Module "Design & architecture"

Submodule "Design patterns and architecture patterns"

Part 2

UA Resource Development Unit

2021

AGENDA

1 MVC (MVP, MVVM)

2 Multilayered and onion architectures

MVC (MVP, MVVM)

MVC (MVP, MVVM)

- Model-View-Controller (MVC)
- Model-View-Presenter (MVP)
- Model-View-View Model (MVVC)

MVC (MVP, MVVM): What Do We Want to Achieve?

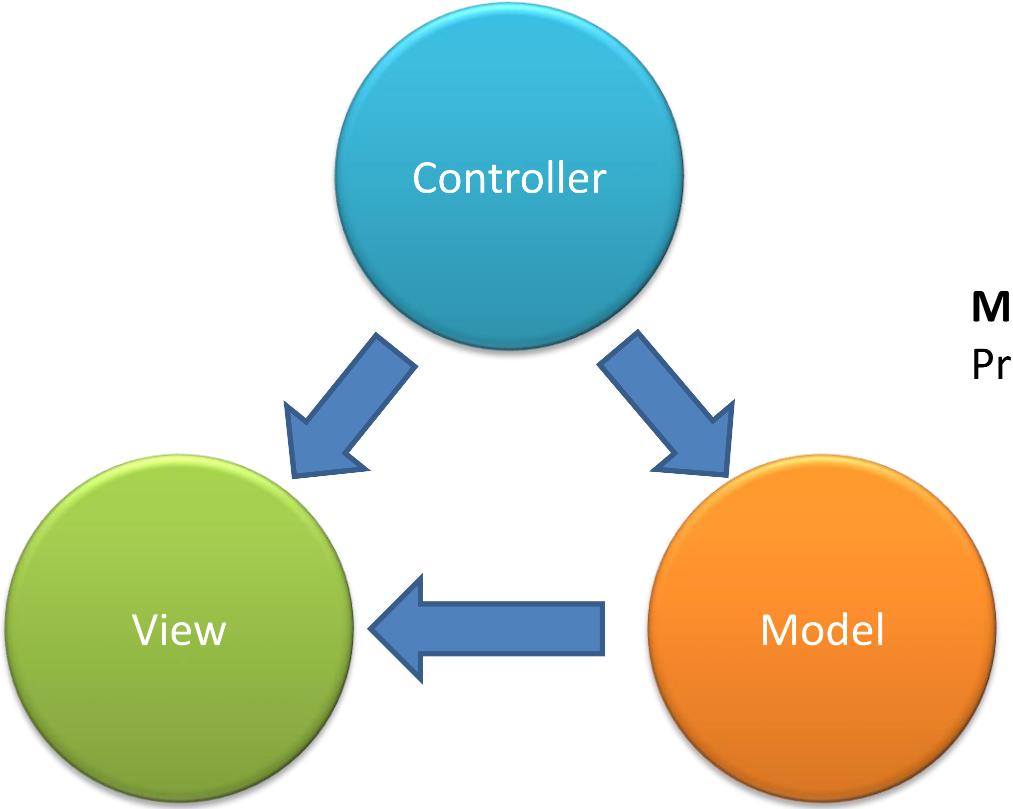
- Scalability
- Maintainability
- Reliability

- Separation of Concerns
- Code Reusability
- Testability

MVC

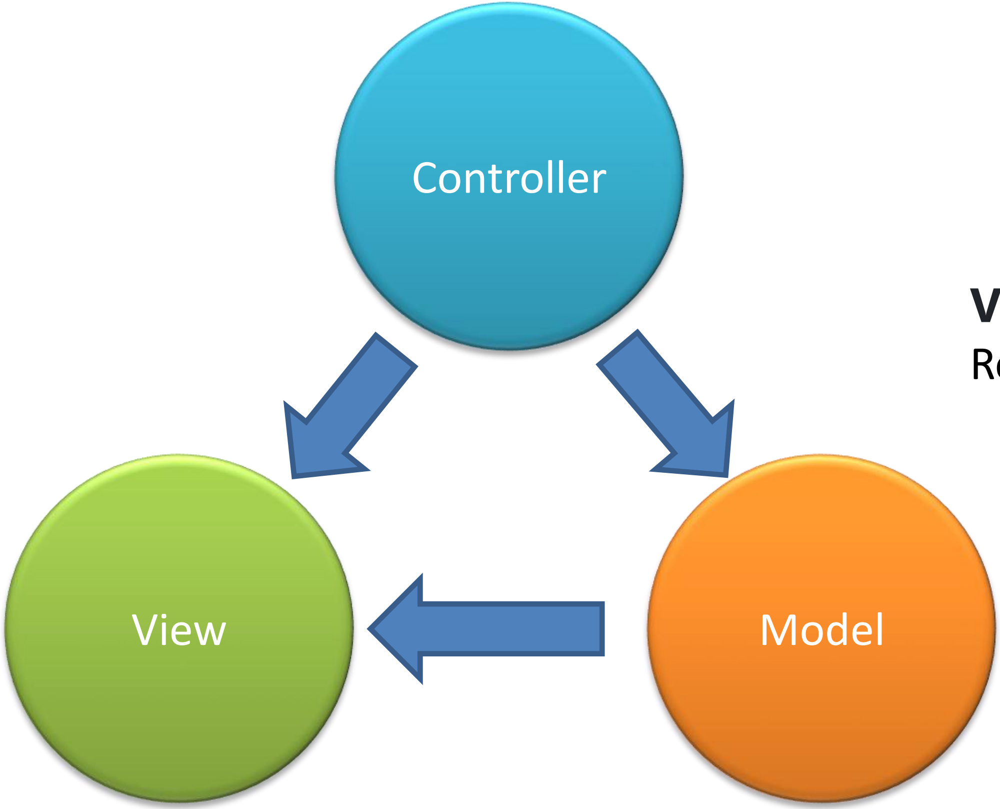
Model View Controller (MVC)

MVC: Model



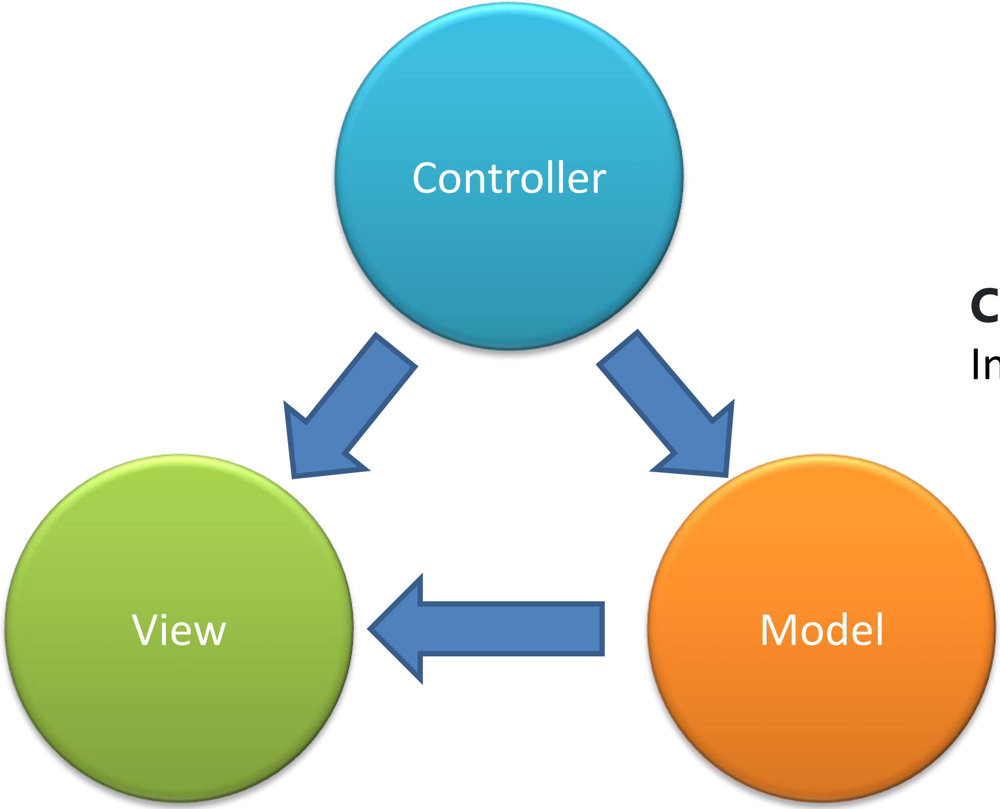
Model
Provide data and associated logic to the View

MVC: View



View
Render the Model to the View

MVC: Controller



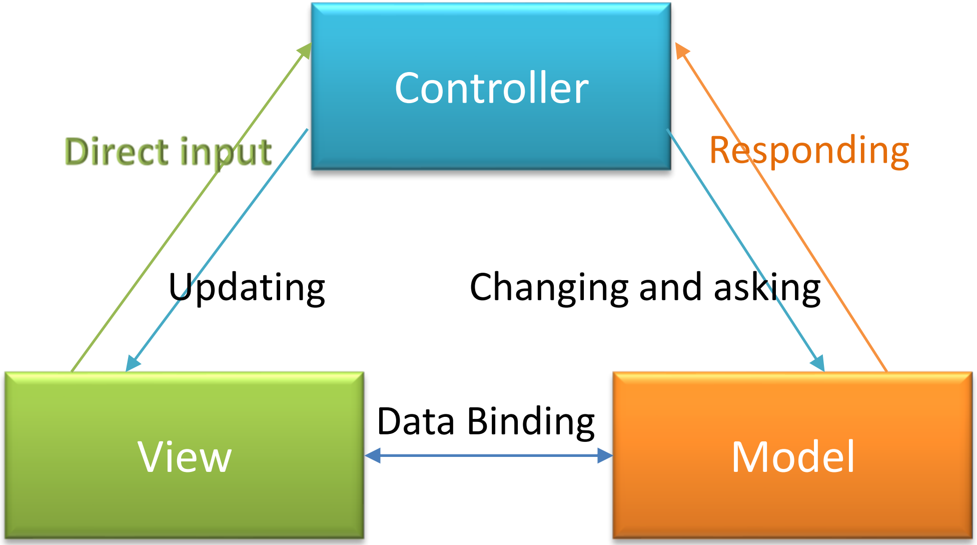
Controller
Interacts with Model and View

MVC

Our MVC has two main variants:

- supervising controller
- passive view

MVC: Supervising Controller



MVC: Supervising Controller- Summary

Characteristic element: View bound with model

Idea: Separation between input and output

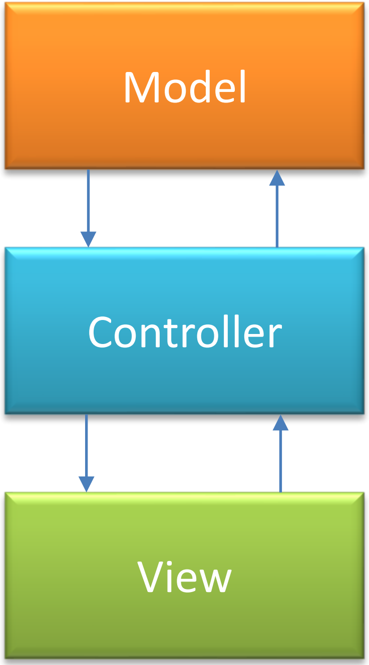
Pros and Cons

+ Less code

- **Hard to unit-test**
- **Low encapsulation**
- **Weak concerns separation**

Good for small projects and demos. Not really scalable.

MVC: Passive View



Characteristic element: Stateless View, fully managed by Controller

Idea: Separation between business and presentation logic

Business logic – business rules that application is implementing. Haw application acts. Implemented in controller.

Presentation logic - how application looks. Implemented in view layer.

MVC: Massive View Controller

We should not treat an Activity as a view. We should treat it as a presentation layer and we should extract our controller as a separate class.
A solution to making view controllers smaller is by splitting views or defining subviews with their own controllers. Writing the MVC pattern this way is easy to separate, and it is also easy to split.

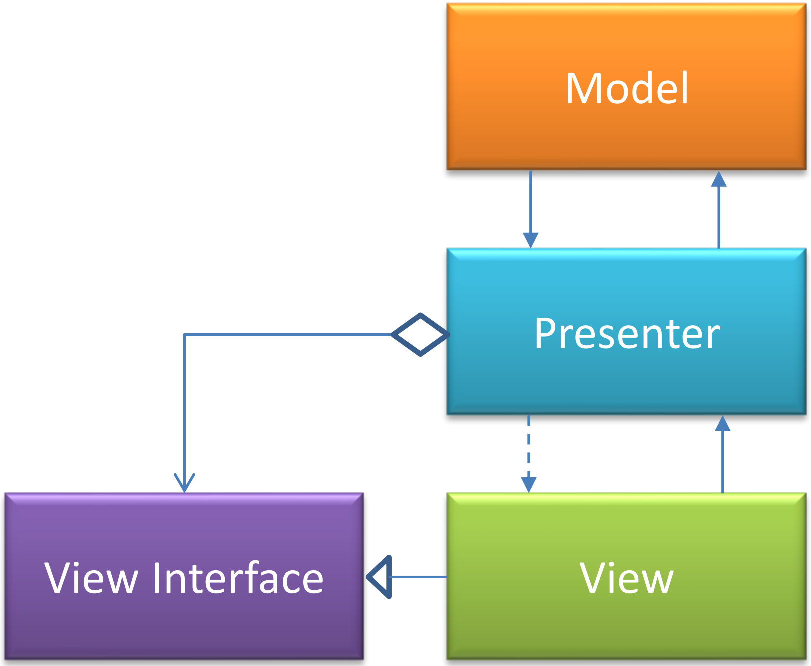
But, there are issues here with my implementation.

