Yuriy Fedkovych Chernivtsi National University
Faculty of Mathematics and Computer Science

O. Matvyi
H. Melnyk
V. Melnyk

Modern Client Web-Technologies

Textbook

Chernivtsi, 2025

Modern Client Web-Technologies

Published by the decision of the Editorial and Publishing Council of Yuriy Fedkovych Chernivtsi National University

Textbook prepared for students studying modern approaches in web development.

Examples throughout the book are primarily based on the .NET Web API framework, with coverage of technologies such as Blazor, React, REST API, GraphQL, JWT authentication, Webhooks, WebSockets, security practices (including CORS), logging, and architectural patterns for web development.

Recommended for students of the Faculty of Mathematics and Informatics.

Contents

Introduction.

The web has evolved into the central platform for delivering applications of every scale—from simple informational pages to complex systems used daily in finance, healthcare, communication, and entertainment. What distinguishes the modern web is not only its global reach but also the diversity of technologies that make rich, interactive, and secure applications possible.

This textbook, Modern Client Web-Technologies, introduces students to the essential concepts and practices of today's web development with a strong focus on the .NET ecosystem. While many resources describe web technologies in isolation, this book emphasizes how client-side applications interact with powerful backends, using .NET Web API as the foundation.

Students will begin by exploring how to build and consume REST APIs, then move toward more flexible approaches such as GraphQL. Along the way, practical examples will show how authentication with JWT, real-time communication through WebSockets and webhooks, and the enforcement of security standards such as CORS fit into professional applications. Special attention is given to patterns for web development that help ensure maintainability and scalability, as well as practices like structured logging that support reliability in production environments.

On the client side, two of the most widely adopted technologies—Blazor and React—will be presented. Students will learn how each framework approaches rendering, state management, and integration with APIs, gaining the ability to compare and evaluate them in different scenarios.

The overarching goal of this textbook is not simply to provide a catalog of technologies, but to give students the skills to design and implement complete, modern web solutions. By working through real-world examples, students will understand how the pieces fit together: from building secure APIs, to connecting them with interactive client applications, to applying architectural patterns that keep projects manageable as they grow.

In mastering these technologies, students will be prepared to build applications that are not only functional and efficient, but also secure, reliable, and future-proof—qualities that define professional web development today.

Chapter 1

ASP.NET and Blazor.

ASP.NET is a robust framework developed by Microsoft for building modern web applications. It provides a scalable and high-performance environment for web development, supporting various technologies such as MVC, Razor Pages, Web APIs, and Blazor.

Blazor is a relatively new framework within the ASP.NET ecosystem that enables developers to build interactive web applications using C# and .NET instead of JavaScript. With Blazor, developers can create both client-side and server-side web applications with a consistent programming model.

Blazor is a component-based UI framework that allows developers to build web applications using C# and Razor templates instead of JavaScript. It offers two hosting models: Blazor Server (the application runs on the server, and UI updates are sent over a SignalR connection) and Blazor WebAssembly (the application runs entirely in the user's browser using WebAssembly)

ASP.NET and Blazor together provide a powerful framework for building modern web applications. Whether using Blazor Server or WebAssembly, developers can leverage C# and .NET to create interactive and scalable web applications.

Implementation.

Let's create a new solution using Visual Studio, and choose Blazor Web App from the list of available template options.

After we have created a new solution with Blazor project (in our case, we named the project 'demo-sln'),

When we run our application, we are greeted with a standard landing page, that we can now configure to our liking.

Let's start by adding folder 'Models' and adding new item 'Product.cs' that would encapsulate information about certain product our application would be handling.

```
namespace demo_sln.Models;

public class Product
{
    public int Id { get; set; }
```

```
    public string? Name { get; set; }
    public string? Description { get; set; }
    public decimal Price { get; set; }
    public string? Category { get; set; }
}
```

Now we have to create a service, that would be handling these products. This service will be returning products upon request and adding new products to a mock database (we will not be actually creating a database as a part of this particular lesson). Let's name our service 'ProductService.cs'. We will first create an interface and then an actual implementation class:

```
using demo_sln.Models;

namespace demo_sln.Services;

public interface IProductsService
{
    void AddProduct(Product product);
    List<Product> GetProducts();
    Product GetProduct(int id);
}
```

Implementation of that interface will inherit the interface:

```
using demo_sln.Models;

namespace demo_sln.Services;

public class ProductsService : IProductsService
{
    private List<Product> _products;

    public ProductsService()
    {
        _products = new List<Product>()
            {
                new Product { Id = 1, Name = "Test name",
                   Description = "Descr"},
                new Product { Id = 2, Name = "Test name 2",
                   Description = "Descr 2"}
            };
    }
```

```
    public List<Product> GetProducts()
    {
        return _products;
    }

    public void AddProduct(Product product)
    {
        product.Id = _products.Count() + 1;
        _products.Add(product);
    }

    public Product GetProduct(int id)
    {
        var product = _products.FirstOrDefault(p => p.Id == id);
        if (product == null)
        {
            throw new KeyNotFoundException();
        }
        return product;
    }
}
```

A few notes about interfaces and why we need them in C#:

An interface in C# is a contract that defines a set of methods and properties without providing an implementation. Classes that implement the interface must provide concrete implementations for all its members.

Using interfaces in C# provides several benefits. First of all, they provide abstraction - they hide implementation details and expose only necessary functionalities. Loose coupling - interfaces reduce dependencies between classes, making the system more modular. When it comes to testability - they make unit testing easier by allowing mock implementations. And code maintainability - simplify updating or replacing components without breaking the existing code.

Here we use interfaces to provide dependency injection to our code: design pattern that allows injecting dependencies instead of creating them inside a class. This approach improves testability and maintainability. Thus, we use at all times interfaces instead of actual implementations.

Here is an example of how we can provide actual dependency injection of our ProductService:

```
builder.Services.AddSingleton<IProductsService, ProductsService>();
```

Note, that we choose AddSingleton option - that is important. We will provide more information on singleton pattern later - for now what is important, is that this way we can ensure data consistency for the application lifetime without resorting to actually adding database to our application for now.

Now let's add our first razor component - file " to folder Components//Pages

```
@using demo_sln.Models

<div class="item-row">
    <p class="item-name">@Product?.Name</p>

    <p class="item-description">@Product?.Description</p>
</div>

@code {
    [Parameter]
    public Product? Product { get; set; }
}
```

This is a Razor component. Blazor Razor components are the building blocks of Blazor applications. Think of them as reusable UI pieces, like buttons, forms, or entire pages, written in a mix of C# and HTML. Here in the 'code' segment we see C# code inside the component, and '@Product' variable binds data in code part with the view. Product is market with attribute '[Parameter]' which means that it is a parameter that is passed to that component from another component. Before we define the higher level component, let's first add a ProductItem.razor.css file, that would contain styles for this component:

```
.item-row {
    display: flex;
    align-items: center;
    justify-content: space-between;
    padding: 10px;
    border: 1px solid #ddd;
    border-radius: 8px;
    background-color: #f9f9f9;
    box-shadow: 2px 2px 5px rgba(0, 0, 0, 0.1);
    margin: 10px auto;
}

.item-name {
    font-weight: bold;
    font-size: 18px;
```

```
    color: #333;
    margin-right: 10px;
}

.item-description {
    font-size: 14px;
    color: #666;
    flex-grow: 1;
}
```

And now let's define top-level component Products.razor:

```
@page "/products"
@using demo_sln.Models
@using demo_sln.Services
@rendermode InteractiveServer
@inject IProductsService _productsService

<h3>Products</h3>

@if (_products == null)
{
    <h3>Please wait a bit</h3>
}
else
{
    <div>
        @foreach (var product in _products)
        {
            <ProductItem Product="product"></ProductItem>
        }
    </div>
}
<div>
    <input @bind="_name" />
    <input @bind="_description" />
    <button @onclick="SubmitData">Click me to add new
       product</button>

</div>
```

```
@code {
    private List<Product>? _products;

    private string? _name;
    private string? _description;

    protected override Task OnInitializedAsync()
    {
        _products = _productsService.GetProducts();
        return base.OnInitializedAsync();
    }

    private void SubmitData()
    {
        _productsService.AddProduct(
            new Product() { Name = _name, Description = _description
               }
        );
        _name = String.Empty;
        _description = String.Empty;
        _products = _productsService.GetProducts();
    }
}
```

In this component:

- @page "/products" - Defines the route where this component will be accessible.
- @inject IProductsService _productsService - Injects a service to fetch product data.
- @if (_products == null) - Displays a loading message if data isn't available yet.
- @foreach (var product in _products) - Iterates over the products list and displays them using the ProductItem component.
- @bind - Binds input fields to _name and _description properties.
- @onclick="SubmitData" - Calls SubmitData() when the button is clicked.

Before we start our application, we need to add a link that would re-address us to the products list - we do it in the NavMenu.razor component in folder Layout, modify it in the following way:

```
<div class="top-row ps-3 navbar navbar-dark">
    <div class="container-fluid">
        <a class="navbar-brand" href="">demo-sln</a>
    </div>
</div>

<input type="checkbox" title="Navigation menu"
   class="navbar-toggler" />

<div class="nav-scrollable"
   onclick="document.querySelector('.navbar-toggler').click()">
    <nav class="flex-column">
        <div class="nav-item px-3">
            <NavLink class="nav-link" href=""
               Match="NavLinkMatch.All">
                <span class="bi bi-house-door-fill-nav-menu"
                   aria-hidden="true"></span> Home
            </NavLink>
        </div>

        <div class="nav-item px-3">
            <NavLink class="nav-link" href="products">
                <span class="bi bi-plus-square-fill-nav-menu"
                   aria-hidden="true"></span> Products
            </NavLink>
        </div>
    </nav>
</div>
```

Now suppose we want to add a page for each individual product item. We can do that with routing: define a route pattern with id of the product in the address of the page - add item ProductPage.razor:

```
@page "/products/{Id:int}"
@inject IProductsService _productsService
@using demo_sln.Models
@using demo_sln.Services
<h3>Product</h3>

<p>@_product?.Name</p>
```

```
<p>@_product?.Description</p>


@code {
    [Parameter]
    public int Id { get; set; }

    private Product? _product { get; set; }

    protected override Task OnInitializedAsync()
    {
        _product = _productsService.GetProduct(Id);
        return base.OnInitializedAsync();
    }
}
```

Here directive @page "/products/Id:int" defines the route by which we can access that component. Now we modify our ProductItem.razor component a bit to have a link to that page:

```
@using demo_sln.Models

<div class="item-row">
    <span class="item-name">@Product?.Name</span>

    <span class="item-description">@Product?.Description</span>

    <NavLink href=@link>Click</NavLink>
</div>

@code {
    [Parameter]
    public Product? Product { get; set; }

    public string link => "/products/" + Product?.Id;
}
```

Now when we click on the link, we are re-addressed to the page of a specific product entity, that we can now modify to our liking.

A few words on JS Interop.

In essence, JS Interop allows bidirectional communication between C# and JavaScript, in other words, it allows C# code to invoke JS functions, and JS functions to call

C# code.

Let's integrate JS Interop to our code. First, add folder named 'js' to wwwroot folder and add app.js file there. Write the following to it:

```
function sayHello(name) {
    alert(`Hello ${name}`);
}
```

And add the following line to the App.razor file:

```
<script src="js/app.js"></script>
```

Modify Prodcuts.razor file:

```
@page "/products"
@using demo_sln.Models
@using demo_sln.Services
@rendermode InteractiveServer
@inject IProductsService _productsService
@inject IJSRuntime Js

<h3>Products</h3>

@if (_products == null)
{
    <h3>Please wait a bit</h3>
}
else
{
    <div>
        @foreach (var product in _products)
        {
            <ProductItem Product="product"></ProductItem>
        }
    </div>
}
<div>
    <input @bind="_name" />
    <input @bind="_description" />
    <button @onclick="SubmitData">Click me to add new
       product</button>
    <button @onclick="SayHello">Say Hello</button>
</div>
```

```
@code {
    private List<Product>? _products;

    private string? _name;
    private string? _description;

    protected override Task OnInitializedAsync()
    {
        _products = _productsService.GetProducts();
        return base.OnInitializedAsync();
    }

    private void SubmitData()
    {
        _productsService.AddProduct(
            new Product() { Name = _name, Description = _description
               }
        );
        _name = String.Empty;
        _description = String.Empty;
        _products = _productsService.GetProducts();
    }

    private async Task SayHello()
    {
        await Js.InvokeVoidAsync("sayHello", args: "products");
    }
}
```

Now you see browser alert any time you click button. You can integrate more Java Script functionality to your Razor components!

