue();

//False

actual.Should().BeFalse();

//Empty

actual.Should().BeEmpty();

//Not Empty

actual.Should().NotBeEmpty();

//Null or empty

actual.Should().BeNullOrEmpty();

//Should not be null or empty

actual.Should().NotBeNullOrEmpty();

//number should be positive

actual.Should().BePositive();

//number should be negative

actual.Should().BeNegative();

//number should be >= expected

actual.Should().BeGreaterThanOrEqualTo(expected);

//number should be <= expected

actual.Should().BeLessThanOrEqualTo(expected);

//number should be in given range

actual.Should().BeInRange(minimum, maximum);

//number should not be in given range

actual.Should().NotBeInRange(minimum, maximum);

//number should be in given range

actual.Should().BeInRange(minimum, maximum);

//number should not be in given range

actual.Should().NotBeInRange(minimum, maximum);

//check data type (same type)

actual.Should().BeOfType<ExpectedType>();

//check data type (same type or derived type)

actual.Should().BeAssignableTo<ExpectedType>();

//Compare properties of two objects (Equals method NEED NOT be overridden)

actual.Should().BeEquivalentTo(expected);

//Compare properties (should not equal) of two objects (Equals method NEED NOT be overridden)

actual.Should().BeNotEquivalentTo(expected);

**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);

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);

actualCollection.Should().NotBeInAscendingOrder(temp => temp.Property);

actualCollection.Should().NotBeInDescendingOrder(temp => temp.Property);

delegateObj.Should().Throw<ExceptionType>();

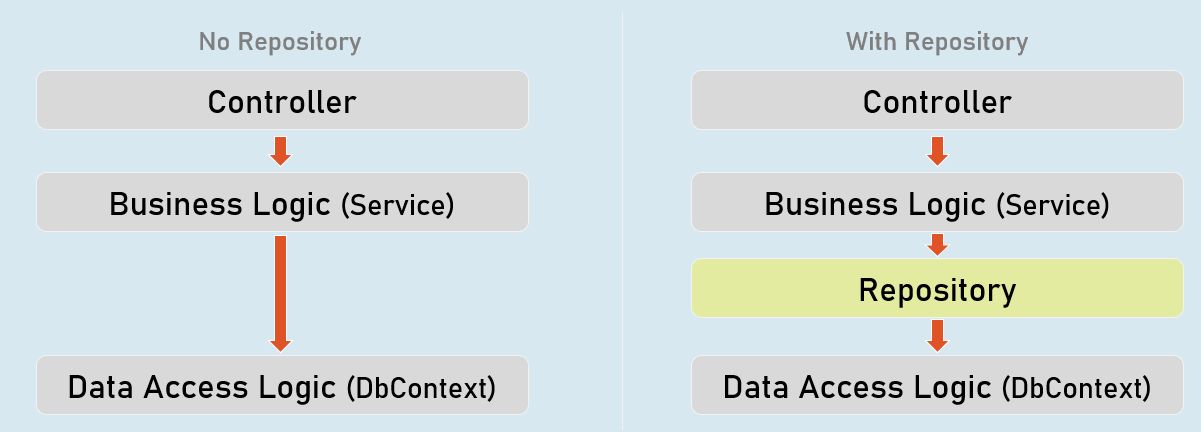
delegateObj.Should().NotThrow<ExceptionType>();

await delegateObj.Should().ThrowAsync<ExceptionType>();