- Mixing presentation and business logic.
- Doing this makes it more difficult to test.

MVP

Model View Presenter (MVP)

Model View Presenter (MVP)



Characteristic element: View is hidden behind interface

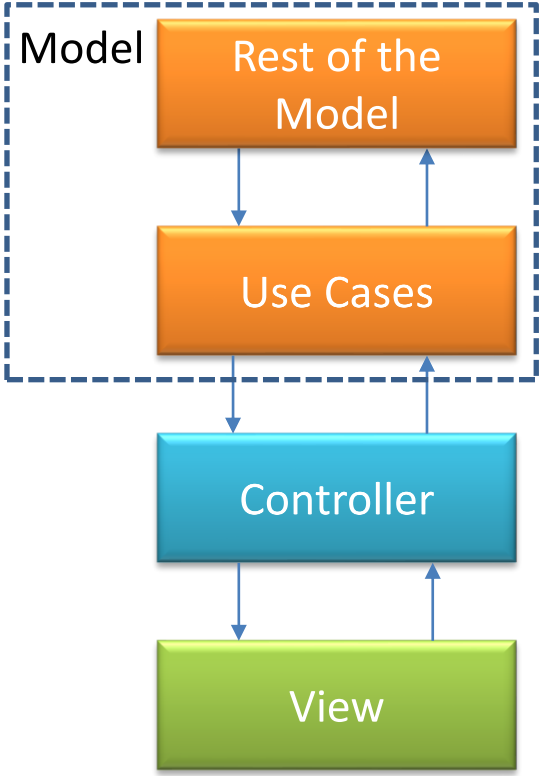
Idea: Make Presenter independent from View

Pros and Cons

+ High testability of Presenter
+ Higher Presenter reusability
+ Presenter can be used in common modules

- Need to create and maintain interface for views
- Additional boilerplate

Use cases



Change: Extract business logic into separate classes clalled use cases

Idea: Apply single responsibility principle to business logic rules

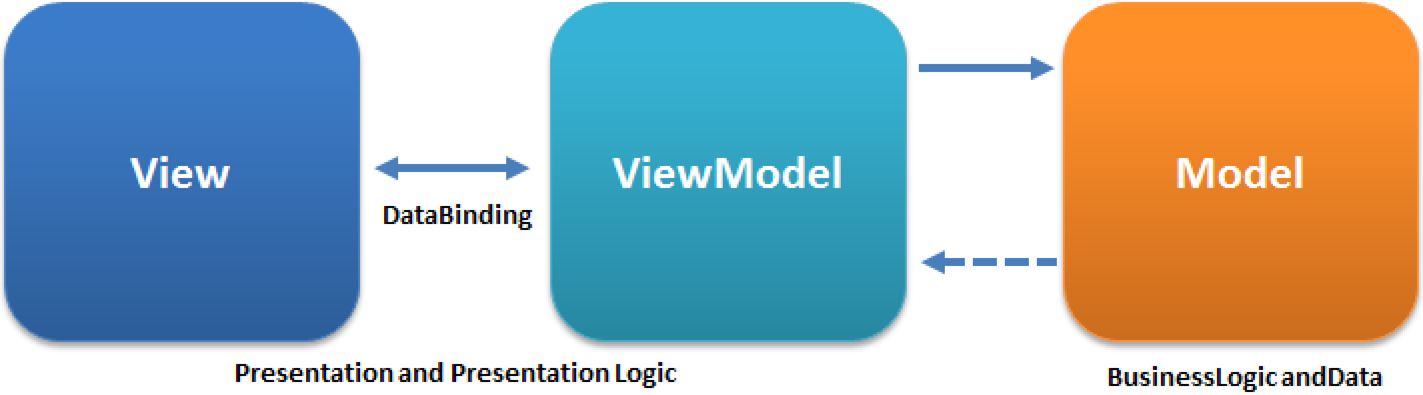
Model-View-ViewModel (MVVM)

Model-View-ViewModel (MVVM)

Model-View-ViewModel (MVVM)

Model–view–viewmodel (MVVM) is a software architectural pattern that facilitates the separation of the development of the graphical user interface (the view) – be it via a markup language or GUI code – from the development of the business logic or back-end logic (the model) so that the view is not dependent on any specific model platform. The view model of MVVM is a value converter, meaning the view model is responsible for exposing (converting) the data objects from the model in such a way that objects are easily managed and presented. In this respect, the view model is more model than view, and handles most if not all of the view's display logic. The view model may implement a mediator pattern, organizing access to the back-end logic around the set of use cases supported by the view.

Model-View-ViewModel (MVVM)



Multilayered and onion architectures

What's the difference between “Layers” and “Tiers”?

In software engineering, **multitier architecture** (often referred to as **n-tier architecture**) or **multilayered** architecture is a client–server architecture in which presentation, application processing and data management functions are physically separated.

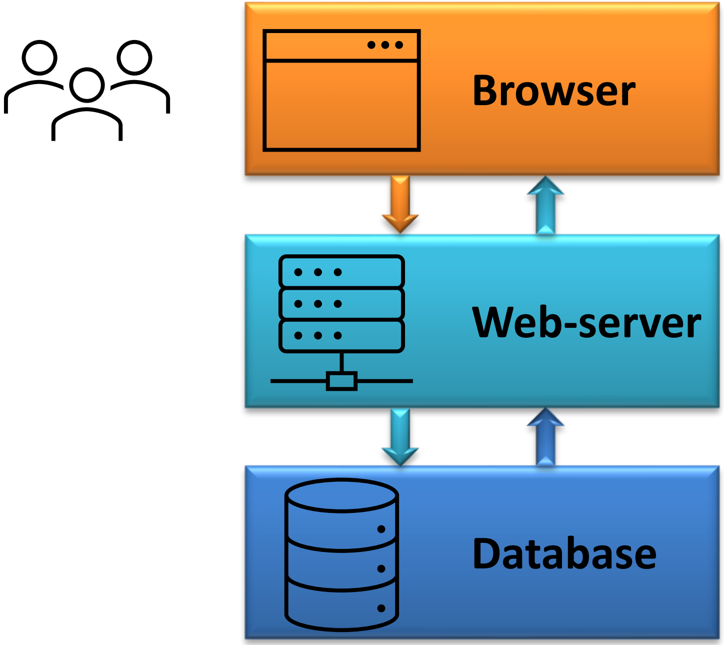
Physics components

N-Tier ≠ **N-Layer**

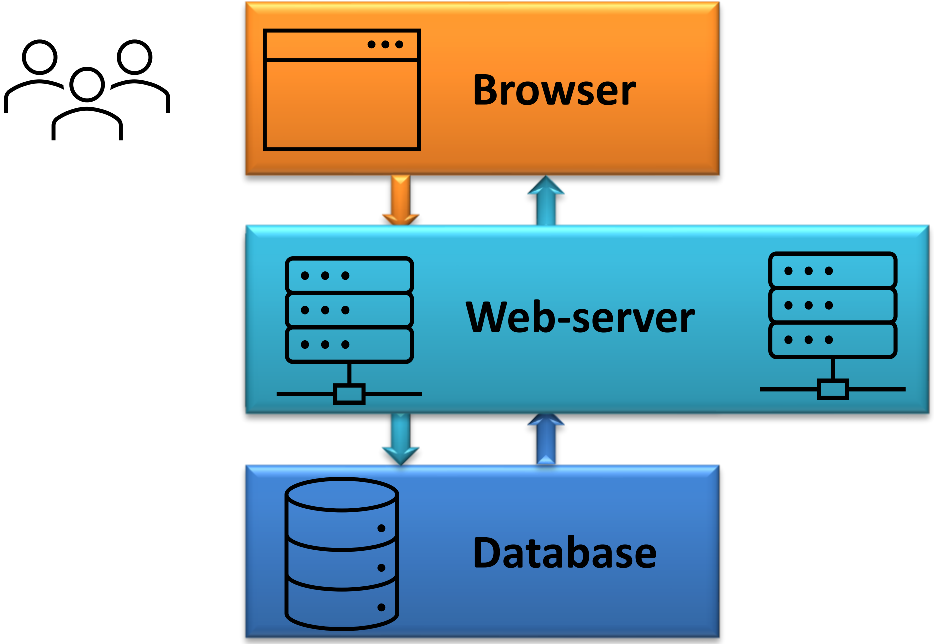
Logics components

N-Tier and N-Layer are entirely different concepts. People generally use this term during the design of the application architecture. **N-Tier** refers to the actual n system components of your application. On the other hand, **N-Layers** refer to the internal architecture of your component.

N-Tier Architecture



N-Tier Architecture



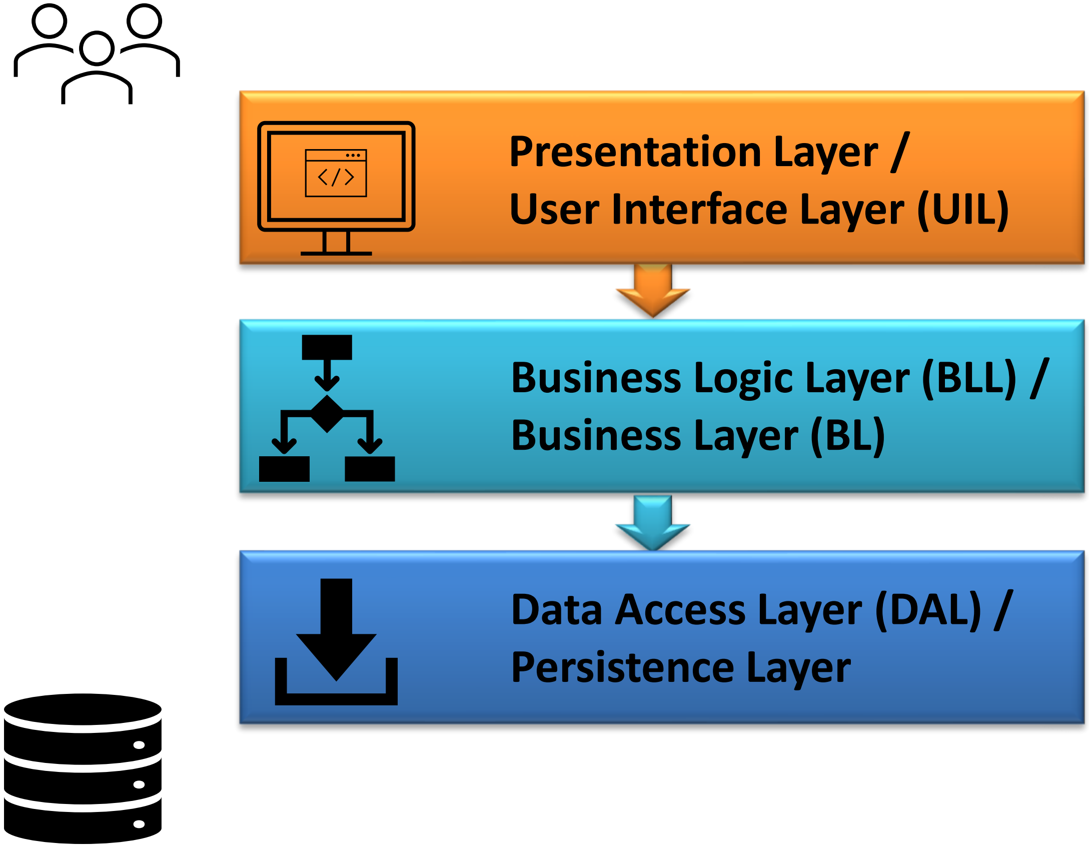
HTTP protocol

connectionString="Server=.\SQLEXPRESS;Database=ProductDB;
user=sa;password=12345"

Multilayered (N-Layer)

Multilayered (N-Layer)