We can also call C# code from JavaScript file using JSInterop. Take a look at the following static method in ProductsService.cs:

```
[JSInvokable("LogMessage")]
public static Product LogMessage(int id)
{
    Console.WriteLine("Hello message " + id);
    return new Product() { Id = 3, Name = "test"};
```

```
}
```

We can call it from JavaScript file by specifying assembly (in our case, 'demo-sln'), method identifier (LogMessage) and parameter to be passed:

```
async function callCSharpMethod() {
    const result = await DotNet.invokeMethodAsync("demo-sln",
       "LogMessage", 1);
    console.log(result);
}
```

Noiw by adding the following button:

```
<button onclick="callCSharpMethod()">Call C# Method</button>
```

We can see the execution result being logged to console in both browser and DotNet console.

Chapter 2

React basics.

React.js is a powerful JavaScript library for building user interfaces. It allows developers to create dynamic and responsive web applications efficiently. One of its key features is its declarative and component-based architecture, which makes UI development more structured and manageable.

React was developed by Jordan Walke, a software engineer at Facebook, and was first released in 2013. It was created to improve Facebook's complex UI development by introducing a more efficient way to manage updates and rendering. Over time, React gained widespread adoption due to its performance benefits and reusability of components.

React development is streamlined with automatic reload and hot reloading features. When changes are made to the code, they are instantly reflected in the browser without needing a full page refresh. This significantly speeds up the development process and improves the developer experience.

React facilitates the development of Single Page Applications (SPA). Unlike traditional multi-page applications, an SPA loads a single HTML file and dynamically updates the content as needed without requiring a page reload. This results in faster navigation and a more seamless user experience.

It is important to understand difference between React and React Native. Whereas React is used for building web applications, React Native is a framework used primarily for mobile apps. React runs in browsers, React Native on Native UI components on iOS and Android.

There were several major versions of React:

- React 16 (2017): Introduced the Fiber reconciliation engine for better performance.
- React 17 (2020): Improved gradual upgrades without breaking changes.
- React 18 (2022): Added features like Concurrent Rendering and Suspense for better performance.
- Latest Version: React continues to evolve with new features and optimizations.

Example simple React application

Let's use create-react-app tool to create a new react app:

```
npx create-react-app new-app
```

After that:

```
cd new-app
```

And:

```
npm start
```

React development is streamlined with automatic reload and hot reloading features. When changes are made to the code, they are instantly reflected in the browser without needing a full page refresh. This significantly speeds up the development process and improves the developer experience.

Lets build single page app, that returns single html file. Take a look at a generated repository. The index.js file serves as the entry point of a React application. It is responsible for importing necessary modules and rendering the root component into the DOM. The ReactDOM.render() function (or its modern equivalent, createRoot) is executed to render the main App component inside a specific DOM element, usually an element with the id root in index.html.

The App.js file contains the main application component. In React, components are written as JavaScript functions that return JSX. JSX (JavaScript XML) allows developers to write HTML-like code within JavaScript, making UI development more intuitive.

Let's rewrite the App.js component:

```
import './App.css';
import React from 'react';

function App() {
  return (
    <div className='container'>
      <h2>Student list </h2>
      <ul className='stud-list'>
        <li>Ivanov</li>
        <li>Petrov</li>
        <li>Frolov</li>
      </ul>
    </div>
  );
}
```

```
export default App;
```

Note: we can set styles in App.css, for example, class container. As the word class is reserved in JS, instead of class of the element we write className.

Add the following in App.css file:

```
.container {
    max-width: 400px;
    margin: 20px auto;
    padding: 20px;
    background: #f8f9fa;
    border-radius: 10px;
    box-shadow: 0 4px 6px rgba(0, 0, 0, 0.1);
    text-align: center;
}

h2 {
    font-size: 1.5rem;
    color: #333;
    margin-bottom: 15px;
}

.stud-list {
    list-style: none;
    padding: 0;
}

    .stud-list li {
        background: #ffffff;
        padding: 10px;
        margin: 5px 0;
        border-radius: 5px;
        box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
        transition: background 0.3s ease-in-out;
    }

        .stud-list li:hover {
            background: #e3e6ea;
            cursor: pointer;
        }
```

List element should be outsourced to the components. Add the components folder in `src`, there add StudList.js file.

There we add the following:

```
import React from 'react';

const StudList = () => {
    return (
        <ul className='stud-list'>
        <li>Ivanov</li>
        <li>Petrov</li>
        <li>Frolov</li>
      </ul>
    )
}

export default StudList;
```

And change the App.js to the following:

```
import './App.css';
import React from 'react';
import StudList from './components/StudList';

function App() {
  return (
    <div className='container'>
      <h2>Student list </h2>
      <StudList />
    </div>
  );
}

export default App;
```

If we save now, we see the same output and the same styling as before. But it is still a good practice to add a separate css file for a component. StudList.css in components folder:

```
.stud-list {
    list-style: none;
    padding: 0;
}
```

```
    .stud-list li {
        background: #ffffff;
        padding: 10px;
        margin: 5px 0;
        border-radius: 5px;
        box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
        transition: background 0.3s ease-in-out;
    }

        .stud-list li:hover {
            background: #e3e6ea;
            cursor: pointer;
        }
```

React Props.

With splitting our code into components, we have now achieved separation of concerns. But how do we pass data between the components? Here useful concept to introduce is the concept of props. Props are arguments that are passed automatically by react to the component.

Let's rewrite `App.js`:

```
import './App.css';
import React from 'react';
import StudList from './components/StudList';

function App() {
  const sutdList = [
    {id:1, text: "Ivanov"},
    {id:2, text: "Petrov"},
    {id:3, text: "Frolov"}
  ];

  return (
    <div className='container'>
      <h2>Student list </h2>
      <StudList studs={sutdList}/>
    </div>
  );
}

export default App;
```

`StudList.js`: let's map array of elements to an array of jsx objects. Also added `console.log` to see, what comes in props.

```
import React from 'react';
import './StudList.css'

const StudList = props => {
    console.log(props.studs);

    return (
        <ul className='stud-list'>
            {
                props.studs.map((stud) => {
                    return <li>{stud.text}</li>
                })
            }
        </ul>
    )
}

export default StudList;
```

React Events

Add component NewStudent.js

```
import React from 'react';
import './NewStudent.css'

const NewStudent = props => {
    const addStudentHandler = event => {
        event.preventDefault();

        const newStud = {
            id: 1,
            text: 'text'
        }

        props.onAddStud(newStud);
    }

    return (
```

```
    <form className='new-student' onSubmit={addStudentHandler}>
        <input type='text'/>
        <button type='submit'>Add Student</button>
    </form>
    )
}

export default NewStudent;
```

Add styles for it:

```
.new-student {
  display: flex;
  gap: 10px;
  max-width: 400px;
  margin: 20px auto;
  padding: 15px;
  background: #f8f9fa;
  border-radius: 10px;
  box-shadow: 0 4px 6px rgba(0, 0, 0, 0.1);
}

.new-student input {
  flex: 1;
  padding: 10px;
  border: 2px solid #ccc;
  border-radius: 5px;
  font-size: 1rem;
  outline: none;
  transition: border-color 0.3s;
}

.new-student input:focus {
  border-color: #007bff;
}

.new-student button {
  padding: 10px 15px;
  background: #007bff;
  color: #fff;
  border: none;
  border-radius: 5px;
```

```
  cursor: pointer;
  font-size: 1rem;
  transition: background 0.3s ease-in-out;
}

.new-student button:hover {
  background: #0056b3;
}
```

And in App.js:

```
import './App.css';
import React from 'react';
import StudList from './components/StudList';
import NewStudent from './components/NewStudent/NewStudent'

function App() {
  const studList = [
    {id:1, text: "Ivanov"},
    {id:2, text: "Petrov"},
    {id:3, text: "Frolov"}
  ];

  const addNewStudHandler = (newStud) => {
    studList.push(newStud);
  }

  return (
    <div className='container'>
      <h2>Student list </h2>
      <NewStudent onAddStud={addNewStudHandler} />
      <StudList studs={studList}/>
    </div>
  );
}

export default App;
```

Now every time `addNewStudHandler` method is triggered in the child component via submittion of the form, the parent event is notified and has a new entry in the array.

React State

State in React is a way to manage dynamic data within a component. It allows components to store, update, and react to changes in data over time. State is typically declared using the useState hook in functional components.

In App.js:

```
import './App.css';
import React, {useState} from 'react';
import StudList from './components/StudList';
import NewStudent from './components/NewStudent/NewStudent';


function App() {
  const [studList, setStud ] = useState(
    [
      {id:1, text: "Ivanov"},
      {id:2, text: "Petrov"},
      {id:3, text: "Frolov"}
    ]
  );

  // const studList = [
  //   {id:1, text: "Ivanov"},
  //   {id:2, text: "Petrov"},
  //   {id:3, text: "Frolov"}
  // ];

  const addNewStudHandler = (newStud) => {
    // setStud(studList.concat(newStud));
    setStud( (oldStuList) => {return oldStuList.concat(newStud)} );
  }

  return (
    <div className='container'>
      <h2>Student list </h2>
      <NewStudent onAddStud={addNewStudHandler} />
      <StudList studs={studList}/>
    </div>
  );
}

export default App;
```

Here setStud - method that allows to handle state updates. State can be anything. UseState always returns array with two elements - state and method to update it.

React Data binding

Data binding in React is the process of linking data between the component's state or props and the UI. React primarily uses one-way data binding, meaning that data flows in a single direction—from the component's state or props to the UI. This makes it predictable and easier to debug.

In NewStud.js:

```
import React, {useState} from 'react';
import './NewStudent.css'

const NewStudent = props => {
    const [enteredText, setText] = useState('');

    const addStudentHandler = event => {
        event.preventDefault();

        const newStud = {
            id: 1,
            text: enteredText
        }

        setText('');

        props.onAddStud(newStud);
    }

    const textChangeHandler = event => {
        setText(event.target.value);
    }

    return (
    <form className='new-student' onSubmit={addStudentHandler}>
        <input type='text' value={enteredText}
           onChange={textChangeHandler}/>
        <button type='submit'>Add Student</button>
    </form>
    )
}

export default NewStudent;
```

Chapter 3

REST.

A REST API is a set of rules and conventions for building and interacting with web services. It follows the principles of Representational State Transfer (REST), a lightweight architecture that uses standard web protocols such as HTTP. REST APIs are designed to enable communication between clients (like a web browser or mobile app) and servers in a stateless, resource-based manner.

The REST API is built on several key principles that make it an efficient and flexible tool for data exchange.

- Stateless: Why every HTTP request happens in complete isolation? The REST API works in stateless mode, which means that each HTTP request sent to the server contains all the necessary data for its processing. The server does not store information about previous requests from the client. This makes the system more reliable and scalable.
- Client-Server: client and server autonomy and their interaction. The REST API strictly separates the client and server parts of the application. This means that they can develop independently of each other. The client can be a web browser, a mobile application, or any other client application, and the server can be written in any programming language.
- Uniform Interface: four interfaces to achieve uniformity The REST API relies on four main interfaces: resources, HTTP methods (GET, POST, PUT, DELETE), resource representations (how data is presented to the client), and links between resources. This creates uniformity in the way the client and server interact.
- Cacheable: How caching improves application performance The REST API supports caching, which means that the client can temporarily store data to reduce server load and improve performance.
- Layered System: advantages of a layered system of architecture REST API servers can be organized in layers, which makes them more flexible and scalable.

Each layer performs specific functions and can be added or removed without changing the client code.

- Code on Demand: An optional restriction that allows you to download client code. This principle is optional and allows the server to send executable code to the client. This is rarely used and requires additional security measures.

In general, rest api principles provide efficient and flexible interaction between the client and the server, as well as standardization, reliability and stability of the system.

Example REST application with .Net and React.

Let's create an example REST Api application with .Net on back-end. For example, our REST Api project will provide CRUD operations for lists of students and courses they visit in university.

Add new Asp .NET Core Web API project to your solution (in our case, we named the project 'demo-rest'). Also, add new class library project for database access to your solution (we named it 'demo-rest.dal', where 'dal' stands for data access layer) and add a class library project for business logic (we named it 'demo-rest.bll', where 'bll' stands for business logic layer). Do not forget to reference dal project from web api and bll projects, and reference bll project from web api project (you can do it in Visual Studio by right clicking on the project solution explorer and choosing "Add project reference" option).