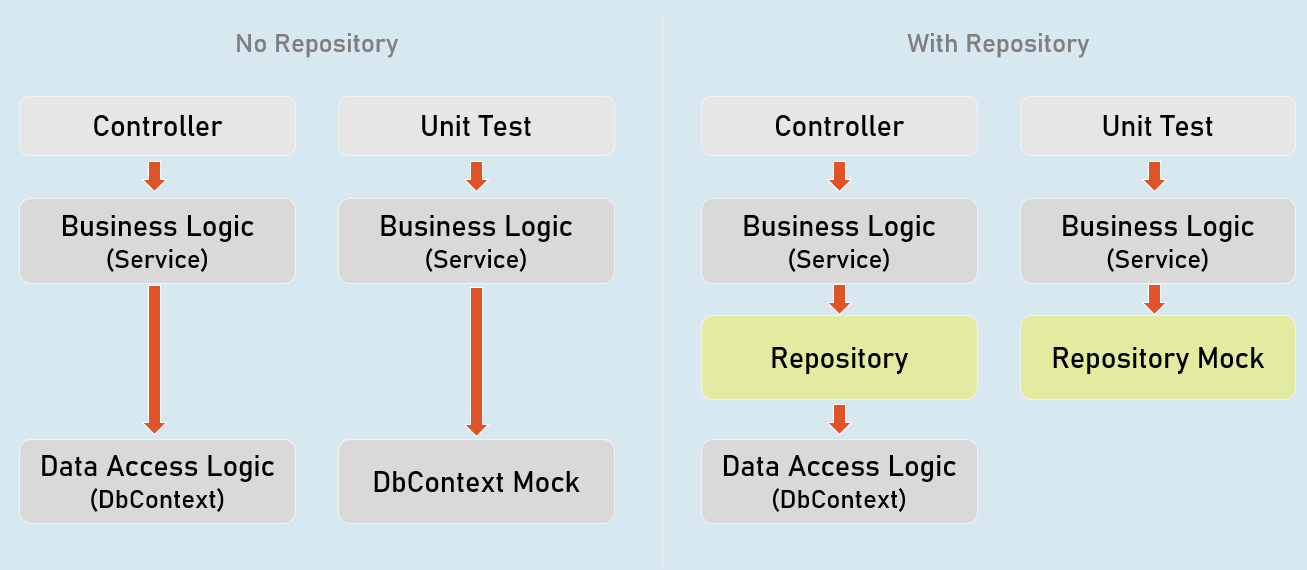
await delegateObj.Should().NotThrowAsync<ExceptionType>();

Repository

Repository (or Repository Pattern) is an abstraction between Data Access Layer (EF DbContext) and business logic layer (Service) of the application.



**Unit Testing**



Benefits of Repository Pattern

**Loosely-coupled business logic (service) & data access.**

(You can independently develop them).

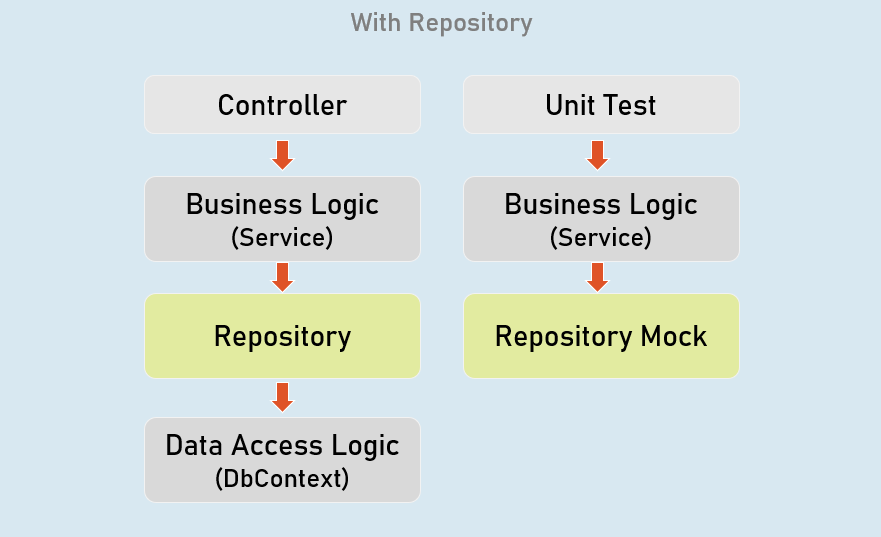
**Changing data store**

(You can create alternative repository implementation for another data store, when needed).

**Unit Testing**

(Mocking the repository is much easier (and preferred) than mocking DbContext).