Multilayered and onion architectures

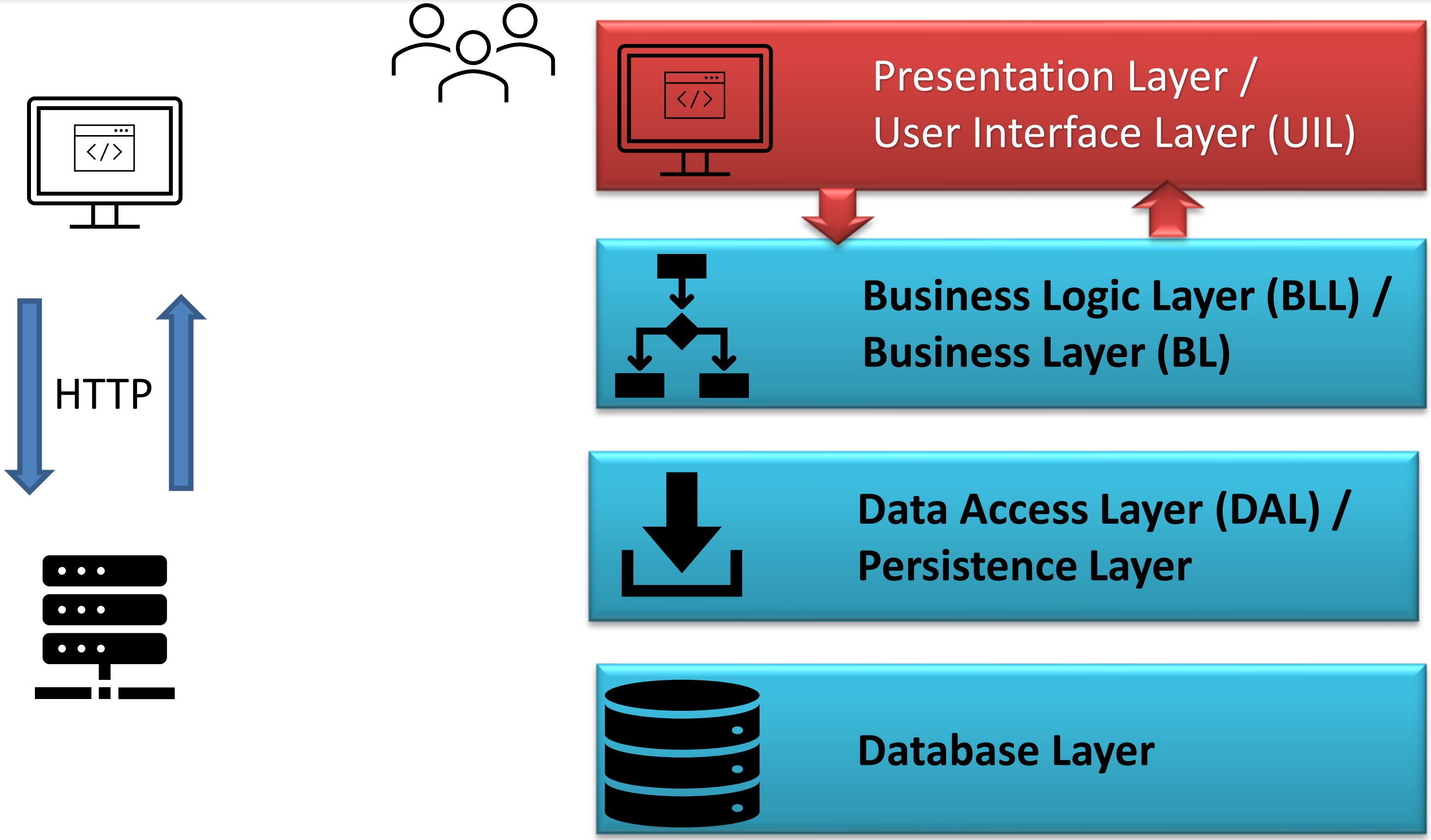


Presentation Layer /
User Interface Layer (UIL)

Business Logic Layer (BLL) /
Business Layer (BL)

Data Access Layer (DAL) /
Persistence Layer

Multilayered architectures: Presentation Layer



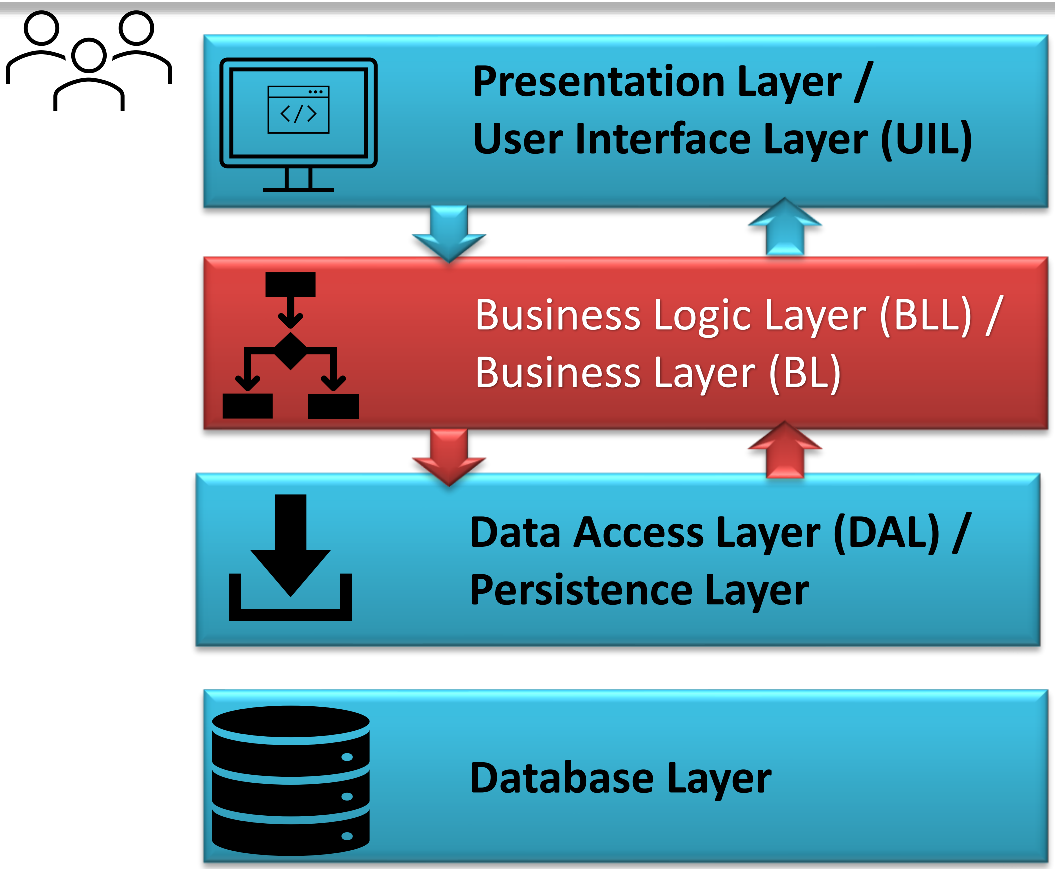
Presentation Layer / User Interface Layer (UIL)

Business Logic Layer (BLL) / Business Layer (BL)

Data Access Layer (DAL) / Persistence Layer

Database Layer

Multilayered architectures: Business Logic Layer (BLL)



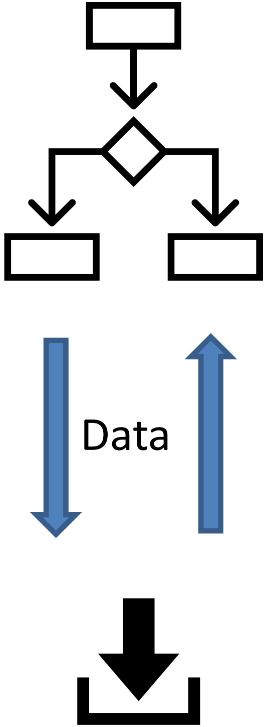
Presentation Layer /
User Interface Layer (UIL)

Business Logic Layer (BLL) /
Business Layer (BL)

Data Access Layer (DAL) /
Persistence Layer

Database Layer

Multilayered architectures: Data Access Layer (DAL)

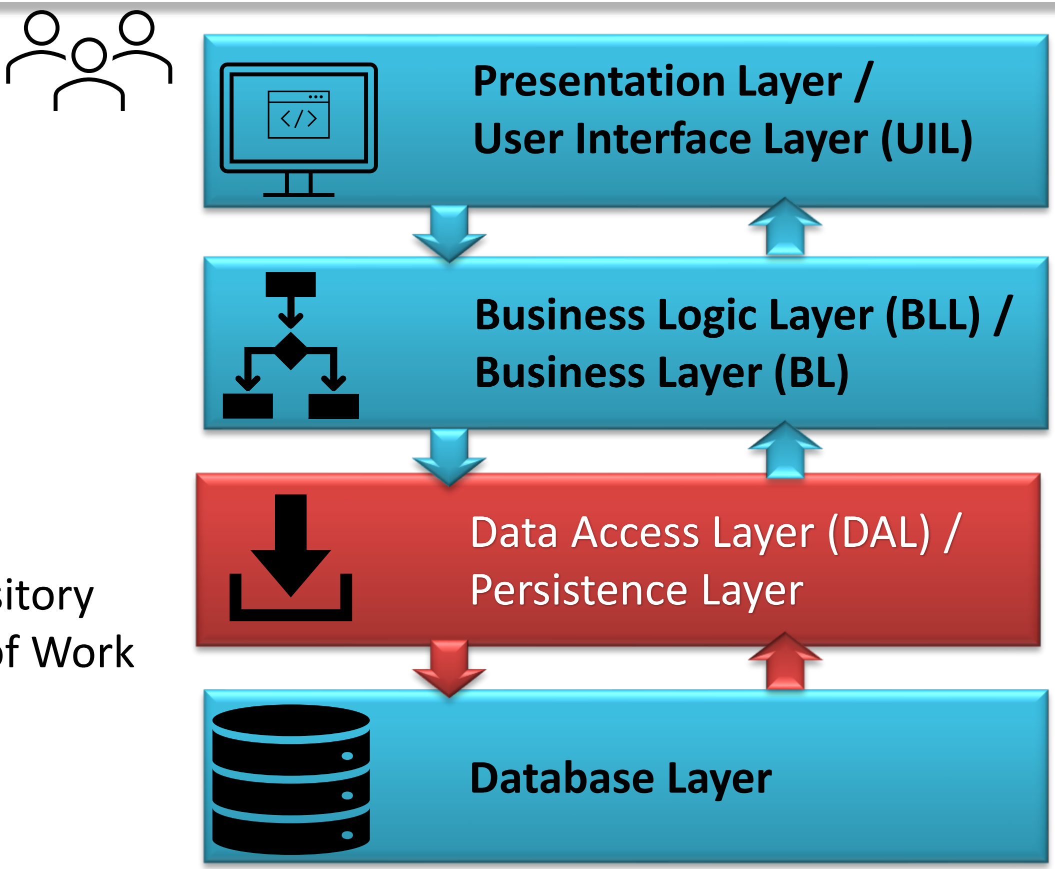


SQL

ORM

Repository

Unit of Work



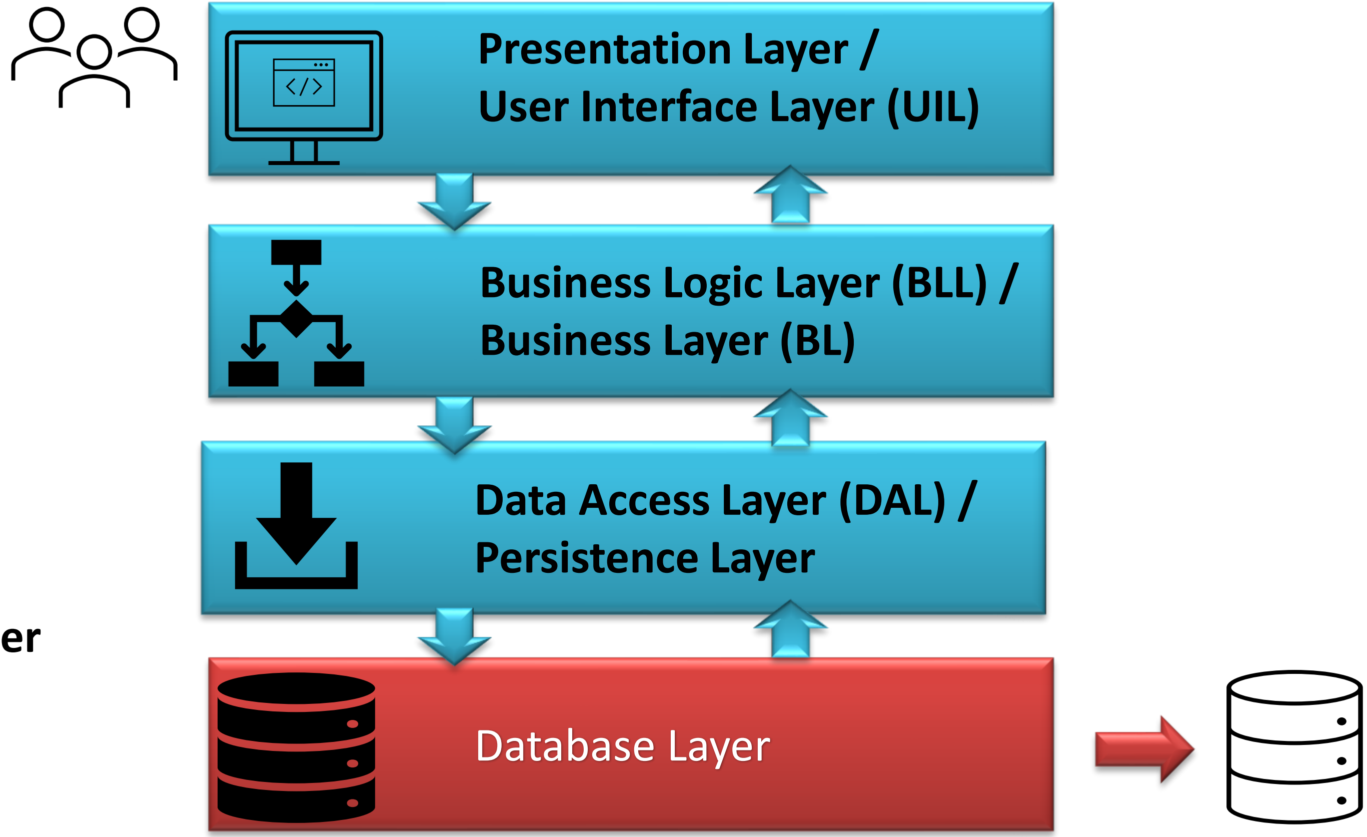
Presentation Layer /
User Interface Layer (UIL)