Then, we should install Entity Framework 8 to our dal project. We can do it via NuGet manager in Visual Studio or via powershell. In particular, for our needs we are going to install the following NuGet libraries:

- Install-Package Microsoft.EntityFrameworkCore
- Install-Package Microsoft.EntityFrameworkCore.Design
- Install-Package Microsoft.EntityFrameworkCore.Tools
- Install-Package Microsoft.EntityFrameworkCore.SqlServer

Entity Framework (EF) is an open-source object-relational mapper (ORM) for .NET applications. It simplifies the interaction between .NET applications and relational databases by providing a high-level abstraction over database operations. EF is a part of the .NET ecosystem and is widely used for data access in ASP.NET applications.

Entity Framework is particularly useful for building data-driven .NET applications quickly while adhering to modern development best practices.

In this demo we will use EntityFrameworkCore version 9.0, if you use newer version you might want to consult with the official documentation, especially, if you run into any unexpected errors.

Before adding database access services to our application, we are going to need to add connection string to our database to application.

A connection string is a string of text used by an application to establish a connection to a database. It contains the necessary information, such as the database type, location, and credentials, for connecting to the desired database system. Connection strings act as a bridge between the application and the database, specifying how the application should communicate with the database server.

We add connection string by modifying our appsetting.json file (in Web Api project) in the following way:

```
{
  "ConnectionStrings": {
    "DefaultConnection":
       "Server=(localdb)\\mssqllocaldb;Database=DemoDb;Trusted_Connection=T
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

The field "DefaultConnection" contains an example of the connection string, that references local database.

Now, we add models (entities) to our data access project, that would represent entries in our database. Class Student will incapsulate information about a particular student (in this case, his name and date of birth), class Course - information about a particular course in the curriculum (name and description of the course), class StudentCourse will store information about which students signed up to which courses - it is an intermediate or association table linking the two tables together.

```
using System.ComponentModel.DataAnnotations;

namespace demo_rest.dal.Entities;

public class Student
{
    [Key]
```

```
    public int Id { get; set; }
    public string Name { get; set; }
    public DateTime BirthDate { get; set; }

    public ICollection<StudentCourse> StudentCourses { get; set; }
}
```

```
using System.ComponentModel.DataAnnotations;

namespace demo_rest.dal.Entities;

public class Course
{
    [Key]
    public int Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }

    public ICollection<StudentCourse> StudentCourses { get; set; }
}
```

```
namespace demo_rest.dal.Entities;

public class StudentCourse
{
    public int StudentId { get; set; }
    public Student Student { get; set; }

    public int CourseId { get; set; }
    public Course Course { get; set; }
}
```

The attribute `[Key]` represents field, that is a primary key in the database. The line `public ICollection<StudentCourse> StudentCourses` represents navigational property, that allows us to access related row in the table, that is linked via primary key.

After that, we must add DbContext to our application. In Entity Framework, the DbContext class is the central point for interacting with the database. It acts as a bridge between your application and the database, managing database operations and providing functionality for querying and saving data. Example DbContext in our demo project:

```
using demo_rest.dal.Entities;
using Microsoft.EntityFrameworkCore;

namespace demo_rest.dal;

public class DemoDbContext : DbContext
{
    public DemoDbContext(DbContextOptions options) : base(options)
    {
    }

    public DbSet<Student> Students { get; set; }
    public DbSet<Course> Courses { get; set; }
    public DbSet<StudentCourse> StudentCourses { get; set; }

    protected override void OnModelCreating(ModelBuilder
       modelBuilder)
    {
        modelBuilder.Entity<StudentCourse>()
          .HasKey(sc => new { sc.StudentId, sc.CourseId });

        modelBuilder.Entity<StudentCourse>()
          .HasOne(sc => sc.Student)
          .WithMany(s => s.StudentCourses)
          .HasForeignKey(sc => sc.StudentId);

        modelBuilder.Entity<StudentCourse>()
          .HasOne(sc => sc.Course)
          .WithMany(c => c.StudentCourses)
          .HasForeignKey(sc => sc.CourseId);
    }
}
```

The line `modelBuilder.Entity<StudentCourse>().HasKey(...` configures the table StudentCourses to have a composite key, that would consist of fields StudentId and CourseId. The method `.HasForeignKey(...` configures foreign key relations.

Now we can use this dbContext to interact with the database. For example, if we want to retrieve an information about a particular student, given that we have id of his entry in the database entry, we can do it using method `FirstOrDefaultAsync`, and pass that id via a delegate, for example:

```
public async Task<Student?> GetStudentAsync(int id)
{
    return await _demoDbContext.Students
        .FirstOrDefaultAsync(s => s.Id == id);
}
```

That method will return a student entry from the database with a given id, or Null, if there is not entry with that id in the database.

After that, we can create a service, that would retrieve entity information on request, as well as provide functionality to adds, modify and delete records in our database. Here is example from demo application:

```
using demo_rest.bll.Interfaces;
using demo_rest.dal;
using demo_rest.dal.Entities;
using Microsoft.EntityFrameworkCore;

namespace demo_rest.bll.Services;

public class StudentService : IStudentService
{
    private DemoDbContext _demoDbContext;

    public StudentService(DemoDbContext demoDbContext)
    {
        _demoDbContext = demoDbContext;
    }

    public async Task<Student?> GetStudentAsync(int id)
    {
        return await _demoDbContext.Students
            .FirstOrDefaultAsync(s => s.Id == id);
    }

    public async Task<IEnumerable<Student>> GetStudentsAsync(int
       skip, int take)
    {
        return await _demoDbContext.Students
            .Skip(skip)
            .Take(take)
            .ToListAsync();
    }
```

```
    public async Task<Student> AddStudentAsync(Student student)
    {
        var studentAdded = await _demoDbContext.AddAsync(student);
        await _demoDbContext.SaveChangesAsync();
        return studentAdded.Entity;
    }

    public async Task<Student> UpdateStudentAsync(Student student)
    {
        var studentModified = _demoDbContext.Update(student);
        await _demoDbContext.SaveChangesAsync();
        return studentModified.Entity;
    }

    public async Task DeleteStudentAsync(int id)
    {
        var student = await _demoDbContext.Students
            .FirstOrDefaultAsync(s => s.Id == id);
        if (student == null)
        {
            throw new KeyNotFoundException();
        }
        _demoDbContext.Remove(student);
        await _demoDbContext.SaveChangesAsync();
    }
}
```

We also declare interface, that would contain all of these methods, and provide access to them for other services via dependency injection:

```
using demo_rest.dal.Entities;

namespace demo_rest.bll.Interfaces;

public interface IStudentService
{
    Task<Student> AddStudentAsync(Student student);
    Task DeleteStudentAsync(int id);
    Task<Student?> GetStudentAsync(int id);
    Task<IEnumerable<Student>> GetStudentsAsync(int skip, int take);
    Task<Student> UpdateStudentAsync(Student student);
```

```
}
```

For the purposes of separation of responsibility, we can add a data transfer object or DTO to our Web Api project:

```
namespace demo_rest.Models;

public class StudentDto
{
    public string Name { get; set; }
    public DateTime BirthDate { get; set; }
}
```

After that, we add a controller to Web API project, that would provide the data to API upon requests.

In the context of ASP.NET Web API, a controller is a class that handles incoming HTTP requests, processes them, and returns HTTP responses to the client. Controllers are at the core of the Model-View-Controller (MVC) architectural pattern, where they act as intermediaries between the model (data and business logic) and the view (response sent to the client).

Example of the controller:

```
using demo_rest.bll.Interfaces;
using demo_rest.dal.Entities;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;

namespace demo_rest.Controllers;

[Route("[controller]/[action]")]
[ApiController]
public class StudentController : ControllerBase
{
    private IStudentService _studentService;

    public StudentController(IStudentService studentService)
    {
        _studentService = studentService;
    }

    [HttpGet(Name = "GetStudent")]
    public async Task<Student> GetStudent(int id)
    {
```

```
        var student = await _studentService.GetStudentAsync(id);
        if (student == null)
        {
            throw new KeyNotFoundException();
        }
        return student;
    }

    [HttpGet(Name = "GetStudents")]
    public async Task<IEnumerable<Student>> GetStudents(
        int skip, int take)
    {
        return await _studentService.GetStudentsAsync(skip, take);
    }

    [HttpPost(Name = "UpdateStudent")]
    public async Task<Student> UpdateStudent(Student student)
    {
        return await _studentService.UpdateStudentAsync(student);
    }

    [HttpPost(Name = "AddStudent")]
    public async Task<Student> AddStudent(StudentDto studentDto)
    {
        var student = new Student()
        {
            Name = studentDto.Name,
            BirthDate = studentDto.BirthDate
        };
        return await _studentService.AddStudentAsync(student);


    [HttpPost(Name = "DeleteStudent")]
    public async Task DeleteStudent(int id)
    {
        await _studentService.DeleteStudentAsync(id);
    }
}
```

Now we have to register our services in Program.cs class of Web API project via dependency injection.

Dependency Injection (DI) is a design pattern and technique used in .NET to achieve Inversion of Control (IoC). It enables the creation and management of object dependencies by delegating this responsibility to an external framework or container, rather than directly instantiating those dependencies within a class. This promotes loose coupling, testability, and maintainability in applications.

An example dependency injection in .Net can be implemented in the following way:

```
builder.Services.AddTransient<IService, Service>();
```

Where IService is the abstract inteface, and Service is a concrete implementation of that interface.

Dependency injection example for the types, that we have defined above:

```
using demo_rest.bll.Interfaces;
using demo_rest.bll.Services;
using demo_rest.dal;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();
// Learn more about configuring Swagger/OpenAPI at
   https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

builder.Services.AddDbContext<DemoDbContext>(options =>
{
    options.UseSqlServer(builder.Configuration.GetConnectionString("Default
});
builder.Services.AddTransient<IStudentService, StudentService>();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}
```

```
app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Before we startup our application, we have to create our database via migration.

In Entity Framework, a migration is a feature that enables you to manage and apply incremental changes to your database schema while keeping it in sync with your application's data model. Migrations are particularly useful in scenarios where the data model evolves over time as the application grows or requirements change.

But first, do not forget to create the database before running the update-database command. In our case, the name of the database is DemoDb - you can see it as part of the connection string.

- Add-Migration InitialCreate
- Update-Database

Before running the migration, we can check out the migration class, generated by entity framework:

```
using System;
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace demo_rest.dal.Migrations
{
    /// <inheritdoc />
    public partial class InitialCreate : Migration
    {
        /// <inheritdoc />
        protected override void Up(MigrationBuilder
           migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "Courses",
                columns: table => new
                {
```

```
            Id = table.Column<int>(type: "int", nullable:
                false)
                .Annotation("SqlServer:Identity", "1, 1"),
            Name = table.Column<string>(type:
                "nvarchar(max)", nullable: false),
            Description = table.Column<string>(type:
                "nvarchar(max)", nullable: false)
        },
        constraints: table =>
        {
            table.PrimaryKey("PK_Courses", x => x.Id);
        });

    migrationBuilder.CreateTable(
        name: "Students",
        columns: table => new
        {
            Id = table.Column<int>(type: "int", nullable:
                false)
                .Annotation("SqlServer:Identity", "1, 1"),
            Name = table.Column<string>(type:
                "nvarchar(max)", nullable: false),
            BirthDate = table.Column<DateTime>(type:
                "datetime2", nullable: false)
        },
        constraints: table =>
        {
            table.PrimaryKey("PK_Students", x => x.Id);
        });

    migrationBuilder.CreateTable(
        name: "StudentCourses",
        columns: table => new
        {
            StudentId = table.Column<int>(type: "int",
                nullable: false),
            CourseId = table.Column<int>(type: "int",
                nullable: false)
        },
        constraints: table =>
        {
```

```
                    table.PrimaryKey("PK_StudentCourses", x => new {
                       x.StudentId, x.CourseId });
                    table.ForeignKey(
                        name: "FK_StudentCourses_Courses_CourseId",
                        column: x => x.CourseId,
                        principalTable: "Courses",
                        principalColumn: "Id",
                        onDelete: ReferentialAction.Cascade);
                    table.ForeignKey(
                        name: "FK_StudentCourses_Students_StudentId",
                        column: x => x.StudentId,
                        principalTable: "Students",
                        principalColumn: "Id",
                        onDelete: ReferentialAction.Cascade);
                });

            migrationBuilder.CreateIndex(
                name: "IX_StudentCourses_CourseId",
                table: "StudentCourses",
                column: "CourseId");
        }

        /// <inheritdoc />
        protected override void Down(MigrationBuilder
           migrationBuilder)
        {
            migrationBuilder.DropTable(
                name: "StudentCourses");

            migrationBuilder.DropTable(
                name: "Courses");

            migrationBuilder.DropTable(
                name: "Students");
        }
    }
}
```

Now, when we start our project in Visual Studio, we can use CRUD operations using Swagger. Swagger is a set of open-source tools for designing, building, documenting, and consuming RESTful APIs. In the .NET ecosystem, Swagger is

implemented using Swashbuckle, a popular library that integrates seamlessly with ASP.NET Core to generate interactive API documentation. Swagger is built on the OpenAPI Specification (OAS), a standard format for API definitions.