Mocking the Repository



Install-Package Moq

**Mocking the Repository:**

//mock the repository

Mock<IRepository> repositoryMock = new Mock<IRepository>();

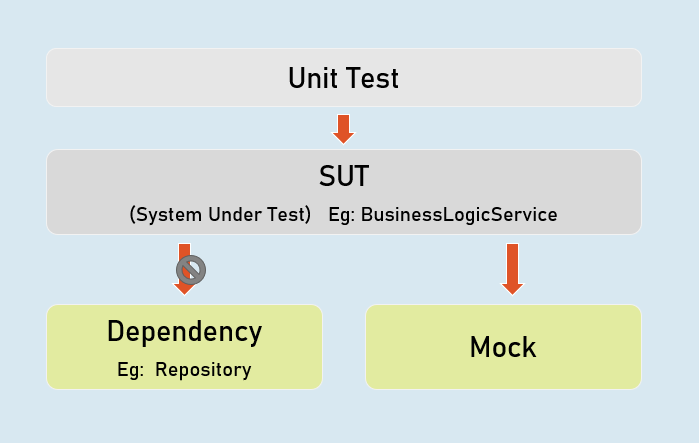
//mock a method repository method

repositoryMock.Setup(temp => temp.MethodName(It.Any<ParameterType>()))

.Returns(return\_value);

//create service instance with mocked repository

var service = newServiceClass(repositoryMock.Object);



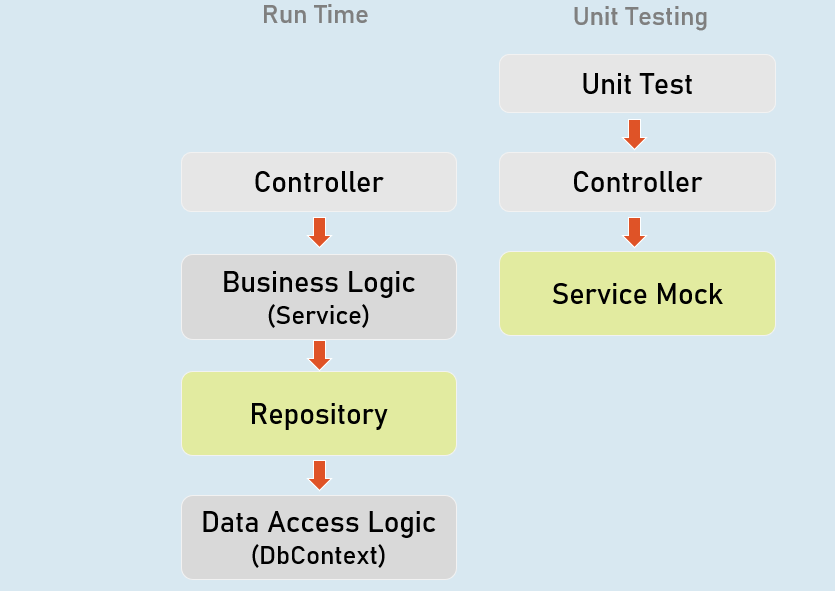
**Mock<IPersonsRepository>**

Used to mock the methods of IPersonsRepository.

**IPersonsRepository**

Represents the mocked object that was created by Mock<T>.

Unit Testing the Controller



**Unit Testing the Controller:**

//Arrange

ControllerName controller = new ControllerName();

//Act

IActionResult result = controller.ActionMethod();