Business Logic Layer (BLL) /
Business Layer (BL)

Data Access Layer (DAL) /
Persistence Layer

Database Layer

Multilayered architectures: Database Layer



Presentation Layer / User Interface Layer (UIL)

Business Logic Layer (BLL) / Business Layer (BL)

Data Access Layer (DAL) / Persistence Layer

Database Layer

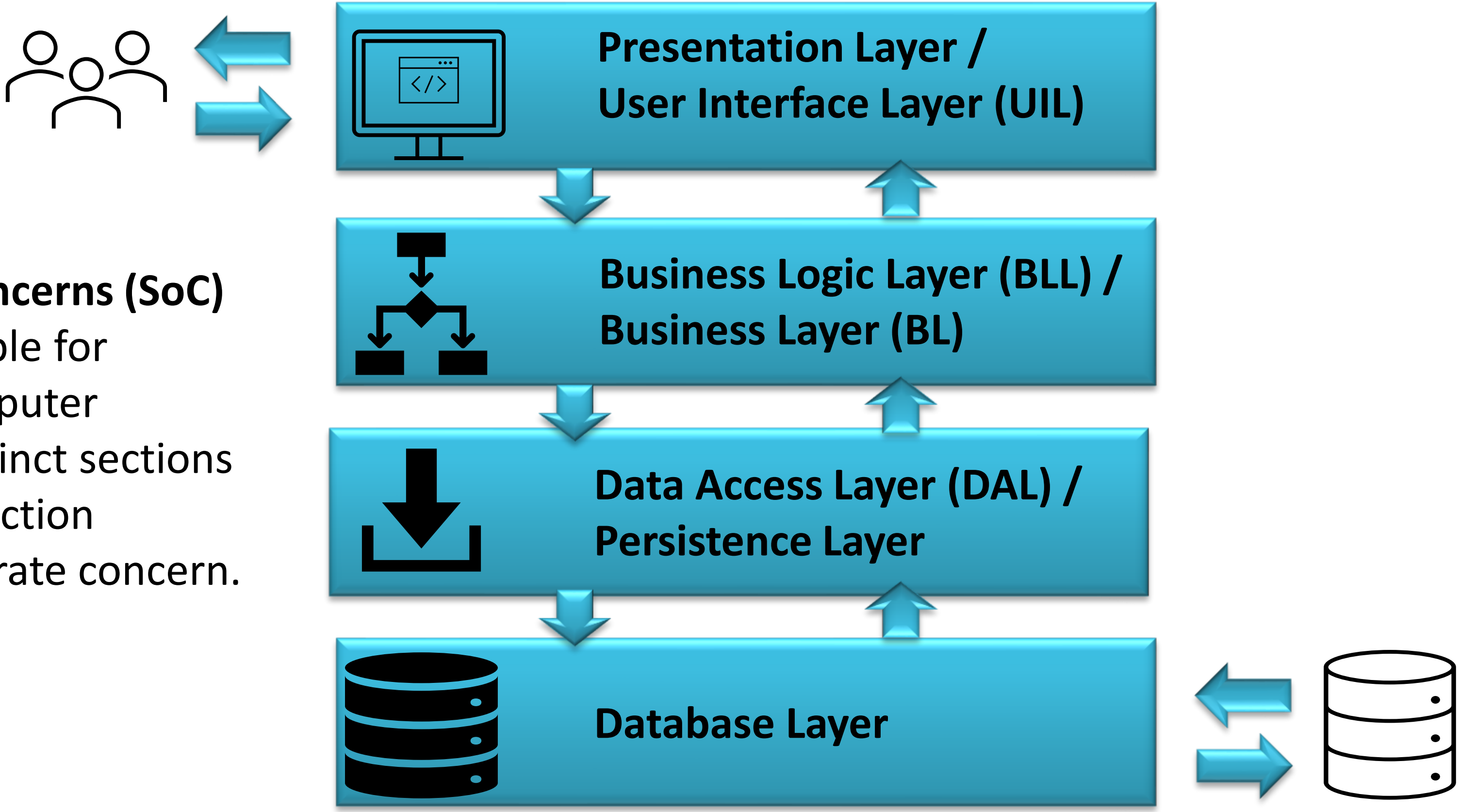
ODBC

ADO.Net driver

Oracle Data Provider

Separation of Concerns Principle

Separation of concerns (SoC) is a design principle for separating a computer program into distinct sections such that each section addresses a separate concern.



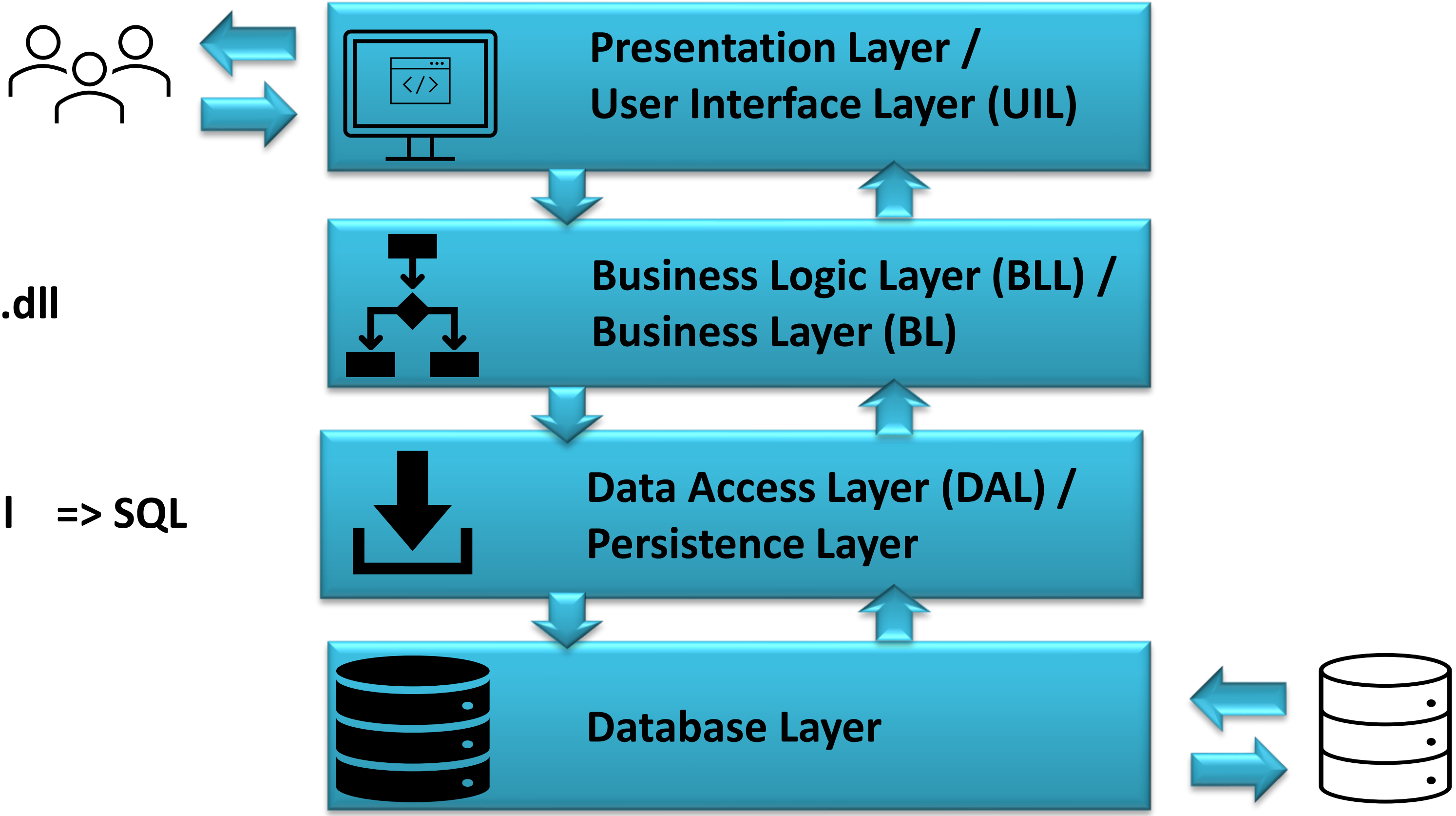
Presentation Layer / User Interface Layer (UIL)
Business Logic Layer (BLL) / Business Layer (BL)
Data Access Layer (DAL) / Persistence Layer
Database Layer

Multilayered architectures: Closed Layers

ASP.MVC

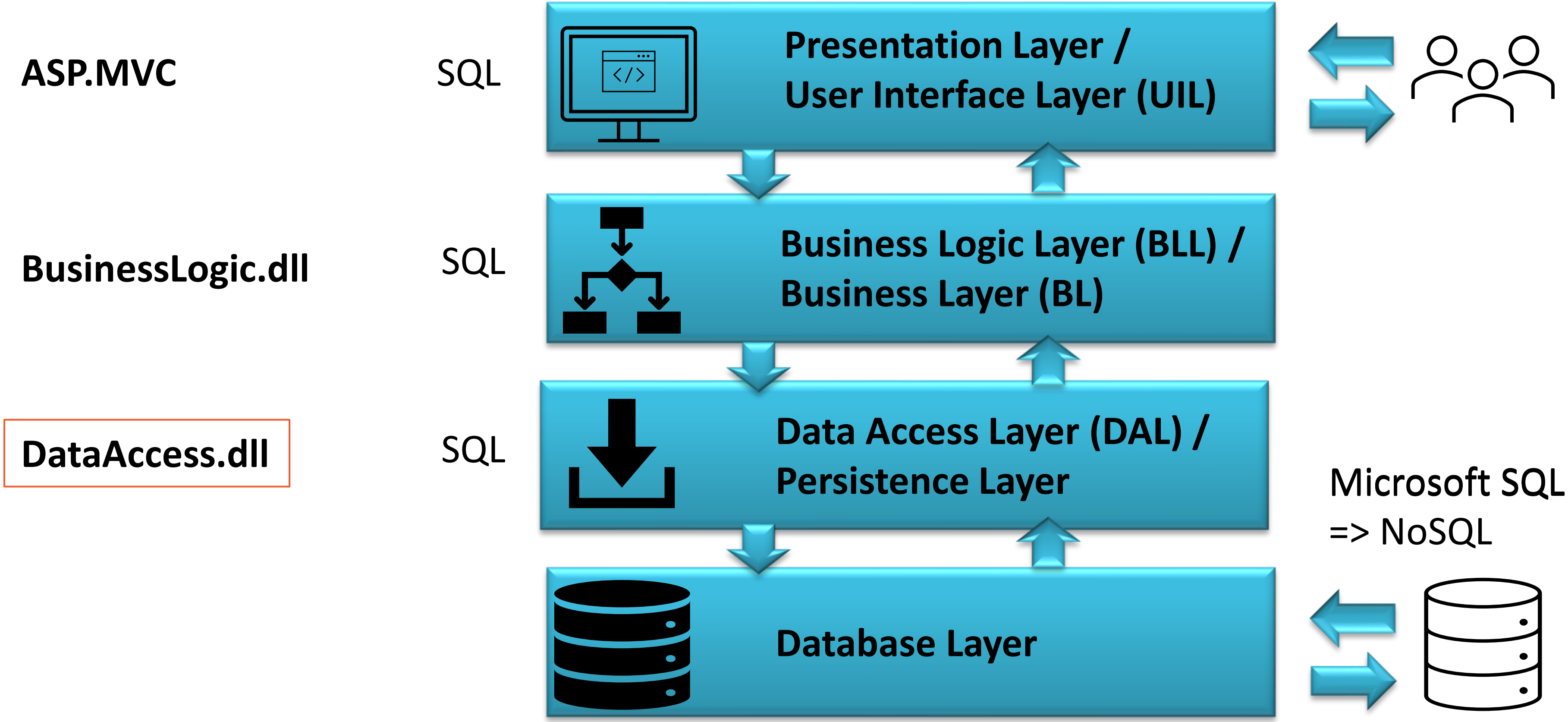
BusinessLogic.dll

DataAccess.dll => SQL



Presentation Layer /
User Interface Layer (UIL)
Business Logic Layer (BLL) /
Business Layer (BL)
Data Access Layer (DAL) /
Persistence Layer
Database Layer