In .NET 8, new Web API projects that you intialize via Visual Studio, have Swagger built in, so you should be greeted with a landing page in your browser, that lists all the endpoints, that you have described in StudentController.cs.

You can now try adding new entries in your application, retrieving, modifying and deleting them.

Chapter 4

CQRS pattern in web-development.

Modern web applications often face increasing complexity as they grow in scale, functionality, and number of users. A critical challenge for developers is maintaining clarity between how an application reads data and how it modifies data. A traditional architecture often handles both operations within the same models and services, which can lead to inefficiencies, poor scalability, and unnecessary coupling. This is where the Command and Query Responsibility Segregation (CQRS) pattern becomes significant.

At its core, CQRS is based on a simple but powerful idea: the operations that read data (queries) and the operations that change data (commands) should be separated into different models or components. This separation is not only a matter of code organization but also a deliberate design decision that can impact scalability, performance, and maintainability.

The motivation behind CQRS comes from the different requirements of reading and writing data. Queries usually need to be fast and efficient, often optimized for reporting or user-facing dashboards. They may require denormalized views of the data that can be retrieved with minimal joins or transformations. Commands, on the other hand, are concerned with ensuring business rules are enforced consistently when data is modified. They typically involve validation, transactional consistency, and domain logic. By segregating these responsibilities, each side of the system can be designed and optimized independently.

One of the key benefits of CQRS is scalability. Applications with a large number of users often have an imbalance between reads and writes, where queries vastly outnumber commands. By splitting the two responsibilities, developers can scale query handling independently of command processing. For example, caching layers, read replicas, or even entirely different database technologies can be introduced for queries without affecting the integrity of command operations.

Another important advantage is clearer domain modeling. Commands express intentions that modify the system's state, such as "create a new order" or "update customer details." Queries express information needs, such as "list all pending orders" or "retrieve a customer profile." This separation reflects real-world thinking and allows developers to design business logic in a way that is more aligned with

the domain language and user scenarios.

CQRS also lays the foundation for more advanced patterns, particularly event sourcing. When combined, these approaches enable full traceability of system changes, audit capabilities, and the possibility to reconstruct the system state at any point in time. This is particularly valuable in domains where accountability, compliance, or historical analysis is essential.

Despite its advantages, CQRS is not always necessary. For smaller systems with straightforward requirements, separating commands and queries may introduce unnecessary complexity. It is best applied when the system has complex business rules, large-scale read and write operations, or requirements for independent optimization of queries and commands. In these cases, the clarity, scalability, and maintainability offered by CQRS outweigh the added architectural overhead.

In conclusion, CQRS is a pattern that embodies a separation of concerns at the architectural level. It helps developers design systems where queries and commands are handled independently, leading to greater flexibility and scalability. When used in appropriate contexts, it becomes a powerful tool in the modern web developer's toolkit, enabling applications to evolve gracefully as their complexity and usage grow.

Example of application.

Let's extend the previous chapter with a practical example in .NET Web API where we apply CQRS to an application that handles students. We'll keep it structured and realistic, but clear enough for everybody to follow.

In a traditional architecture, we might have a StudentController with methods like GetStudents, GetStudentById, AddStudent, and UpdateStudent, all calling a single StudentService or StudentRepository. While this works for small applications, it can quickly become messy when the application grows. Queries and commands are mixed, and optimizations for reading can interfere with rules for writing.

Using CQRS, we separate these two concerns.

Commands (Write Operations)

Commands represent intentions to change the system's state. In our student management system, commands might include: add a new student, update student information and delete a student

Each command is represented by a dedicated object (DTO) and handled by a corresponding command handler. For example:

```
// Command
public class AddStudentCommand
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Faculty { get; set; }
}
```

```
// Command Handler
public class AddStudentHandler
{
    private readonly ApplicationDbContext _context;

    public AddStudentHandler(ApplicationDbContext context)
    {
        _context = context;
    }

    public async Task Handle(AddStudentCommand command)
    {
        var student = new Student
        {
            Name = command.Name,
            Age = command.Age,
            Faculty = command.Faculty
        };

        _context.Students.Add(student);
        await _context.SaveChangesAsync();
    }
}
```

Queries are read-only requests for information. They do not change the system. In our example, queries might include: get a list of all students, get a student by ID and search students by faculty.

Each query has its own DTO and query handler. For example:

```
// Query
public class GetStudentByIdQuery
{
    public int Id { get; set; }
}

// Query Handler
public class GetStudentByIdHandler
{
    private readonly ApplicationDbContext _context;

    public GetStudentByIdHandler(ApplicationDbContext context)
```

```
    {
        _context = context;
    }

    public async Task<Student?> Handle(GetStudentByIdQuery query)
    {
        return await _context.Students.FindAsync(query.Id);
    }
}
```

The StudentController delegates commands and queries to the appropriate handlers instead of doing the work itself.

```
[ApiController]
[Route("api/[controller]")]
public class StudentController : ControllerBase
{
    private readonly AddStudentHandler _addHandler;
    private readonly GetStudentByIdHandler _getByIdHandler;

    public StudentController(AddStudentHandler addHandler,
       GetStudentByIdHandler getByIdHandler)
    {
        _addHandler = addHandler;
        _getByIdHandler = getByIdHandler;
    }

    [HttpPost]
    public async Task<IActionResult> AddStudent(AddStudentCommand
       command)
    {
        await _addHandler.Handle(command);
        return Ok("Student added successfully");
    }

    [HttpGet("{id}")]
    public async Task<IActionResult> GetStudent(int id)
    {
        var student = await _getByIdHandler.Handle(new
           GetStudentByIdQuery { Id = id });
        if (student == null) return NotFound();
        return Ok(student);
```

```
    }
}
```

Chapter 5

GraphQL basics, queries and mutations. GraphQL subscriptions.

GraphQL is an open-source data query and manipulation language for APIs, and a runtime for fulfilling queries with existing data. GraphQL was developed internally by Facebook in 2012 before being publicly released in 2015.

Simply put, GraphQL is a query language for APIs and a runtime for executing those queries with existing data. GraphQL provides a complete and understandable description of the data in your API, gives clients the power to ask for exactly what they need and nothing more, makes it easier to evolve APIs over time, and enables powerful developer tools.

Advantages of GraphQL:

- Data retrieval efficiency: GraphQL allows clients to request exactly the data they need, minimizing the chances of requesting too much or too little information. This customized approach improves performance and reduces unnecessary traffic, which ultimately leads to faster server response times.
- One Request, Multiple Resources: Unlike REST, which often requires multiple endpoints for different resources, GraphQL allows clients to retrieve related data in a single request. This reduces the infamous "N+1" problem and simplifies data aggregation.
- Flexible schema evolution: GraphQL's schema-based design allows developers to evolve the API without major changes. New fields and types can be added without impacting existing clients, promoting flexibility and adapting to changing business requirements.
- Strict typing: GraphQL provides strict schema typing by providing clear definitions of available data and operations. This eliminates ambiguity and reduces runtime errors, creating more robust code bases that can be maintained.

Disadvantages:

- Complexity in Caching: Caching in GraphQL can be more challenging due to the dynamic nature of queries. Implementing efficient caching strategies requires careful consideration and additional effort.
- Potential for Over-Fetching: While GraphQL's flexibility is an advantage, inexperienced or poorly optimized queries might still lead to over-fetching of data, impacting performance.

In GraphQL, queries and mutations are two fundamental operations that clients can use to interact with the server and retrieve or manipulate data. They serve distinct purposes and are designed to facilitate efficient and flexible communication between clients and the GraphQL server.

Queries: Queries in GraphQL are used to request data from the server. They resemble a data structure that defines the shape and structure of the data the client is interested in fetching. With queries, clients can specify exactly what fields they need, allowing them to retrieve only the necessary data and avoid over-fetching.

Example of the query:

```
{
  students {
    name
    group
  }
}
```

This query will return the list of students, listing name of the student and group for each student.

If we want only name of each student, we can re-write the query in the following way:

```
{
  students {
    name
  }
}
```

This query will return only a list of student names. Thus, we have mitigated a common problem of over-fetching when working with REST api.

Mutations: Mutations in GraphQL are used to modify data on the server. They provide a way for clients to create, update, or delete data. Mutations are similar in structure to queries but are executed with the intent of making changes to the data on the server.

Example of GraphQL with .Net HotChocolate.

Let's create a new WebApi project in our .NET solution (or you can create a new .net solution from scratch). In it, we will install the following packages (via powershell or NuGet package manager in Visual Studio):

```
Install-Package HotChocolate
Install-Package HotChocolate.AspNetCore
```

Hot Chocolate is an open-source GraphQL server for the Microsoft .NET platform that is compliant with the newest GraphQL. Basically, it is a library, that will allow us to set up a server side GraphQL.

In this demo we will use HotChocolate version 14.1, if you use newer version you might want to consult with the official documentation.

Now, before we start implementing server-side GraphQL, we need to register GraphQL in Program.cs:

```
app.MapGraphQL();
```

Overall, for now, your Program.cs class will look like this:

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();
// Learn more about configuring Swagger/OpenAPI at
   https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapGraphQL();
```

```
app.MapControllers();

app.Run();
```

Now we will implement some entities, that would encapsulate the data, that our GraphQL application operates with, and we will also implement a mock service, that would give that data. For this demo we will not implement database connection and storage of the data, but you might do it for yourself with EntityFramework, see previous chapter.

Supposedly, our application will store the data about authors and books that they wrote. We will define the following data classes in our graphql project:

```
namespace demo_graphql.Dto;

public class Author
{
    public int Id { get; set; }
    public string Name { get; set; }
    public DateTime BirthDate { get; set; }
    public string Country { get; set; }
    public string Email { get; set; }
    public ICollection<Book> Books { get; set; }
}
```

```
namespace demo_graphql.Dto;

public class Book
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Description { get; set; }

}
```

After that, lets create a new AuthorService class, that would provide methods for storing and retrieving the data about authors:

```
using demo_graphql.Dto;

namespace demo_graphql.Services;

public class AuthorService : IAuthorService
{}
```

Let's mock a database storage with a simple list, that would store the authors data:

```
private ICollection<Author> _storage = new List<Author>()
    {
        new Author() {
            Id = 1,
            BirthDate = DateTime.Now.AddYears(-40),
            Name = "Aizek Asimov",
            Country = "USA",
            Email = "aa@gmail.com",
            Books = new List<Book>()
            {
                new Book(){ Id = 1, Title = "Foundation and Empire",
                   Description = "..." },
                new Book(){ Id = 2, Title = "I, Robot", Description
                   = "..." }
            }
        },
        new Author() {
            Id = 2,
            BirthDate = DateTime.Now.AddYears(-50),
            Name = "Ray Bradbury",
            Country = "USA",
            Email = "rb@gmail.com",
            Books = new List<Book>()
            {
                new Book(){ Id = 3, Title = "Fahrenheit 451",
                   Description = "..." },
                new Book(){ Id = 4, Title = "Martian Chronicles",
                   Description = "..." }
            }
        },
        new Author() {
            Id = 2,
            BirthDate = DateTime.Now.AddYears(-60),
            Name = "Stanislaw Lem",
            Country = "Poland",
            Email = "sl@gmail.com",
            Books = new List<Book>()
```

```
            {
                new Book(){ Id = 5, Title = "Solaris", Description =
                    "..." }
            }
        },
    };
```

And provide methods for retrieving the data from our storage:

```
public IEnumerable<Author> GetAuthors(int skip, int take)
    {
        return _storage.Skip(skip).Take(take);
    }

    public Author GetAuthor(int id)
    {
        var author = _storage.FirstOrDefault(x => x.Id == id);
        if (author == null)
        {
            throw new
                System.Collections.Generic.KeyNotFoundException();
        }
        return author;
    }

    public Book GetBook(int id)
    {
        var book = _storage.SelectMany(a => a.Books)
            .FirstOrDefault(x => x.Id == id);
        if (book == null)
        {
            throw new
                System.Collections.Generic.KeyNotFoundException();
        }
        return book;
    }
```

As well as updating that data:

```
public Author AddAuthor(Author author)
    {
        int newId = _storage.Select(x => x.Id).Max() + 1;
        author.Id = newId;
```