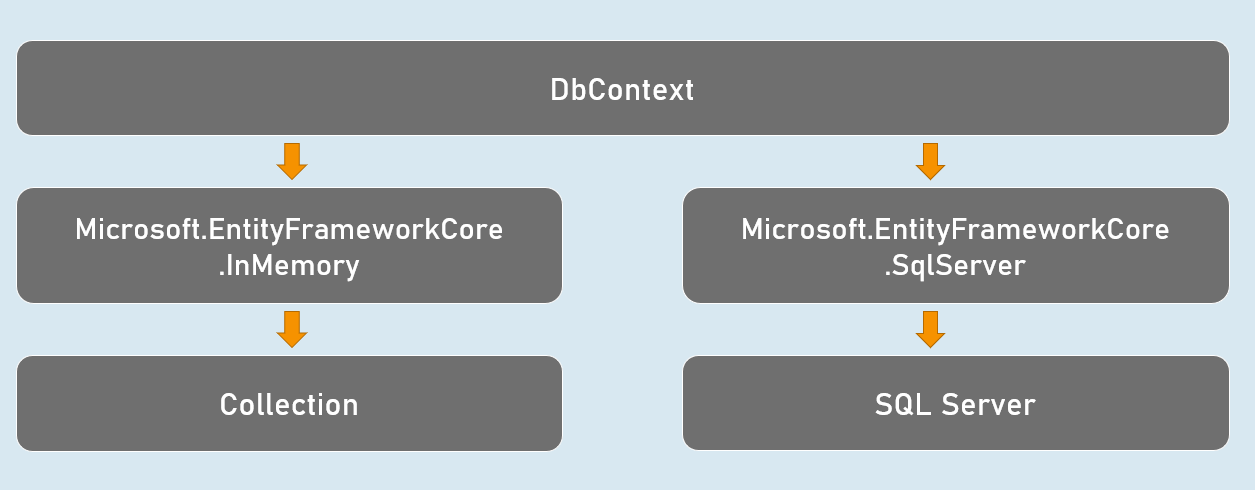
//Assert

result.Should().BeAssignableTo<ActionResultType>(); //checking type of action result

result.ViewData.Model.Should().BeAssignableTo<ExpectedType>(); //checking type of model

result.ViewData.Model.Should().Be(expectedValue); //you can also use any other assertion

EFCore In-Memory Provider



Install-Package Microsoft.EntityFrameworkCore.InMemory

**Using In-memory provider:**

var dbContextOptions =

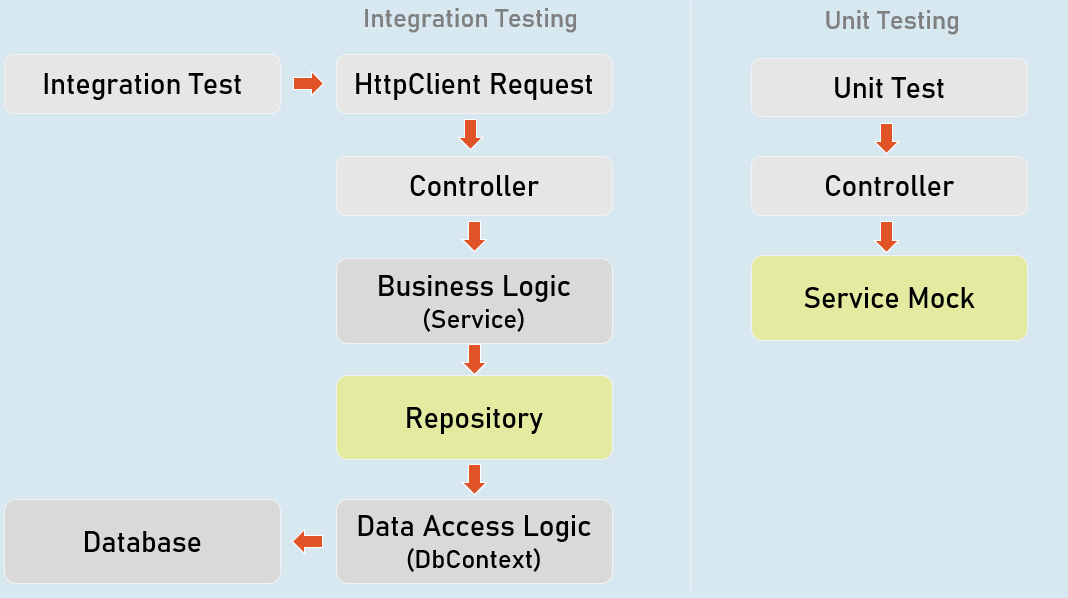
new DbContextOptionsBuilder<DbContextClassName>()

.UseInMemoryDatabase("database\_name");

.Options;

var dbContext = newDbContextClassName(dbContextOptions);

Integration Test



//Create factory

WebApplicationFactory factory = new WebApplicationFactory();

//Create client

HttpClient client = factory.CreateClient();

//Send request client

HttpResponseMessage response = await client.GetAsync("url");

//Assert

result.Should().BeSuccessful(); //Response status code should be 200 to 299