Multilayered architectures: Closed Layers



ASP.MVC

SQL

Presentation Layer / User Interface Layer (UIL)

BusinessLogic.dll

SQL

Business Logic Layer (BLL) / Business Layer (BL)

DataAccess.dll

SQL

Data Access Layer (DAL) / Persistence Layer

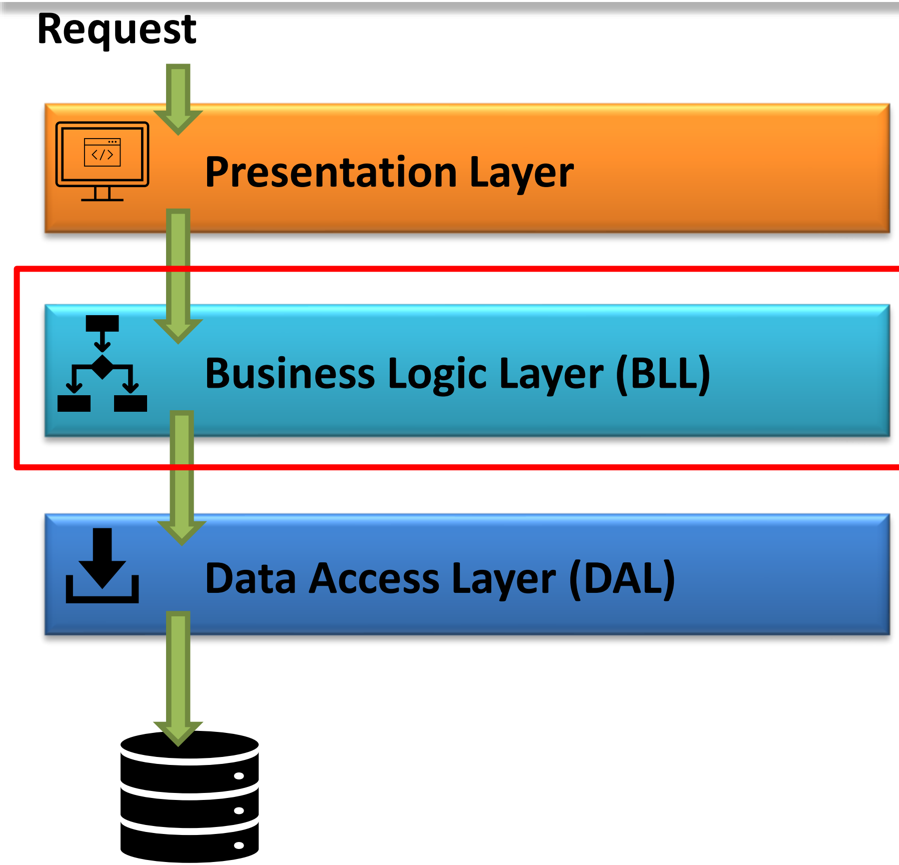
Microsoft SQL
=> NoSQL

Database Layer

Multilayered architectures: Layer isolation Principle

The layers of isolation concept means that changes made in one layer of the architecture generally don’t impact or affect components in other layers: the change is isolated to the components within that layer, and possibly another associated layer (such as a persistence layer containing SQL).

“Architecture sinkhole” anti-pattern



Request

Presentation Layer

Business Logic Layer (BLL)

Data Access Layer (DAL)

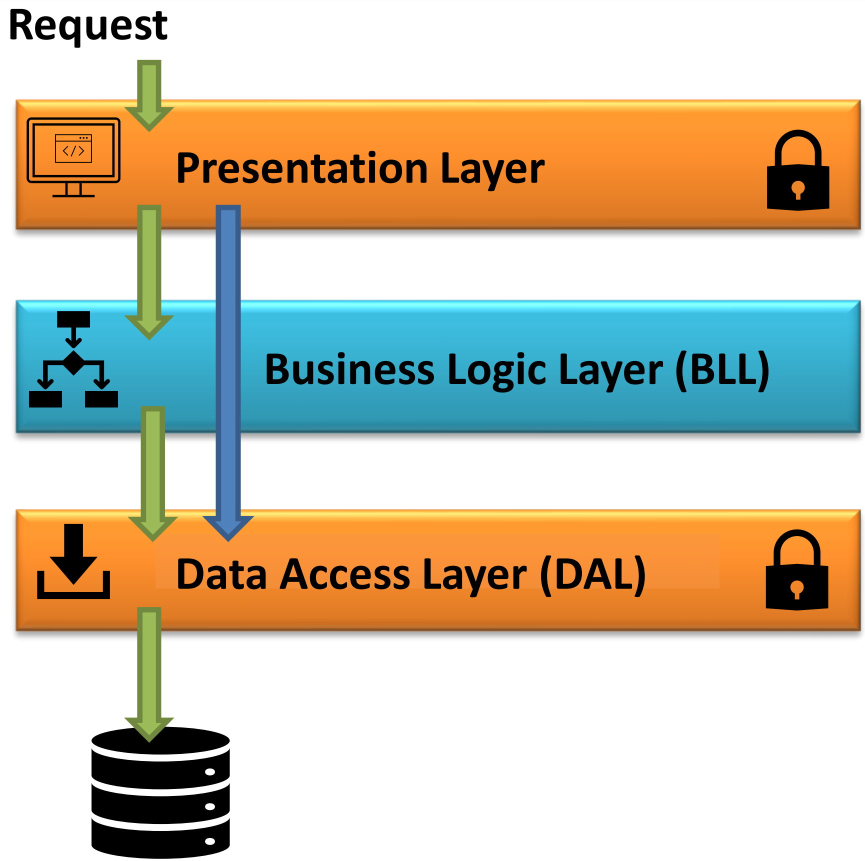
```
public HttpResponseMessage GetAboutPage()
{
    var aboutPage = _pageService.GetPage("about");
    return CreateResponse(HttpStatusCode.OK, aboutPage);
}
```

```
//inside pageService
public string GetPage(string pageName)
{
    _pagesRepository.GetPage(pageName);
}
```

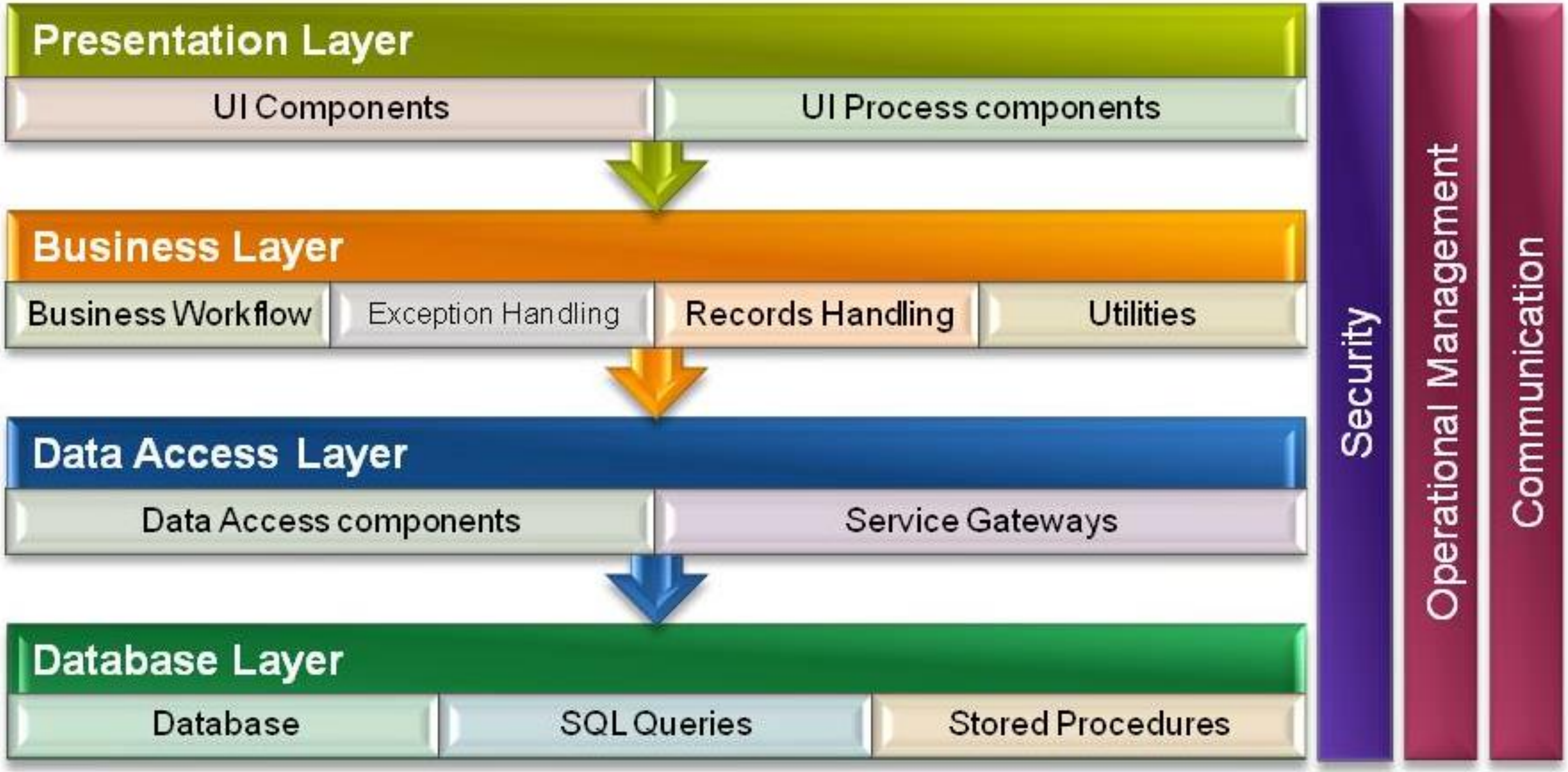
<20%

```
//inside pagesRepository
public string GetPage(string pageName)
{
    return _db.Pages.SingleOrDefaault(pageName);
}
```

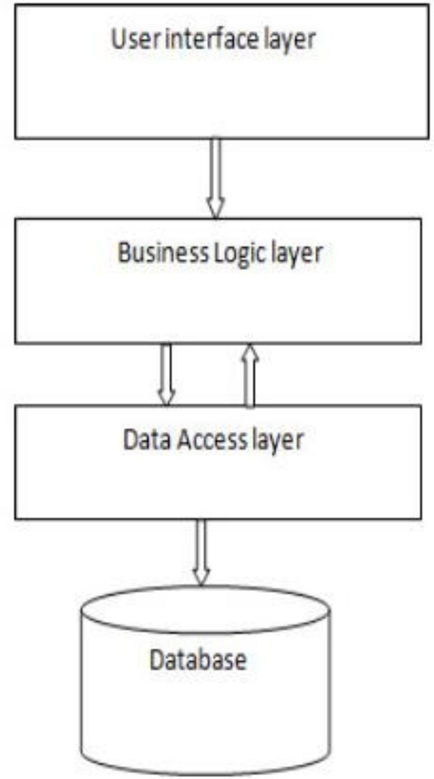
Multilayered architectures: Open Layers



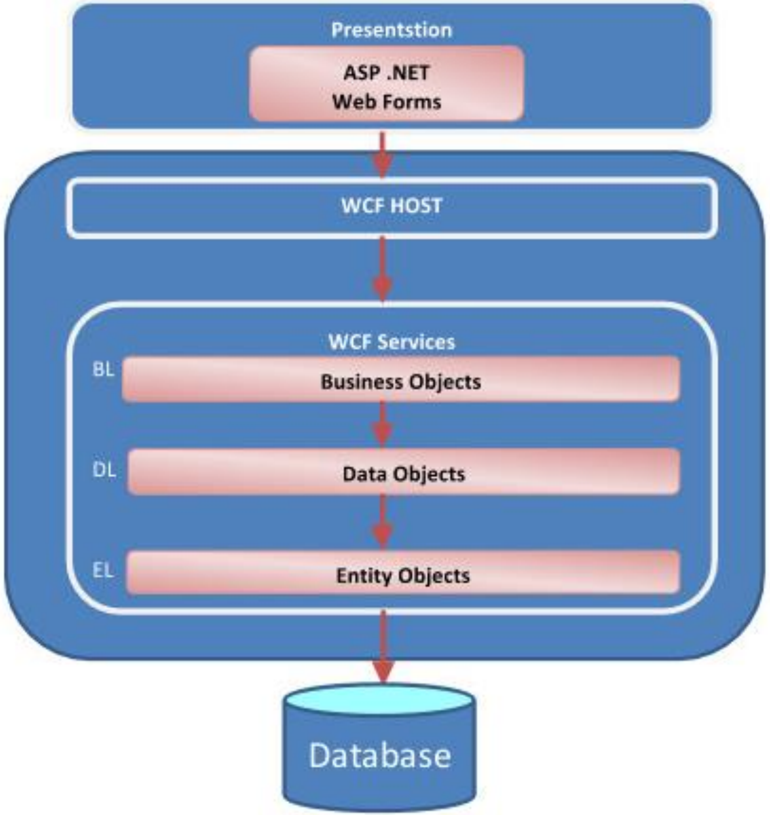
Multilayered Architecture



Multilayered architectures



Layered Architecture



Tiered Architecture

Pros and Cons of Tier and Layerd Architecture

- **Tiers** indicate a physical separation of components, which may mean different assemblies on the same server or multiple servers. **Layers** refers to a logical separation of components, such as having distinct namespaces and classes for the Database Access Layer (DAL), Business Logic Layer (BLL) and User Interface Layer (UIL).