```
        _storage.Add(author);
        return author;
    }
```

All together our service will now look like this:

```
using demo_graphql.Dto;

namespace demo_graphql.Services;

public class AuthorService : IAuthorService
{
    private ICollection<Author> _storage = new List<Author>()
    {
        new Author() {
            Id = 1,
            BirthDate = DateTime.Now.AddYears(-40),
            Name = "Aizek Asimov",
            Country = "USA",
            Email = "aa@gmail.com",
            Books = new List<Book>()
            {
                new Book(){ Id = 1, Title = "Foundation and Empire",
                   Description = "..." },
                new Book(){ Id = 2, Title = "I, Robot", Description
                   = "..." }
            }
        },
        new Author() {
            Id = 2,
            BirthDate = DateTime.Now.AddYears(-50),
            Name = "Ray Bradbury",
            Country = "USA",
            Email = "rb@gmail.com",
            Books = new List<Book>()
            {
                new Book(){ Id = 3, Title = "Fahrenheit 451",
                   Description = "..." },
                new Book(){ Id = 4, Title = "Martian Chronicles",
                   Description = "..." }
            }
        },
```

```
    new Author() {
        Id = 2,
        BirthDate = DateTime.Now.AddYears(-60),
        Name = "Stanislaw Lem",
        Country = "Poland",
        Email = "sl@gmail.com",
        Books = new List<Book>()
        {
            new Book(){ Id = 5, Title = "Solaris", Description =
               "..." }
        }
    },
};

public IEnumerable<Author> GetAuthors(int skip, int take)
{
    return _storage.Skip(skip).Take(take);
}

public Author GetAuthor(int id)
{
    var author = _storage.FirstOrDefault(x => x.Id == id);
    if (author == null)
    {
        throw new
           System.Collections.Generic.KeyNotFoundException();
    }
    return author;
}

public Book GetBook(int id)
{
    var book = _storage.SelectMany(a => a.Books)
        .FirstOrDefault(x => x.Id == id);
    if (book == null)
    {
        throw new
           System.Collections.Generic.KeyNotFoundException();
    }
    return book;
}
```

```
    public Author AddAuthor(Author author)
    {
        int newId = _storage.Select(x => x.Id).Max() + 1;
        author.Id = newId;
        _storage.Add(author);
        return author;
    }
}
```

Lets also define interface, that would provide these methods via dependency injection:

```
using demo_graphql.Dto;

namespace demo_graphql.Services;

public interface IAuthorService
{
    Author AddAuthor(Author author);
    Author GetAuthor(int id);
    IEnumerable<Author> GetAuthors(int skip, int take);
    Book GetBook(int id);
}
```

Now we define a class, that would encapsulate all the graphql queries, that our api service will provide:

```
using demo_graphql.Dto;
using demo_graphql.Services;

namespace demo_graphql;

public class Query
{
    public IEnumerable<Author> GetAuthors(
        [Service] IAuthorService authorService,
        int skip,
        int take)
    {
        return authorService.GetAuthors(skip, take);
    }
```

```
    public Author GetAuthor(
        [Service] IAuthorService authorService,
        int id)
    {
        return authorService.GetAuthor(id);
    }

    public Book GetBook(
        [Service] IAuthorService authorService,
        int id)
    {
        return authorService.GetBook(id);
    }
}
```

And a class, that would provide all the possible mutations:

```
using demo_graphql.Dto;
using demo_graphql.Services;

namespace demo_graphql;

public class Mutation
{
    public Author AddAuthor(
        [Service] IAuthorService authorService,
        string name,
        string email,
        string country,
        DateTime birthDate)
    {
        return authorService.AddAuthor(new Author()
        {
            Name = name,
            Email = email,
            Country = country,
            BirthDate = birthDate
        });
    }
}
```

After that, we have to register our services, queries and mutations, in the Pro-

gram.cs class:

```
using demo_graphql;
using demo_graphql.Services;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();
// Learn more about configuring Swagger/OpenAPI at
   https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

builder.Services.AddSingleton<IAuthorService, AuthorService>();
builder.Services
    .AddGraphQLServer()
    .AddQueryType<Query>()
    .AddMutationType<Mutation>();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapGraphQL();
app.MapControllers();

app.Run();
```

Notice, that we registered our service as a singleton, so that we would always retrieve the same data throughout the runtime of the application (as we mock the database connection). We will later discuss singleton pattern.

Now, let's startup the application, and type in the address line of the browser "https://localhost:port/graphql/" (switch word "port" with the port number, that your application uses locally). You will be greeted with the Nitro greeting screen.

Nitro is a tool for developers, simplifying API creation, debugging, and collaboration. It enables effortless execution of GraphQL queries and mutations, with visual schema exploration.

Click "Create Document" and type in the following query:

```
query GetAuthor($id: Int!){
  author(id: $id){
    name
    email
    country
    birthDate
    books{
      title
    }
  }
}
```

And the variables below:

```
{
  "id": 1
}
```

The response should look like this:

```
{
  "data": {
    "author": {
      "name": "Aizek Asimov",
      "email": "aa@gmail.com",
      "country": "USA",
      "birthDate": "1984-11-24T17:24:44.865+02:00",
      "books": [
        {
          "title": "Foundation and Empire"
        },
        {
          "title": "I, Robot"
        }
      ]
```

```
    }
  }
}
```

Suppose, you do not want to retrieve the book information for the give author, as it is too much information for your application to handle and is just redundant at the moment. Simply rewrite the query in the following way:

```
query GetAuthor($id: Int!){
  author(id: $id){
    name
    email
    country
    birthDate
  }
}
```

The result of this query execution:

```
{
  "data": {
    "author": {
      "name": "Aizek Asimov",
      "email": "aa@gmail.com",
      "country": "USA",
      "birthDate": "1984-11-24T17:24:44.865+02:00"
    }
  }
}
```

We can also try query GetAuthors, that we defined in Query.cs class:

```
query GetAuthors($skip: Int!, $take: Int!){
  authors(skip: $skip, take: $take){
    name
    email
    country
    birthDate
  }
}
```

With variables:

```
{
```

```
  "skip": 0,
  "take": 10
}
```

The result of this query execution:

```
{
  "data": {
    "authors": [
      {
        "name": "Aizek Asimov",
        "email": "aa@gmail.com",
        "country": "USA",
        "birthDate": "1984-11-24T17:24:44.865+02:00"
      },
      {
        "name": "Ray Bradbury",
        "email": "rb@gmail.com",
        "country": "USA",
        "birthDate": "1974-11-24T17:24:44.865+02:00"
      },
      {
        "name": "Stanislaw Lem",
        "email": "sl@gmail.com",
        "country": "Poland",
        "birthDate": "1964-11-24T17:24:44.865+02:00"
      }
    ]
  }
}
```

Notice, that query expects the same variables, as the method GetAuthors, defined in the Query.cs class.

We can also try executing the mutation to add a new entry in the Authors list:

```
mutation AddAuthor($name: String!, $email: String!,
  $country: String!, $birthDate: DateTime!){
  addAuthor(name: $name, email: $email,
      country: $country, birthDate: $birthDate){
    name
    email
    country
    birthDate
```

```
  }
}
```

And variables:

```
{
  "name": "Carl Sagan",
  "email": "cs@gmail.com",
  "country" : "USA",
  "birthDate" : "1934-11-04T00:00:00.000Z"
}
```

After execution of this mutation, the list of authors looks like this:

```
{
  "data": {
    "authors": [
      {
        "name": "Aizek Asimov",
        "email": "aa@gmail.com",
        "country": "USA",
        "birthDate": "1984-11-24T17:24:44.865+02:00"
      },
      {
        "name": "Ray Bradbury",
        "email": "rb@gmail.com",
        "country": "USA",
        "birthDate": "1974-11-24T17:24:44.865+02:00"
      },
      {
        "name": "Stanislaw Lem",
        "email": "sl@gmail.com",
        "country": "Poland",
        "birthDate": "1964-11-24T17:24:44.865+02:00"
      },
      {
        "name": "Carl Sagan",
        "email": "cs@gmail.com",
        "country": "USA",
        "birthDate": "1934-11-04T00:00:00.000Z"
      }
    ]
  }
```

```
}
```

Chapter 6

Authentication, Authorization, JWT and OAuth2.0 .

Authentication and authorization are two critical components of application security. Together, they ensure that only the right users can access specific resources or functionalities within a system, protecting sensitive data and maintaining system integrity.

Authentication is the process of verifying the identity of a user or entity. It ensures that the individual attempting to access a system is who they claim to be.

Authorization is the process of determining what actions or resources an authenticated user is allowed to access. It controls the permissions and privileges granted to a user or entity.

Modern applications use a variety of tools and techniques to implement authentication and authorization securely:

- Session-Based Authentication: The server creates a session for the user upon login and stores it in memory. A session ID is sent to the client as a cookie.
- Token-Based Authentication: After login, the server generates a token (e.g., JWT) that the client includes in each request for authentication.
- Authorization can be implemented via middleware, Access Control Lists (ACLs) or API gateways.

JSON Web Tokens, commonly known as JWTs, are a compact and secure way of transmitting information between parties as a JSON object. This technology has become particularly popular in modern web applications due to its simplicity and robustness in enabling secure, stateless authentication and authorization.

What is JWT? JWT is an open standard (RFC 7519) that defines a way to securely transmit information as a JSON object. The token itself is self-contained, meaning it holds all the information needed for authentication, which can be verified without querying a database. This capability makes JWT highly scalable, as it reduces the server's reliance on session storage.

A JWT consists of three parts, separated by dots (.):

- Header: Contains metadata about the token, including the type of token (JWT) and the hashing algorithm used (e.g., HMAC SHA256 or RSA).
- Payload: Holds the claims, which are statements about the user or other data. Claims can be predefined, such as sub (subject) and exp (expiration), or custom, like role or email.
- Signature: Created by encoding the header and payload using a secret key or a private key, depending on the algorithm.

An example JWT might look like this:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ
SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

Decoded, it would look like this:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),

your-256-bit-secret
)
```

OAuth 2.0, short for Open Authorization 2.0, is a protocol that enables third-party applications to gain limited access to a user's resources without sharing their username and password. It provides an efficient way to implement secure delegated access, focusing on authorization rather than authentication.

Example of JWT authentication in .Net.
Add `Microsoft.AspNetCore.Authentication.JwtBearer` to demo-rest.bll.
Now add the following two models:

```
public class User
{
    public int Id { get; set; }
```

```
    public string Username { get; set; }
    public string PasswordHash { get; set; }
    public bool IsLoggedIn { get; set; }
}

public class LoginRequest
{
    public string Username { get; set; }
    public string Password { get; set; }
}
```

And a TokenService with it's interface:

```
using demo_rest.bll.Interfaces;
using Microsoft.Extensions.Configuration;
using Microsoft.IdentityModel.Tokens;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using System.Text;

namespace demo_rest.bll.Services;

public class TokenService : ITokenService
{
    private readonly IConfiguration _config;
    public TokenService(IConfiguration config)
    {
        _config = config;
    }

    public string GenerateToken(string username)
    {
        var claims = new[]
        {
            new Claim(ClaimTypes.Name, username)
        };

        var key = new
           SymmetricSecurityKey(Encoding.UTF8.GetBytes(_config["Jwt:Key"]))
        var creds = new SigningCredentials(key,
           SecurityAlgorithms.HmacSha256);
```

```
        var token = new JwtSecurityToken(
            issuer: _config["Jwt:Issuer"],
            audience: _config["Jwt:Audience"],
            claims: claims,
            expires: DateTime.UtcNow.AddHours(1),
            signingCredentials: creds);

        return new JwtSecurityTokenHandler().WriteToken(token);
    }
}


public interface ITokenService
{
    string GenerateToken(string username);
}
```