- **Tiers** could be on different machines, so they communicate by Value only – as serialized objects. **Multi-layered** design is suitable for small to mid-size projects only.

- **Tiers** could be on different machines, so they communicate by Value only – as serialized objects. **Layer** communicates with each other either by Value or by Reference.

- **Tiered** Architecture has all advantages of **Layered** Architecture + scalability as application will be deployed in different machines so load will be shared among the **tiers** and scalability will increase. **Layered** Architecture will improve readability and reusability

Onion architecture

Onion architecture

Onion architecture

The Onion Architecture term was coined by Jeffrey Palermo in 2008. This architecture provides a better way to build applications for better testability, maintainability, and dependability on the infrastructures like databases and services.

The idea of the Onion Architecture is to place the Domain and Services Layers at the center of your application. And externalize the Presentation and Infrastructure.

Onion architecture

There are two types of coupling:

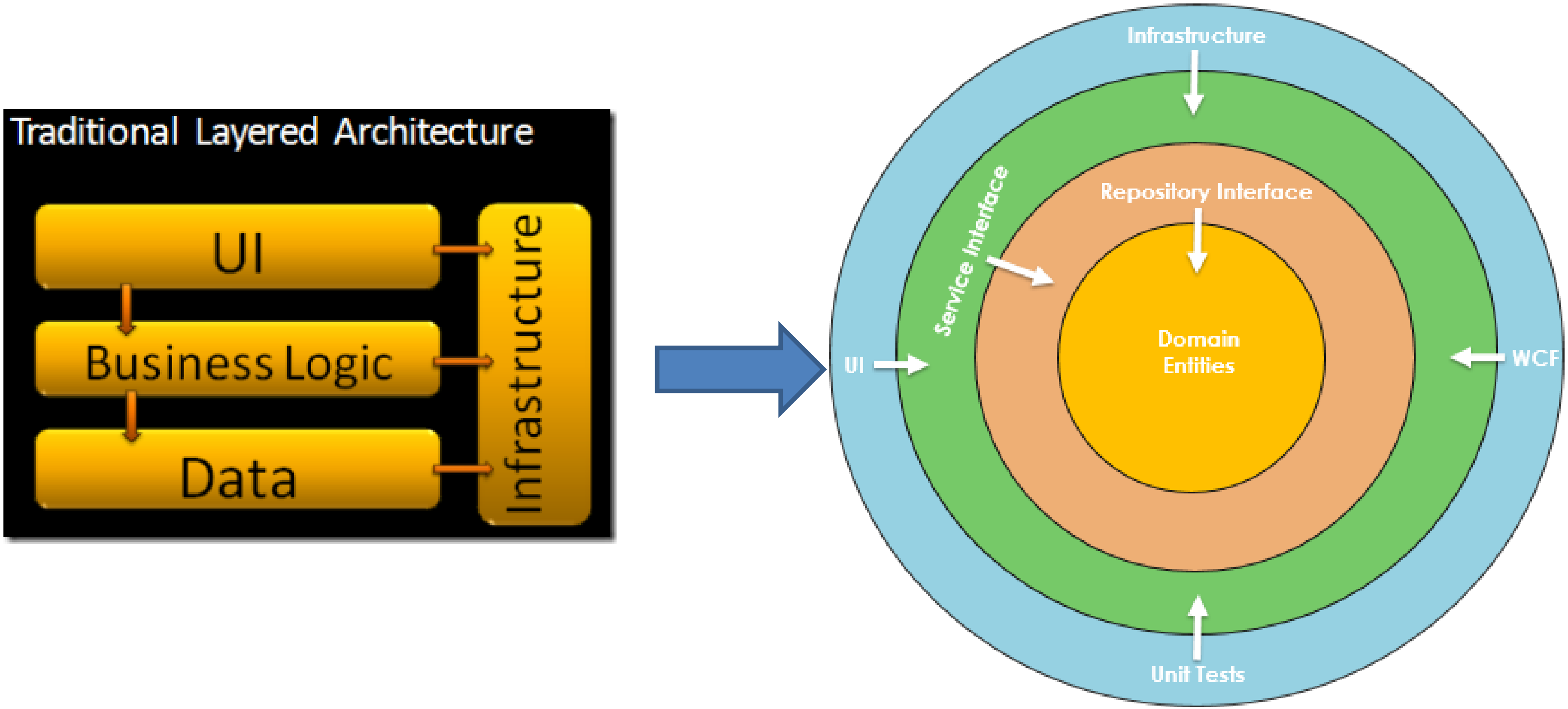
- **Tight Coupling**

When a class is dependent on a concrete dependency, it is said to be tightly coupled to that class. A tightly coupled object is dependent on another object; that means changing one object in a tightly coupled application, often requires changes to a number of other objects. It is not difficult when an application is small but in an enterprise-level application, it is too difficult to make the changes.

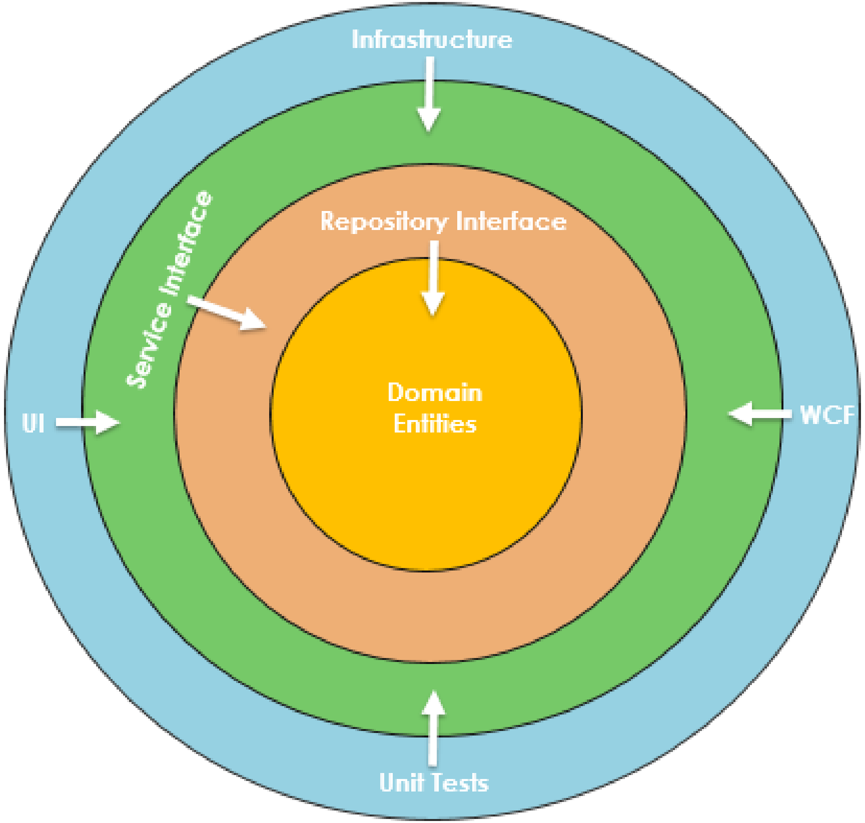
- **Loose Coupling**

It means two objects are independent and an object can use another object without being dependent on it. It is a design goal that seeks to reduce the interdependencies among components of a system with the goal of reducing the risk that changes in one component will require changes in any other component.

Onion architecture



Onion architecture



Onion architecture: Layers

Domain Entities Layer

It is the center part of the architecture. It holds all application domain objects. If an application is developed with the ORM entity framework then this layer holds POCO classes (Code First) or Edmx (Database First) with entities. These domain entities don't have any dependencies.

Repository Layer

The layer is intended to create an abstraction layer between the Domain entities layer and the Business Logic layer of an application. It is a data access pattern that prompts a more loosely coupled approach to data access. We create a generic repository, which queries the data source for the data, maps the data from the data source to a business entity, and persists changes in the business entity to the data source.

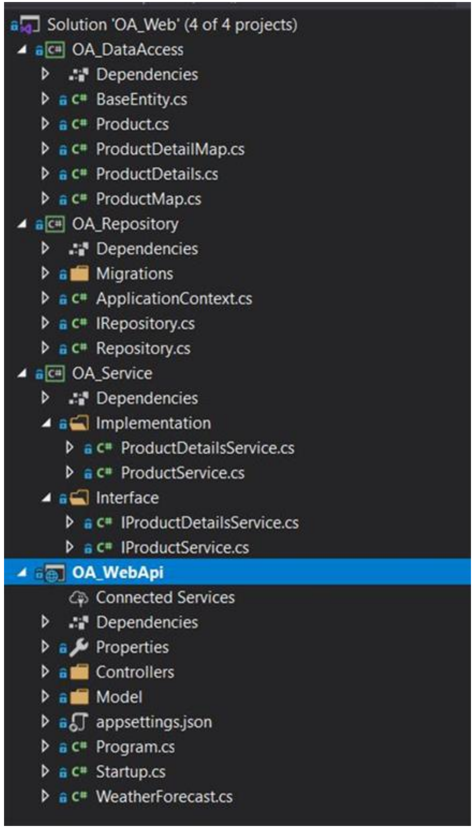
Service Layer

The layer holds interfaces which are used to communicate between the UI layer and repository layer. It holds business logic for an entity so it’s called the business logic layer as well.

UI Layer

It’s the most external layer. It could be the web application, Web API, or Unit Test project. This layer has an implementation of the Dependency Inversion Principle so that the application builds a loosely coupled application. It communicates to the internal layer via interfaces.

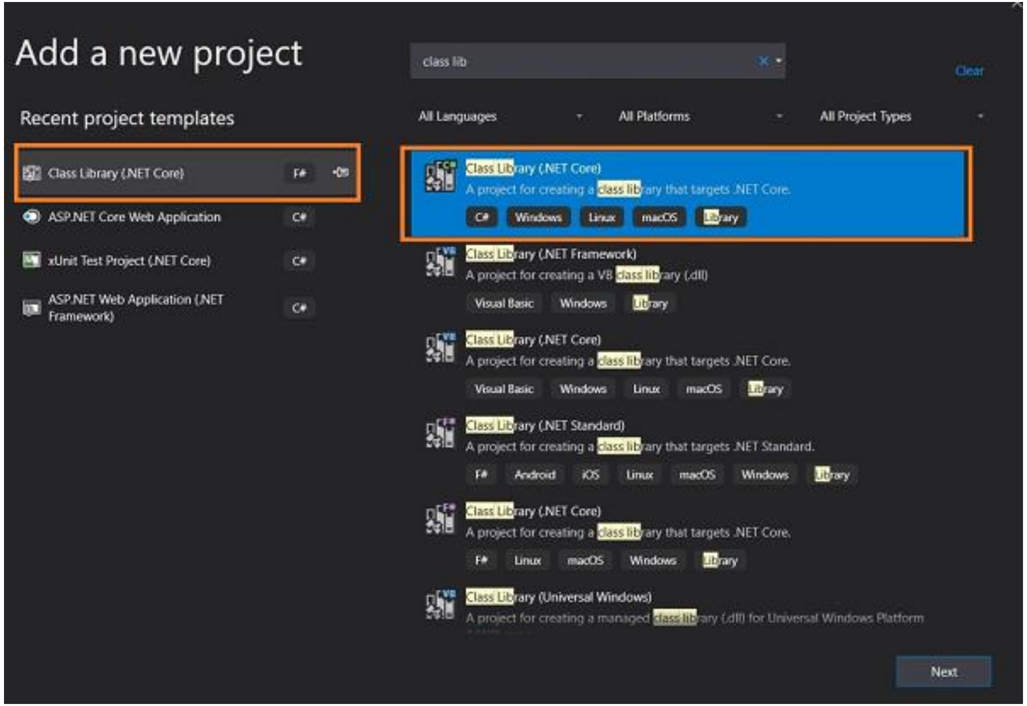
DEMO: Onion architecture: Project Structure



```
Solution 'OA_Web' (4 of 4 projects)
  OA_DataAccess
    Dependencies
    BaseEntity.cs
    Product.cs
    ProductDetailMap.cs
    ProductDetails.cs
    ProductMap.cs
  OA_Repository
    Dependencies
    Migrations
    ApplicationContext.cs
    IRepository.cs
    Repository.cs
  OA_Service
    Dependencies
    Implementation
      ProductDetailsService.cs
      ProductService.cs
    Interface
      IProductDetailsService.cs
      IProductService.cs
  OA_WebApi
    Connected Services
    Dependencies
    Properties
    Controllers
    Model
    appsettings.json
    Program.cs
    Startup.cs
    WeatherForecast.cs
```