Add Account controller, that will provide endpoints for user tracking:

```
using demo_rest.bll.Interfaces;
using demo_rest.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace demo_rest.Controllers;

[Route("[controller]/[action]")]
[ApiController]
public class AccountController : ControllerBase
{
    private readonly ITokenService _tokenService;

    public AccountController(ITokenService tokenService)
    {
        _tokenService = tokenService;
    }

    private static List<User> _users = new()
    {
        new User {
            Id = 1, Username = "admin",
            PasswordHash = "password", IsLoggedIn = false }
```

```
};

[HttpPost("login")]
public IActionResult Login([FromBody] LoginRequest request)
{
    var user = _users.FirstOrDefault(u => u.Username ==
       request.Username && u.PasswordHash == request.Password);
    if (user == null)
        return Unauthorized("Invalid credentials");

    user.IsLoggedIn = true;

    var token = _tokenService.GenerateToken(user.Username);
    return Ok(new { token });
}

[HttpGet("logged-in-users")]
public IActionResult GetLoggedInUsers()
{
    var loggedIn = _users.Where(u => u.IsLoggedIn).Select(u =>
       u.Username);
    return Ok(loggedIn);
}

[HttpGet("me")]
[Authorize]
public IActionResult GetCurrentUser()
{
    var username = User.Identity?.Name;

    if (string.IsNullOrEmpty(username))
        return Unauthorized();

    var user = _users.FirstOrDefault(u => u.Username ==
       username);

    if (user == null)
        return NotFound("User not found");

    return Ok(new
    {
```

```
            user.Id,
            user.Username,
            user.IsLoggedIn
        });
    }

    [HttpPost("logout")]
    public IActionResult Logout()
    {
        var username = User.Identity?.Name;
        var user = _users.FirstOrDefault(u => u.Username ==
           username);
        if (user != null)
            user.IsLoggedIn = false;

        return Ok();
    }
}
```

Register token services in `Program.cs`:

```
using demo_rest.bll.Interfaces;
using demo_rest.bll.Services;
using demo_rest.dal;
using Microsoft.EntityFrameworkCore;
using Microsoft.IdentityModel.Tokens;
using System.Text;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();
// Learn more about configuring Swagger/OpenAPI at
   https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

builder.Services.AddDbContext<DemoDbContext>(options =>
{
    options.UseSqlServer(builder.Configuration.GetConnectionString("Default
});
```

```
builder.Services.AddTransient<IStudentService, StudentService>();
builder.Services.AddTransient<ITokenService, TokenService>();

builder.Services.AddAuthentication("Bearer")
    .AddJwtBearer("Bearer", options =>
    {
        options.TokenValidationParameters = new
           TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = builder.Configuration["Jwt:Issuer"],
            ValidAudience = builder.Configuration["Jwt:Audience"],
            IssuerSigningKey = new SymmetricSecurityKey(
                Encoding.UTF8.GetBytes(builder.Configuration["Jwt:Key"]))
        };
    });

builder.Services.AddAuthorization();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Chapter 7

WebSockets and WebHooks. Integration with WebSocket servers.

WebSockets are a communication protocol that provides full-duplex, persistent connections between a client (such as a browser or mobile app) and a server. Unlike traditional HTTP, which follows a request-response model, WebSockets allow two-way communication over a single, long-lived connection. Once established, the connection remains open, enabling continuous communication. Because it eliminates the overhead of opening and closing connections for each request, WebSockets are ideal for real-time use cases. Both client and server can send messages at any time without waiting for the other to initiate.

Common Use Cases for WebSockets include: live chat and messaging applications, dashboards, collaborative tools, IoT device monitoring.

WebHooks are a mechanism for servers to notify other systems when certain events occur. Instead of continuously polling an API to check for updates, a WebHook pushes data automatically by sending an HTTP POST request to a predefined URL.

Common Use Cases for WebHooks are: payment gateways (notify a merchant's system when a transaction is completed), CI/CD pipelines (trigger automated builds and deployments when code is pushed to a repository), CRM and marketing platforms (sync customer data when new leads are created) and notifications (send alerts to third-party apps when important events occur).

Modern applications often need both real-time communication (WebSockets) and event-driven integration (WebHooks)

WebSockets and WebHooks are complementary technologies that bring real-time interaction and event-driven automation to web applications. WebSockets enable ongoing two-way conversations between clients and servers, while WebHooks empower systems to notify one another when specific events occur. Together, they form the backbone of modern, interactive, and integrated web ecosystems.

By adopting both, developers can build applications that not only respond to user actions instantly but also seamlessly integrate with the broader digital world—meeting the expectations of today's real-time internet.

SignalR is an open-source library for ASP.NET that simplifies adding real-time web functionality to applications. It allows for server-side code to push content to connected clients instantly, enabling features like live updates, chat applications, and dashboards. SignalR handles the complexities of connection management and messaging, allowing developers to focus on building real-time features rather than low-level communication protocols.

SignalR uses WebSockets as the preferred transport when available, but it also supports older transports like Server-Sent Events and long polling for older browsers.

Example of websocket in .net web api.

In `Program.cs`:

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddSignalR(); // Add SignalR service

var app = builder.Build();

app.MapControllers();
app.MapHub<ChatHub>("/chatHub"); // Map SignalR hub endpoint

app.Run();
```

Create a Hub: a Hub is where you define server-side methods clients can call, and vice versa.

```
using Microsoft.AspNetCore.SignalR;

namespace WebSocketDemo.Hubs
{
    public class ChatHub : Hub
    {
        public async Task SendMessage(string user, string message)
        {
            // Broadcast message to all connected clients
            await Clients.All.SendAsync("ReceiveMessage", user,
               message);
        }
    }
}
```

JavaScript Client (HTML Page). You'll need the SignalR JS client library. Add via CDN:

```
<!DOCTYPE html>
<html>
<head>
  <title>SignalR Chat Demo</title>
  <script
     src="https://cdnjs.cloudflare.com/ajax/libs/microsoft-signalr/8.0.0/si
</head>
<body>
  <h1>SignalR Chat</h1>

  <input type="text" id="userInput" placeholder="Your name..." />
  <input type="text" id="messageInput" placeholder="Type a
     message..." />
  <button onclick="sendMessage()">Send</button>

  <ul id="messages"></ul>

  <script>
    // Create connection
    const connection = new signalR.HubConnectionBuilder()
        .withUrl("http://localhost:5000/chatHub") // Match server
           endpoint
        .build();

    // Receive message from server
    connection.on("ReceiveMessage", (user, message) => {
      const li = document.createElement("li");
      li.textContent = `${user}: ${message}`;
      document.getElementById("messages").appendChild(li);
    });

    // Start connection
    connection.start().then(() => {
      console.log("Connected to SignalR hub");
    }).catch(err => console.error(err));

    // Send message
    function sendMessage() {
      const user = document.getElementById("userInput").value;
      const message = document.getElementById("messageInput").value;
      connection.invoke("SendMessage", user, message)
```

```
            .catch(err => console.error(err));
    }
  </script>
</body>
</html>
```

Start your .NET Web API app. After that, open the HTML page in two different browser tabs. Enter a username + message, and click Send. Both tabs will see the broadcasted message instantly.

Chapter 8

XSS, CSRF, CORS, security and OWASP top 10.

Security is a fundamental pillar of web development. Every application that is deployed online becomes a potential target for malicious actors. Data breaches, unauthorized access, and compromised systems can result not only in financial losses but also in the erosion of user trust, which is often far more difficult to repair. This is why developers, architects, and organizations must treat security as an integral part of their development process rather than something to add once the system is already in place.

One of the most common web vulnerabilities is Cross-Site Scripting, or XSS. This occurs when attackers are able to inject malicious scripts into otherwise trusted websites. These scripts can run in the browser of unsuspecting users and can steal session tokens, manipulate the DOM, or redirect users to harmful websites. The root of the problem often lies in insufficient sanitization or encoding of user input. Although modern frameworks provide better protection mechanisms, the risk of XSS still exists whenever data flows from user input directly into the page without proper handling.

Closely related to XSS is another significant vulnerability known as Cross-Site Request Forgery, or CSRF. Unlike XSS, which relies on injecting malicious content, CSRF manipulates authenticated users into performing unintended actions on a web application. For example, a logged-in user could be tricked into clicking a malicious link or loading a hidden form that triggers a money transfer or changes account details. The attack works because the browser automatically attaches authentication cookies, and without proper validation, the server assumes the request is legitimate. Preventing CSRF often requires the use of anti-forgery tokens, SameSite cookies, and explicit verification mechanisms to ensure that the request truly originates from the intended user interaction.

Another concept that frequently appears in discussions of web security is Cross-Origin Resource Sharing, or CORS. By default, browsers enforce a same-origin policy, meaning that scripts running on one domain cannot access resources from another domain. While this policy is essential for preventing data leaks, modern ap-

plications often require communication across different services, domains, or APIs. CORS was introduced as a controlled way to relax the same-origin policy, allowing trusted domains to share resources. Misconfiguration of CORS, however, can be extremely dangerous. If a server allows requests from any origin without restrictions, it essentially opens the door to malicious sites extracting sensitive data from unsuspecting users.

To understand these risks in a broader context, developers often turn to the OWASP Top 10, a regularly updated list of the most critical web application security risks. The OWASP Top 10 acts as a guide to the most common and impactful vulnerabilities, including injection flaws, broken authentication, insecure design, sensitive data exposure, and insufficient logging and monitoring. Each item on the list is not only a technical challenge but also a reminder of the need for secure thinking in the development lifecycle. Security is never just about fixing individual vulnerabilities but about building a culture where threats are anticipated, risks are evaluated, and defenses are designed from the ground up.

The importance of security in web development becomes even clearer when one considers the consequences of neglecting it. Beyond financial damages and regulatory penalties, insecure applications can compromise personal data such as medical information, banking details, or private communications. This not only violates legal and ethical responsibilities but also causes long-lasting reputational harm. By applying security best practices, staying updated with the OWASP Top 10, and being aware of threats such as XSS, CSRF, and misconfigured CORS, developers create safer applications that protect both businesses and their users.

- A01:2021-Broken Access Control moves up from the fifth position; 94% of applications were tested for some form of broken access control. The 34 Common Weakness Enumerations (CWEs) mapped to Broken Access Control had more occurrences in applications than any other category.

- A02:2021-Cryptographic Failures shifts up one position to #2, previously known as Sensitive Data Exposure, which was broad symptom rather than a root cause. The renewed focus here is on failures related to cryptography which often leads to sensitive data exposure or system compromise.

- A03:2021-Injection slides down to the third position. 94% of the applications were tested for some form of injection, and the 33 CWEs mapped into this category have the second most occurrences in applications. Cross-sit Scripting is now part of this category in this edition.

- A04:2021-Insecure Design is a new cate gory for 2021, with a focus on risks related to design flaws. If we genuinely want to ПїSmove leftПїS as an industry, it calls for more use of threat modeling, secure design patterns and principles, and reference architectures.

- A05:2021-Security Misconfiguration moves up from #6 in the previous edition; 90% of applications were tested for some form of misconfiguration. With more shifts into highly configurable software, itпїЅs not surprising to see this category move up. The former category for XML External Entities (XXE) is now part of this category.

- A06:2021-Vulnerable and Outdated Components was previously titled Using Components with Known Vulnerabilities and is #2 in the Top 10 community survey, but also had enough data to make the Top 10 via data analysis. This category moves up from #9 in 2017 and is a known issue that we struggle to test and assess risk. It is the only category not to have any Common Vulnerability and Exposures (CVEs) mapped to the included CWEs, so a default exploit and impact weights of 5.0 are factored into their scores.

- A07:2021-Identification and Authentication Failures was previously Broken Authentication and is sliding down from the second position, and now includes CWEs that are more related to identification failures. This category is still an integral part of the Top 10, but the increased availability of standardized frameworks seems to be helping.

- A08:2021-Software and Data Integrity Failures is a new category for 2021, focusing on making assumptions related to software updates, critical data, and CI/CD pipelines without verifying integrity. One of the highest weighted impacts from Common Vulnerability and Exposures/Common Vulnerability Scoring System (CVE/CVSS) data mapped to the 10 CWEs in this category. Insecure Deserialization from 2017 is now a part of this larger category.

- A09:2021-Security Logging and Monitoring Failures was previously Insufficient Logging & Monitoring and is added from the industry survey (#3), moving up from #10 previously. This category is expanded to include more types of failures, is challenging to test for, and isnпїЅt well represented in the CVE/CVSS data. However, failures in this category can directly impact visibility, incident alerting, and forensics.

- A10:2021-Server-Side Request Forgery is added from the Top 10 community survey (#1). The data shows a relatively low incidence rate with above average testing coverage, along with above-average ratings for Exploit and Impact potential. This category represents the scenario where the security community members are telling us this is important, even though itпїЅs not illustrated in the data at this time.

Example of XSS handling in .net

In a .NET Web API, you donпїЅt usually have direct HTML rendering (like in MVC views or Razor Pages), but XSS can still creep in if your API sends unsafe

data that is later rendered in a browser (e.g., in a React/Angular UI or directly in HTML).

The key principle is: sanitize or encode any untrusted input before returning or displaying it.