DEMO: Onion architecture - Step 1

Step 1 - Create the project for Domain/Data Access Layer - OA_DataAccess



Add a new project

class lib

Recent project templates

Clear
All Languages
All Platforms
All Project Types
Class Library (.NET Core) F#
ASP.NET Core Web Application C#
xUnit Test Project (.NET Core) C#
ASP.NET Web Application (.NET Framework) C#

Class Library (.NET Core)
A project for creating a class library that targets .NET Core.
C# Windows Linux macOS Library

Class Library (.NET Framework)
A project for creating a VB class library (.dll)
Visual Basic Windows Library

Class Library (.NET Core)
A project for creating a class library that targets .NET Core.
Visual Basic Windows Linux macOS Library

Class Library (.NET Standard)
A project for creating a class library that targets .NET Standard.
F# Android iOS Linux macOS Windows Library

Class Library (.NET Core)
A project for creating a class library that targets .NET Core.
F# Linux macOS Windows Library

Class Library (Universal Windows)
A project for creating a managed class library (.dll) for Universal Windows Platform

DEMO: Onion architecture - Step 1

BaseEntity.cs

```
public class BaseEntity
{
   public int ProductId { get; set; }
}
```

Product.cs

```
public class Product : BaseEntity
{
   public string ProductName { get; set; }
   public virtual ProductDetails ProductDetails { get; set; }
}
```

ProductMap.cs

```
public class ProductMap
{
public ProductMap(EntityTypeBuilder<Product> entityBuilder)
{
   entityBuilder.HasKey(p => p.ProductId);
   entityBuilder.HasOne(p => p.ProductDetails).WithOne(p => p.Product
   ).HasForeignKey<ProductDetails>(x => x.ProductId);
}
```

Make sure Microsoft.EntityFrameworkCore package is added using NuGet package manager.

DEMO: Onion architecture - Step 1

ProductDetail.cs

```
public class ProductDetails : BaseEntity
{
   public int StockAvailable { get; set; }
   public decimal Price { get; set; }
   public virtual Product Product{get;set;}
}
```

ProductDetailMap.cs

```
public class ProductDetailMap
{
   public ProductDetailMap(EntityTypeBuilder<ProductDetails> entityBuilder)
   {
      entityBuilder.HasKey(p => p.ProductId);
      entityBuilder.Property(p => p.StockAvailable).IsRequired();
      entityBuilder.Property(p => p.Price);
   }
}
```

DEMO: Onion architecture - Step 2

Step 2 - Create the project for Repository Layer - OA_Repository

ApplicationContext.cs

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

   protected override void OnModelCreating(ModelBuilder modelBuilder)
   {
      base.OnModelCreating(modelBuilder);
      new ProductMap(modelBuilder.Entity<Product>());
      new ProductDetailMap(modelBuilder.Entity<ProductDetails>());
   }
}
```

Make sure Microsoft.EntityFrameworkCore package is added using NuGet package manager.

DEMO: Onion architecture - Step 2

IRepository.cs

```
public interface IRepository<T> where T : BaseEntity
{
   T Get(int id);
   IEnumerable<T> GetAll();
}
```

Repository.cs

```
public class Repository<T> : IRepository<T> where T : BaseEntity
{
    private readonly ApplicationContext context;
    private readonly DbSet<T> entities;
    public Repository(ApplicationContext context)
    {
        this.context = context;
        entities = context.Set<T>();
    }
    public IEnumerable<T> GetAll()
    {
        return entities.AsEnumerable();
    }
    public T Get(int id)
    {
        return entities.SingleOrDefault(p =>p.ProductId  == id);
    }
}
```

DEMO: Onion architecture - Step 3

Step 3 - Create the project for Service Layer - OA_Service

IProductService.cs

```
public interface IProductService
{
   IEnumerable<OA_DataAccess.Product> GetProduct();
   OA_DataAccess.Product GetProduct(int id);
}
```

IProductDetailService.cs

```
public interface IProductDetailsService
{
   OA_DataAccess.ProductDetails GetProductDetail(int id);
}
```

DEMO: Onion architecture - Step 3

ProductService.cs

```
public class ProductService: IProductService
{
   private IRepository<Product> productRepository;
   private IRepository<ProductDetails> productDetailRepository;

   public ProductService(IRepository<Product> productRepository, IRepository<ProductDetails>
          productDetailRepository)
   {
      this.productRepository = productRepository;
      this.productDetailRepository = productDetailRepository;
   }

   public IEnumerable<Product> GetProduct()
   {
      return productRepository.GetAll();
   }

   public Product GetProduct(int id)
   {
      return productRepository.Get(id);
   }
}
```

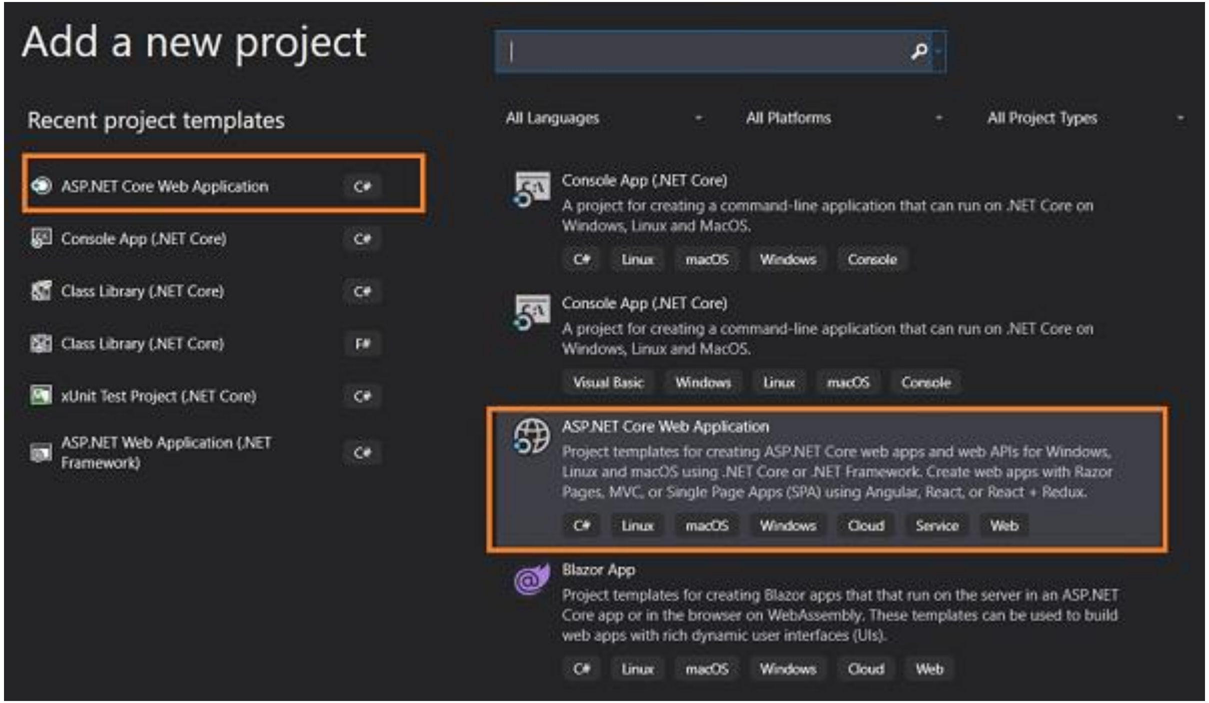
DEMO: Onion architecture - Step 3

ProductDetailService.cs

```
public class ProductDetailsService: IProductDetailsService
{
   private IRepository<ProductDetails> productDetailsRepository;

   public ProductDetailsService(IRepository<ProductDetails> productDetailsRepository)
   {
      this.productDetailsRepository = productDetailsRepository;
   }

   public ProductDetails GetProductDetail(int id)
   {
      return productDetailsRepository.Get(id);
   }
}
```

DEMO: Onion architecture - Step 4

Step 4 - Create ASP.NET WEB API application for UI/Presentation Layer



Add a new project

Recent project templates

- ASP.NET Core Web Application — C#
- Console App (.NET Core) — C#
- Class Library (.NET Core) — C#
- Class Library (.NET Core) — F#
- xUnit Test Project (.NET Core) — C#
- ASP.NET Web Application (.NET Framework) — C#

All Languages | All Platforms | All Project Types

Console App (.NET Core)
A project for creating a command-line application that can run on .NET Core on Windows, Linux and MacOS.
C# | Linux | macOS | Windows | Console

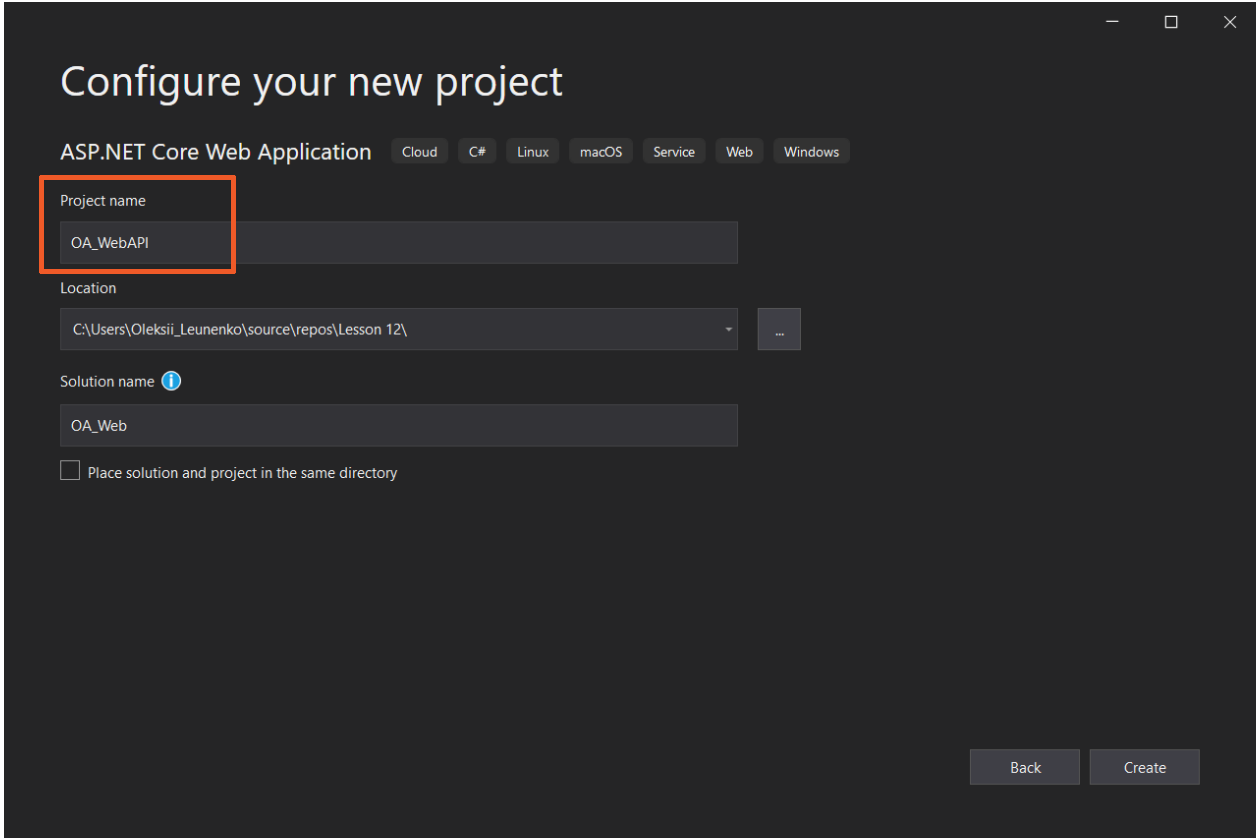
Console App (.NET Core)
A project for creating a command-line application that can run on .NET Core on Windows, Linux and MacOS.
Visual Basic | Windows | Linux | macOS | Console

ASP.NET Core Web Application
Project templates for creating ASP.NET Core web apps and web APIs for Windows, Linux and macOS using .NET Core or .NET Framework. Create web apps with Razor Pages, MVC, or Single Page Apps (SPA) using Angular, React, or React + Redux.
C# | Linux | macOS | Windows | Cloud | Service | Web

Blazor App
Project templates for creating Blazor apps that that run on the server in an ASP.NET Core app or in the browser on WebAssembly. These templates can be used to build web apps with rich dynamic user interfaces (UIs).
C# | Linux | macOS | Windows | Cloud | Web

DEMO: Onion architecture - Step 4

OA_WebAPI



Configure your new project

ASP.NET Core Web Application

Project name: OA_WebAPI

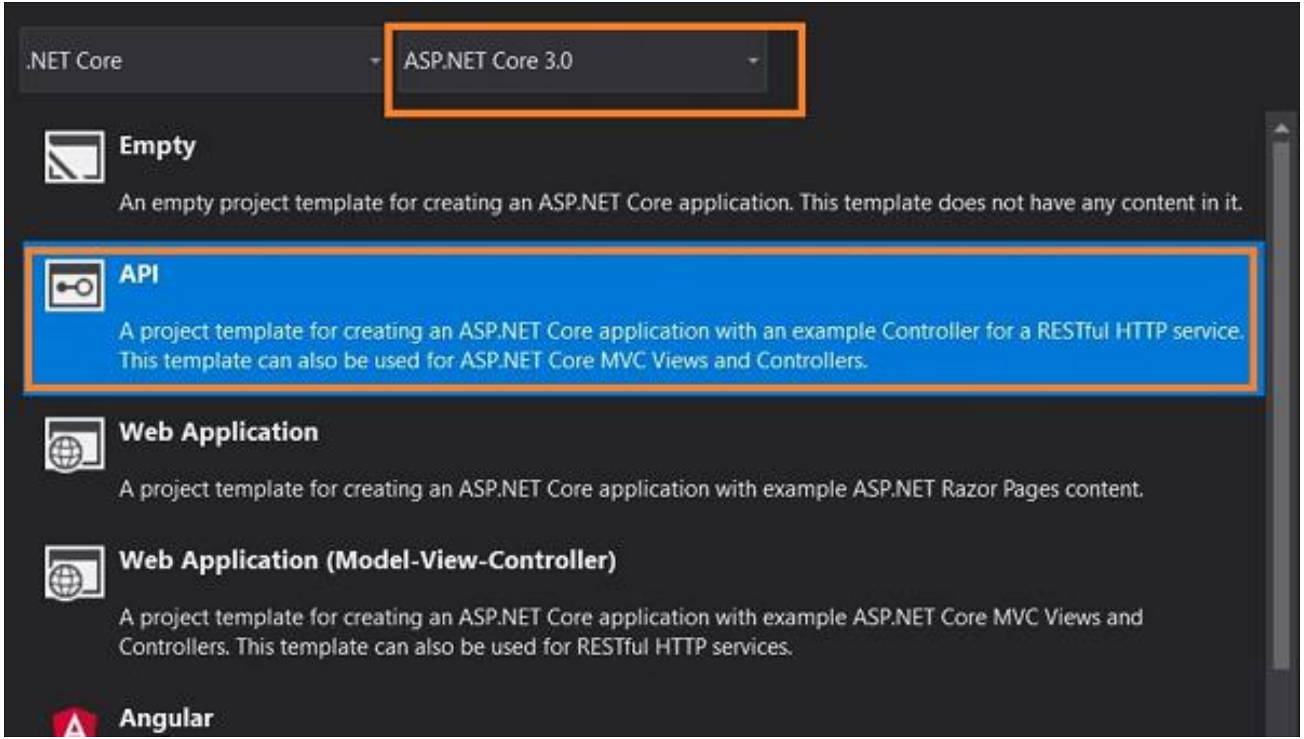
Location: C:\Users\Oleksii_Leunenko\source\repos\Lesson 12\

Solution name: OA_Web

□ Place solution and project in the same directory

DEMO: Onion architecture - Step 4

Choose the API template



Empty

An empty project template for creating an ASP.NET Core application. This template does not have any content in it.

API

A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.

Web Application

A project template for creating an ASP.NET Core application with example ASP.NET Razor Pages content.

Web Application (Model-View-Controller)

A project template for creating an ASP.NET Core application with example ASP.NET Core MVC Views and Controllers. This template can also be used for RESTful HTTP services.

Angular

DEMO: Onion architecture - Step 4

Go to Startup.cs file and add the below code under ConfigureService method.