HereпїЅs a simple example of unsafe code:

```
// Model
public class CommentDto
{
    public string Author { get; set; }
    public string Message { get; set; }
}

// Controller
[ApiController]
[Route("api/[controller]")]
public class CommentsController : ControllerBase
{
    private static readonly List<CommentDto> _comments = new();

    [HttpPost]
    public IActionResult PostComment([FromBody] CommentDto comment)
    {
        _comments.Add(comment);
        return Ok(comment); // returns raw user input, can contain
           <script>
    }

    [HttpGet]
    public IActionResult GetComments()
    {
        return Ok(_comments);
    }
}


// Malicious user post:
{
  "author": "Hacker",
  "message": "<script>alert('XSS!')</script>"
}
```

And your frontend renders it directly, youпїЅll get a popup script execution пїЅ classic XSS.

We sanitize or encode the content before returning it:

```
using System.Web;

[ApiController]
[Route("api/[controller]")]
public class CommentsController : ControllerBase
{
    private static readonly List<CommentDto> _comments = new();

    [HttpPost]
    public IActionResult PostComment([FromBody] CommentDto comment)
    {
        // Encode before storing/returning
        var safeComment = new CommentDto
        {
            Author = HttpUtility.HtmlEncode(comment.Author),
            Message = HttpUtility.HtmlEncode(comment.Message)
        };

        _comments.Add(safeComment);
        return Ok(safeComment);
    }

    [HttpGet]
    public IActionResult GetComments()
    {
        return Ok(_comments);
    }
}
```

In this case, for the malicious input the API will return:

```
{
  "author": "Hacker",
  "message": "&lt;script&gt;alert('XSS!')&lt;/script&gt;"
}
```

Example of CSRF handling in .net

Unlike XSS, CSRF is not about injecting malicious scripts, but about tricking a victimпїЅs browser into making an authenticated request to your API without their

consent. For example: if your Web API has an endpoint /api/account/transfer and the victim is logged in, a malicious site could cause their browser to unknowingly send a request to that endpoint.

Suppose you have a controller like this:

```
[ApiController]
[Route("api/[controller]")]
public class AccountController : ControllerBase
{
    [HttpPost("transfer")]
    public IActionResult Transfer([FromBody] TransferRequest
       request)
    {
        // Imagine this transfers money between accounts
        return Ok(new { status = "success", amount = request.Amount
           });
    }
}

public class TransferRequest
{
    public string ToAccount { get; set; }
    public decimal Amount { get; set; }
}
```

If your app uses cookie-based authentication (common with Identity or Azure AD B2C), then the browser will automatically send cookies with this request пїЅ even if it originates from a malicious site. ThatпїЅs the CSRF attack.

If youпїЅre using ASP.NET Core with cookie auth, you can require an anti-forgery token with every request that modifies data.

```
// Program.cs:

builder.Services.AddControllersWithViews(options =>
{
    options.Filters.Add(new
       AutoValidateAntiforgeryTokenAttribute());
});
builder.Services.AddAntiforgery(options =>
{
    options.HeaderName = "X-CSRF-TOKEN"; // wепїЅll expect this
       header
});
```

```
// Controller
[HttpPost("transfer")]
[ValidateAntiForgeryToken]
public IActionResult Transfer([FromBody] TransferRequest request)
{
    return Ok(new { status = "success", amount = request.Amount });
}
```

On the frontend:

```
fetch("/api/account/transfer", {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
    "X-CSRF-TOKEN": tokenFromServer // usually injected into page
  },
  body: JSON.stringify({ toAccount: "12345", amount: 100 })
});
```

Example of CORS handling in .net

By default, browsers enforce the same-origin policy. A frontend running at https://app.example.com can freely call APIs hosted on the same origin (https://app.examp But if it tries to call an API at https://api.otherdomain.com, the browser will block the request unless the API explicitly allows it.

This is where CORS comes in пїS itпїSs the server telling the browser: пїSYes, I trust requests from this origin. You may allow them.пїS Without proper CORS handling your legitimate frontend might not be able to reach your API, and if configured too loosely, your API could be open to abuse from untrusted origins.

Example of CORS Setup in .NET Web API, in Program.cs:

```
var builder = WebApplication.CreateBuilder(args);

// Define named CORS policy
builder.Services.AddCors(options =>
{
    options.AddPolicy("AllowFrontendApp", policy =>
    {
        policy.WithOrigins("https://my-frontend.com") // allow only
           this origin
            .AllowAnyHeader()
            .AllowAnyMethod();
```

```
    });
});

builder.Services.AddControllers();
var app = builder.Build();

// Apply CORS middleware
app.UseCors("AllowFrontendApp");

app.MapControllers();
app.Run();
```

Chapter 9

Web performance optimization, lazy loading.

Web performance optimization is a crucial aspect of modern development because the speed and responsiveness of a website directly influence how users perceive it and whether they choose to stay or leave. One of the most effective techniques for improving performance is lazy loading. The idea behind lazy loading is simple: instead of forcing the browser to load every single asset or piece of content immediately when a page opens, resources are loaded only when they are needed. This reduces the initial loading time and improves the overall user experience.

Consider a webpage that contains dozens of high-resolution images. If all of them are downloaded at once, the page will take longer to render, especially on slower networks or mobile devices. This can frustrate users who might leave before even seeing the content. With lazy loading, only the images visible in the user's viewport are fetched, while others are loaded later as the user scrolls down. This means the page feels faster and more responsive right from the beginning.

Lazy loading is also beneficial for conserving bandwidth and server resources. For example, on a news site, a user may only read the first few articles and never scroll further. Without lazy loading, all the images and media for the entire page are downloaded unnecessarily, wasting both client and server capacity. By deferring these requests, the system becomes more efficient and scales better under heavy traffic.

In practice, lazy loading has become particularly important in the context of mobile browsing, where network conditions are often less reliable. A fast and responsive mobile experience can make the difference between keeping a user engaged or losing them to a competitor's site. That is why lazy loading is not just a technical trick but a user-centered strategy that aligns with the broader goal of web performance optimization.

Example.

Imagine you have an endpoint that returns product data including images. Without optimization, you might return everything at once: product details, specifications, and all images in full resolution. This is wasteful, especially if the user only

needs the list of products to begin with.

A lazy-loading approach would be to create two endpoints. The first one delivers lightweight product data such as name, price, and a small thumbnail. The second endpoint provides detailed product information, including high-resolution images, but it is only called when the user requests to view the product details.

```
// Lightweight endpoint
[HttpGet("products")]
public IActionResult GetProducts()
{
    var products = _context.Products
        .Select(p => new {
            p.Id,
            p.Name,
            p.Price,
            ThumbnailUrl = p.Images.FirstOrDefault(i =>
               i.IsThumbnail).Url
        })
        .ToList();

    return Ok(products);
}

// Detailed endpoint
[HttpGet("products/{id}")]
public IActionResult GetProductDetails(int id)
{
    var product = _context.Products
        .Where(p => p.Id == id)
        .Select(p => new {
            p.Id,
            p.Name,
            p.Price,
            p.Description,
            Images = p.Images.Select(i => i.Url)
        })
        .FirstOrDefault();

    return Ok(product);
}
```

On the frontend, lazy loading ensures that large assets like images are loaded

only when they become visible. Modern browsers support this directly with the loading="lazy" attribute:

```
<img src="product-thumbnail.jpg"
     data-full="product-large.jpg"
     loading="lazy"
     alt="Product Image">
```

Or with JavaScript using an Intersection Observer to load images only when they enter the viewport:

```
const images = document.querySelectorAll('img[data-src]');

const lazyLoad = (entry, observer) => {
  entry.forEach(e => {
    if (e.isIntersecting) {
      const img = e.target;
      img.src = img.dataset.src;
      observer.unobserve(img);
    }
  });
};

const observer = new IntersectionObserver(lazyLoad, { threshold:
   0.1 });

images.forEach(img => observer.observe(img));
```

Chapter 10

Architecture patterns, GoF, SOLID.

What are SOLID Design Principles?

SOLID is an acronym that stands for:

- Single Responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

In the coming sections, weпїSll look at what each of those principles means in detail.

The SOLID design principles are a subcategory of many principles introduced by the American computer scientist and instructor, Robert C. Martin (A.K.A Uncle Bob) in a 2000 paper.

Following these principles can result in a very large codebase for a software system. But in the long run, the main aim of the principles is never defeated. That is, helping software developers make changes to their code without causing any major issues.

The Single Responsibility Principle (SRP)

The single responsibility principle states that a class, module, or function should have only one reason to change, meaning it should do one thing.

For example, a class that shows the name of an animal should not be the same class that displays the kind of sound it makes and how it feeds.

Example:

```
public class BadBook {
    private String name;
    private String author;
    private String text;
    //...

    void printTextToConsole(){
        // our code for formatting and printing the text
    }
}
```

This code violates the single responsibility principle we outlined earlier.

To fix our mess, we should implement a separate class that deals only with printing our texts:

```
public class BookPrinter {

    // methods for outputting text
    void printTextToConsole(String text){
        //our code for formatting and printing the text
    }

    void printTextToAnotherMedium(String text){
        // code for writing to any other location..
    }
}
```

The Open-Closed Principle (OCP)

The open-closed principle states that classes, modules, and functions should be open for extension but closed for modification.

This principle might seem to contradict itself, but you can still make sense of it in code. It means you should be able to extend the functionality of a class, module, or function by adding more code without modifying the existing code.

Example:

```
public class Guitar {

    private String make;
    private String model;
    private int volume;
```

```
    //Constructors, getters & setters
}
```

What if we later decide the Guitar is a little boring and could use a cool flame pattern to make it look more rock and roll.

At this point, it might be tempting to just open up the Guitar class and add a flame pattern пїS but who knows what errors that might throw up in our application.

Instead, letпїSs stick to the open-closed principle and simply extend our Guitar class:

```
public class SuperCoolGuitarWithFlames extends Guitar {

    private String flameColor;

    //constructor, getters + setters
}
```

The Liskov Substitution Principle (LSP)

The Liskov substitution principle is one of the most important principles to adhere to in object-oriented programming (OOP). It was introduced by the computer scientist Barbara Liskov in 1987 in a paper she co-authored with Jeannette Wing.

The principle states that child classes or subclasses must be substitutable for their parent classes or super classes. In other words, the child class must be able to replace the parent class. This has the advantage of letting you know what to expect from your code.

Example:

```
public interface Car {

    void turnOnEngine();
    void accelerate();
}
```

Above, we define a simple Car interface with a couple of methods that all cars should be able to fulfill: turning on the engine and accelerating forward.

But suppose, we have an electric car:

```
public class ElectricCar implements Car {

    public void turnOnEngine() {
        throw new AssertionError("I don't have an engine!");
    }
```

```
    public void accelerate() {
        //this acceleration is crazy!
    }
```

By throwing a car without an engine into the mix, we are inherently changing the behavior of our program. This is a blatant violation of Liskov substitution and is a bit harder to fix than our previous two principles.

One possible solution would be to rework our model into interfaces that take into account the engine-less state of our Car.

```
public interface EnginelessCar {

    void accelerate();
}
```

The Interface Segregation Principle (ISP)

The interface segregation principle states that clients should not be forced to implement interfaces or methods they do not use.

More specifically, the ISP suggests that software developers should break down large interfaces into smaller, more specific ones, so that clients only need to depend on the interfaces that are relevant to them. This can make the codebase easier to maintain.

This principle is fairly similar to the single responsibility principle (SRP). But itпїЅs not just about a single interface doing only one thing пїЅ itпїЅs about breaking the whole codebase into multiple interfaces or components.

Think about this as the same thing you do while working with frontend frameworks and libraries like React, Svelte, and Vue. You usually break down the codebase into components you only bring in when needed.

This means you create individual components that have functionality specific to them. The component responsible for implementing scroll to the top, for example, will not be the one to switch between light and dark, and so on.

Example:

```
public interface BearKeeper {
    void washTheBear();
    void feedTheBear();
    void petTheBear();
}
```

As avid zookeepers, wепїЅre more than happy to wash and feed our beloved bears. But wепїЅre all too aware of the dangers of petting them. Unfortunately, our

interface is rather large, and we have no choice but to implement the code to pet the bear.

LetпїЅs fix this by splitting our large interface into three separate ones:

```
public interface BearCleaner {
    void washTheBear();
}

public interface BearFeeder {
    void feedTheBear();
}

public interface BearPetter {
    void petTheBear();
}
```

The Dependency Inversion Principle (DIP)

The dependency inversion principle is about decoupling software modules. That is, making them as separate from one another as possible.

The principle states that high-level modules should not depend on low-level modules. Instead, they should both depend on abstractions. Additionally, abstractions should not depend on details, but details should depend on abstractions.