```
services.AddDbContext<ApplicationContext>(options => options.UseSqlServer(Configuration.
        GetConnectionString("DefaultConnection")));
services.AddScoped(typeof(IRepository<>), typeof(Repository<>));
services.AddTransient<IProductService, ProductService>();
services.AddTransient<IProductDetailsService, ProductDetailsService>();
```

The Default connection string is defined in appsettings.json file

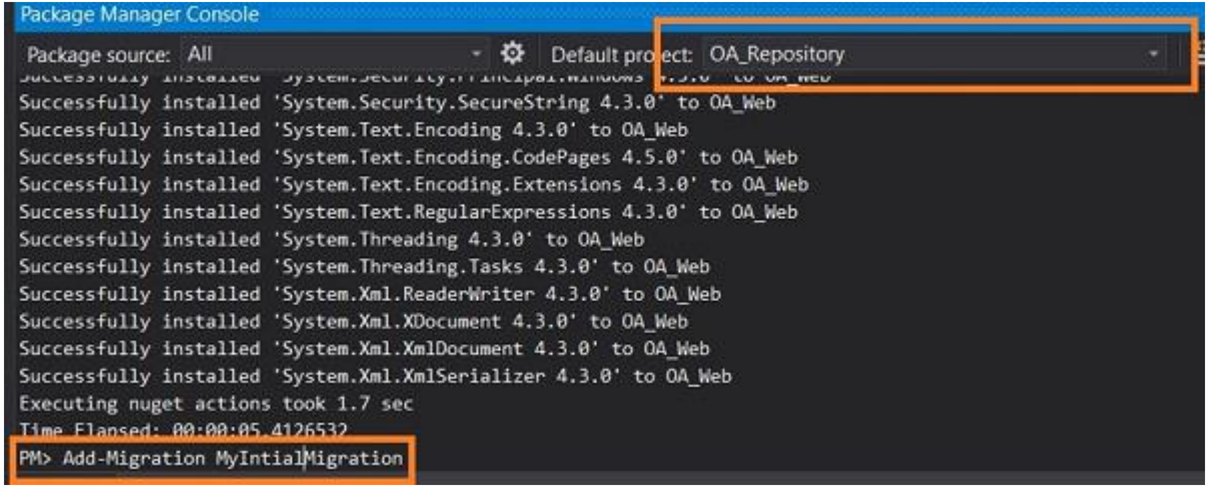
```
"ConnectionStrings": {"DefaultConnection": "Data Source=DESKTOP585QGBN;
Initial Catalog=OATestDB; User ID=sa; Password=[give your SQL instance password]"},
```

DEMO: Onion architecture - Step 4

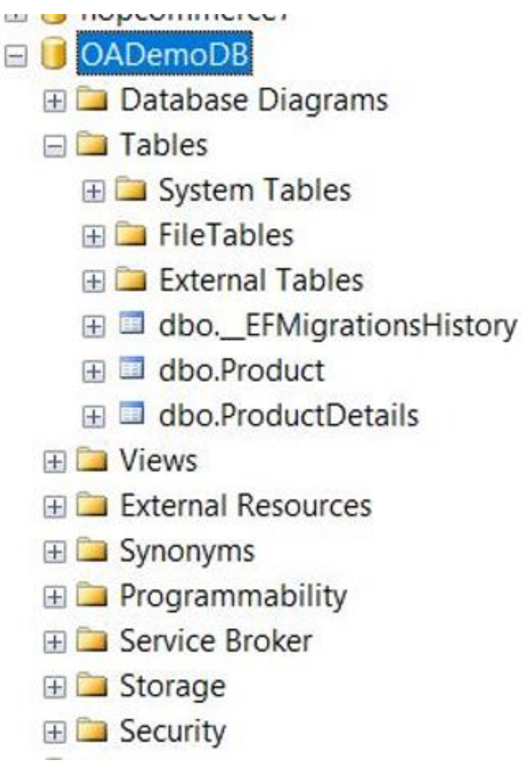
```
private readonly IProductService productService;
private readonly IProductDetailsService productDetailsService;
public ProductController(IProductService productService, IProductDetailsService productDetailsService)
{
    this.productService = productService;
    this.productDetailsService = productDetailsService;
}

[HttpGet]
public List<ProductDetails> Get()
{
    List<ProductDetails> productDetails = new List<ProductDetails>();
    var prodcutList=productService.GetProduct().ToList();
    foreach(var product in prodcutList)
    {
        var productDetailList = productDetailsService.GetProductDetail(product.ProductId);
        ProductDetails details = new ProductDetails
        {
            ProductId = product.ProductId,
            ProductName = product.ProductName,
            Price = productDetailList.Price,
            StockAvailable = productDetailList.StockAvailable,
        };
        productDetails.Add(details);
    }
    return productDetails;
}
```

Create a New Web API controller ProductController.cs

DEMO: Onion architecture - Step 4



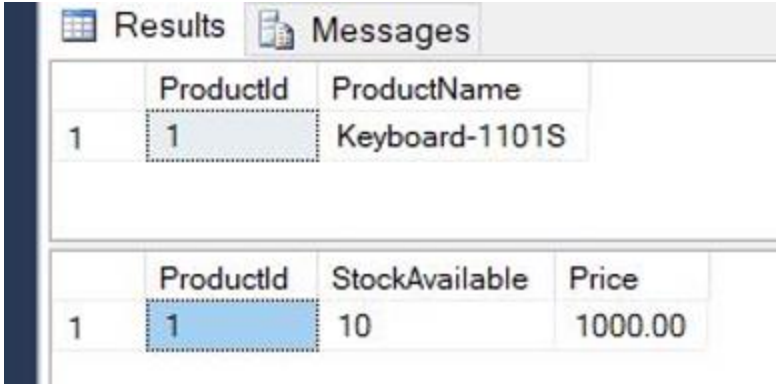
```
Successfully installed 'System.Security.Principal.Windows 4.3.0' to OA_Web
Successfully installed 'System.Security.SecureString 4.3.0' to OA_Web
Successfully installed 'System.Text.Encoding 4.3.0' to OA_Web
Successfully installed 'System.Text.Encoding.CodePages 4.5.0' to OA_Web
Successfully installed 'System.Text.Encoding.Extensions 4.3.0' to OA_Web
Successfully installed 'System.Text.RegularExpressions 4.3.0' to OA_Web
Successfully installed 'System.Threading 4.3.0' to OA_Web
Successfully installed 'System.Threading.Tasks 4.3.0' to OA_Web
Successfully installed 'System.Xml.ReaderWriter 4.3.0' to OA_Web
Successfully installed 'System.Xml.XDocument 4.3.0' to OA_Web
Successfully installed 'System.Xml.XmlDocument 4.3.0' to OA_Web
Successfully installed 'System.Xml.XmlSerializer 4.3.0' to OA_Web
Executing nuget actions took 1.7 sec
Time Elapsed: 00:00:05.4126532
PM> Add-Migration MyIntialMigration
```



```
nopcommerce7
OADemoDB
  Database Diagrams
  Tables
    System Tables
    FileTables
    External Tables
    dbo.__EFMigrationsHistory
    dbo.Product
    dbo.ProductDetails
  Views
  External Resources
  Synonyms
  Programmability
  Service Broker
  Storage
  Security
```

- Microsoft.EntityFrameworkCore
- Microsoft.EntityFrameworkCore.Relational
- Microsoft.EntityFrameworkCore.SqlServer
- Microsoft.EntityFrameworkCore.Tools

DEMO: Onion architecture - Step 4



	ProductId	ProductName
1	1	Keyboard-1101S

	ProductId	StockAvailable	Price
1	1	10	1000.00

Onion architectures: Benefits and Drawbacks

Following are the **benefits** of implementing Onion Architecture:

- Onion Architecture layers are connected through interfaces. Implantations are provided during run time.
- Application architecture is built on top of a domain model.
- All external dependency, like database access and service calls, are represented in external layers.
- No dependencies of the Internal layer with external layers.
- Couplings are towards the center.
- Flexible and sustainable and portable architecture.
- No need to create common and shared projects.
- Can be quickly tested because the application core does not depend on anything.

A few **drawbacks** of Onion Architecture as follows:

- Not easy to understand for beginners, learning curve involved. Architects mostly mess up splitting responsibilities between layers.
- Heavily used interfaces

.NET Online UA Training Course Feedback

I hope that you will find this material useful.

If you find errors or inaccuracies in this material or know how to improve it, please report on to the electronic address:

Oleksii_Leunenko@epam.com

With the note [.NET Online UA Training Course Feedback]

Thank you.

Q&A

<‹epam›>

DRIVEN CANDID CREATIVE ORIGINAL INTELLIGENT EXPERT

UA .NET Online LAB