In simpler terms, this means instead of writing code that relies on specific details of how lower-level code works, you should write code that depends on more general abstractions that can be implemented in different ways.

This makes it easier to change the lower-level code without having to change the higher-level code.

Example:

```
public class Windows {

    private StandardKeyboard keyboard;
    private Monitor monitor;

    public Windows() {
        monitor = new Monitor();
        keyboard = new StandardKeyboard();
    }

}
```

Not only does this make our Windows class hard to test, but weпїЅve also lost the ability to switch out our StandardKeyboard class with a different one should the need arise. And weпїЅre stuck with our Monitor class too.

LetпїЅs decouple our machine from the StandardKeyboard by adding a more general Keyboard interface and using this in our class:

```
public interface Keyboard { }

public class StandardKeyboard implements Keyboard { }

public class Windows{

    private Keyboard keyboard;
    private Monitor monitor;

    public Windows(Keyboard keyboard, Monitor monitor) {
        this.keyboard = keyboard;
        this.monitor = monitor;
    }
}
```

GoF patterns.

Gangs of Four Design Patterns is the collection of 23 design patterns from the book пїЅDesign Patterns: Elements of Reusable Object-Oriented SoftwareпїЅ.

This book was first published in 1994 and itпїЅs one of the most popular books to learn design patterns. The book was authored by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. It got nicknamed as Gangs of Four design patterns because of four authors. Furthermore, it got a shorter name as пїЅGoF Design PatternsпїЅ.

GoF Design Patterns are divided into three categories:

- Creational: The design patterns that deal with the creation of an object (Singleton, Factory Method, Abstract Factory, Builder and Prototype).
- Structural: The design patterns in this category deals with the class structure such as Inheritance and Composition (Adapter, Bridge, Composite, Decorator, Facade, Flyweight and Proxy).
- Behavioral: This type of design patterns provide solution for the better interaction between objects, how to provide lose coupling, and flexibility to extend easily in future.

Singleton

The Singleton Pattern is one of the most commonly used design patterns in software development. It ensures that a class has only one instance while providing a global access point to that instance. This pattern is particularly useful for managing shared resources such as configuration settings, logging, or database connections.

The Singleton Pattern ensures single instance (only one instance of the class exists throughout the applicationпїЅs lifecycle) and global access (provides a centralized point of access to the single instance).

Advantages: controlled access, resource efficiency and global point of access.

Common Use Cases:

- Logging Services: Ensure that all parts of an application write to the same log file.
- Configuration Management: Store and manage application settings.
- Thread Pools: Manage a pool of threads with a single controller.
- Database Connections: Ensure only one connection object interacts with the database.

Example.

```
public class Singleton
{
    // Static variable to hold the single instance
    private static Singleton _instance;
    private static readonly object _lock = new object();

    // Private constructor to prevent instantiation from other
       classes
    private Singleton()
    {
        Console.WriteLine("Singleton instance created.");
    }

    // Public method to provide access to the single instance
    public static Singleton Instance
    {
        get
        {
            // Ensure thread safety
            if (_instance == null)
```

```
            {
                lock (_lock)
                {
                    if (_instance == null)
                    {
                        _instance = new Singleton();
                    }
                }
            }
            return _instance;
        }
    }

    // Example method to demonstrate functionality
    public void ShowMessage(string message)
    {
        Console.WriteLine($"Message from Singleton: {message}");
    }
}

// Example usage
class Program
{
    static void Main(string[] args)
    {
        Singleton instance1 = Singleton.Instance;
        Singleton instance2 = Singleton.Instance;

        instance1.ShowMessage("Hello, Singleton!");

        // Verify that both instances are the same
        Console.WriteLine($"Are instances equal?
           {ReferenceEquals(instance1, instance2)}");
    }
}
```

Factory Method

The Factory Method Pattern is a creational design pattern that provides an interface for creating objects in a superclass, while allowing subclasses to alter the type of objects that will be created. It promotes loose coupling by delegating the instantiation process to child classes instead of instantiating concrete classes directly.

The Factory Method Pattern is used when:

- The exact type of the object to be created is determined at runtime.
- You want to provide a common interface for creating objects while allowing subclasses to decide which specific object to instantiate.

This pattern is especially useful for frameworks where components need to work with objects that adhere to a specific interface but whose concrete classes may vary.

Example.

```
using System;

namespace FactoryMethodPattern
{
    // Product Interface
    public interface IVehicle
    {
        void Drive();
    }

    // Concrete Product: Car
    public class Car : IVehicle
    {
        public void Drive()
        {
            Console.WriteLine("Driving a car.");
        }
    }

    // Concrete Product: Bike
    public class Bike : IVehicle
    {
        public void Drive()
        {
            Console.WriteLine("Riding a bike.");
        }
    }

    // Creator (Abstract Class)
    public abstract class VehicleFactory
    {
        // Factory Method
```

```
    public abstract IVehicle CreateVehicle();

    public void TestDrive()
    {
        IVehicle vehicle = CreateVehicle();
        vehicle.Drive();
    }
}

// Concrete Creator: Car Factory
public class CarFactory : VehicleFactory
{
    public override IVehicle CreateVehicle()
    {
        return new Car();
    }
}

// Concrete Creator: Bike Factory
public class BikeFactory : VehicleFactory
{
    public override IVehicle CreateVehicle()
    {
        return new Bike();
    }
}

// Client Code
class Program
{
    static void Main(string[] args)
    {
        // Use the Car Factory
        VehicleFactory carFactory = new CarFactory();
        carFactory.TestDrive();

        // Use the Bike Factory
        VehicleFactory bikeFactory = new BikeFactory();
        bikeFactory.TestDrive();
    }
}
```

```
}
```

Abstract Factory

The Abstract Factory Pattern is a creational design pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes. It is particularly useful when a system needs to be independent of the way its objects are created or when multiple families of objects are required.

The Abstract Factory Pattern encapsulates a group of factories that share a common theme. Each factory in this pattern can create multiple types of objects that are designed to work together.

Key characteristics of abstract factory pattern: encapsulation of object creation, families of related objects, abstract interface and decoupling.

Use Cases for Abstract Factory Pattern

- Cross-Platform Applications: Creating UI components for different operating systems (e.g., Windows, macOS, Linux).
- Product Families: When different configurations of a product are needed, such as themes or skins.
- Dependency Management: Managing dependencies among objects in complex systems.

Structure of the Abstract Factory pattern:

- Abstract Factory: Defines the interface for creating abstract products.
- Concrete Factory: Implements the creation of specific products.
- Abstract Product: Declares interfaces for a set of related products.
- Concrete Product: Implements the abstract product interface.
- Client: Uses the abstract factory to create objects and does not need to know the specific implementations.

Example.

```
// Abstract product for Button
public interface IButton
{
    void Render();
}

// Abstract product for Textbox
```

```
public interface ITextbox
{
    void Render();
}

// Windows-specific Button
public class WindowsButton : IButton
{
    public void Render()
    {
        Console.WriteLine("Rendering Windows Button");
    }
}

// Mac-specific Button
public class MacButton : IButton
{
    public void Render()
    {
        Console.WriteLine("Rendering Mac Button");
    }
}

// Windows-specific Textbox
public class WindowsTextbox : ITextbox
{
    public void Render()
    {
        Console.WriteLine("Rendering Windows Textbox");
    }
}

// Mac-specific Textbox
public class MacTextbox : ITextbox
{
    public void Render()
    {
        Console.WriteLine("Rendering Mac Textbox");
    }
}
```

```
// Abstract factory interface
public interface IUIFactory
{
    IButton CreateButton();
    ITextbox CreateTextbox();
}


// Factory for Windows UI components
public class WindowsUIFactory : IUIFactory
{
    public IButton CreateButton()
    {
        return new WindowsButton();
    }

    public ITextbox CreateTextbox()
    {
        return new WindowsTextbox();
    }
}

// Factory for Mac UI components
public class MacUIFactory : IUIFactory
{
    public IButton CreateButton()
    {
        return new MacButton();
    }

    public ITextbox CreateTextbox()
    {
        return new MacTextbox();
    }
}


public class Application
{
    private readonly IButton _button;
    private readonly ITextbox _textbox;
```

```
    public Application(IUIFactory factory)
    {
        _button = factory.CreateButton();
        _textbox = factory.CreateTextbox();
    }

    public void RenderUI()
    {
        _button.Render();
        _textbox.Render();
    }
}

// Example usage
public class Program
{
    public static void Main(string[] args)
    {
        Console.WriteLine("Choose Platform (Windows/Mac): ");
        string platform = Console.ReadLine();

        IUIFactory factory;

        if (platform?.ToLower() == "windows")
        {
            factory = new WindowsUIFactory();
        }
        else
        {
            factory = new MacUIFactory();
        }

        var app = new Application(factory);
        app.RenderUI();
    }
```

Builder

The Builder Pattern is a creational design pattern that provides a flexible way to construct complex objects step-by-step. Unlike other creational patterns, the

Builder Pattern separates the construction process from the representation, allowing the same construction process to produce different representations of an object.

This pattern is particularly useful when constructing an object requires multiple steps, configurations, or when objects are immutable.

Key characteristics of the builder pattern: step-by-step construction, separation of concerns, immutability and multiple representations.

Use Cases for Builder Pattern:

- Complex Objects: Constructing objects that have many optional or mandatory attributes.
- Immutable Objects: Building objects that cannot be modified after creation (e.g., configuration objects).
- Different Representations: Constructing objects with varying representations or formats.

Structure of the builder pattern:

- Builder: Defines the interface for building parts of the product.
- Concrete Builder: Implements the builder interface and constructs specific representations of the product.
- Product: Represents the complex object being built.
- Director (Optional): Encapsulates the construction steps, delegating the building process to the builder.
- Client: Uses the builder to construct the object.

Example.

```
class Client
{
   void Main()
   {
       Builder builder = new ConcreteBuilder();
       Director director = new Director(builder);
       director.Construct();
       Product product = builder.GetResult();
   }
}

class Director
{
```

```
    Builder builder;

    public Director(Builder builder)
    {
        this.builder = builder;
    }

    public void Construct()
    {
        builder.BuildPartA();
        builder.BuildPartB();
        builder.BuildPartC();
    }
}



abstract class Builder

{
    public abstract void BuildPartA();

    public abstract void BuildPartB();

    public abstract void BuildPartC();

    public abstract Product GetResult();
}



class Product
{
    List<object> parts = new List<object>();

    public void Add(string part)
    {
        parts.Add(part);
    }
}
```

```
class ConcreteBuilder : Builder
{

   Product product = new Product();

   public override void BuildPartA()
   {
       product.Add("Part A");
   }

   public override void BuildPartB()
   {
       product.Add("Part B");
   }

   public override void BuildPartC()
   {
       product.Add("Part C");
   }

   public override Product GetResult()
   {
       return product;
   }
}
```

Prototype

The Prototype Pattern is a creational design pattern that allows cloning or copying existing objects instead of creating new ones from scratch. This approach is especially useful when the creation of an object is expensive or complex. By copying an existing object, the Prototype Pattern provides a fast and efficient way to duplicate objects with the same or similar properties.

The pattern requires objects to provide a method for creating their own copies. This cloning mechanism can either be a shallow copy or a deep copy, depending on the desired behavior.

Key characteristics of prototype pattern: object cloning, decoupling, customization and performance.

Use cases:

- Resource-Intensive Object Creation: When creating an object from scratch is costly in terms of time or resources.
- Dynamic Object Composition: When objects need to be created at runtime with varying configurations.
- State Preservation: Cloning objects while maintaining their state or copying part of it.

Structure of the prototype pattern:

- Prototype: An interface or abstract class declaring a Clone() method.
- Concrete Prototype: A class implementing the Clone() method to duplicate itself.
- Client: Requests a new object by cloning an existing prototype.

Example.

```
public interface ICloneablePrototype
{
    ICloneablePrototype Clone();
}

// Concrete Prototype: Word Document
public class WordDocument : ICloneablePrototype
{
    public string Title { get; set; }
    public string Content { get; set; }
    public List<string> Authors { get; set; } = new List<string>();

    // Implementing Clone method
    public ICloneablePrototype Clone()
    {
        // Deep Copy
        return new WordDocument
        {
            Title = this.Title,
            Content = this.Content,
            Authors = new List<string>(this.Authors) // Cloning the
               list
        };
    }
```

```
    public override string ToString()
    {
        return $"WordDocument: Title = {Title}, Content =
           {Content}, Authors = {string.Join(", ", Authors)}";
    }
}

// Concrete Prototype: PDF Document
public class PdfDocument : ICloneablePrototype
{
    public string Title { get; set; }
    public int Pages { get; set; }

    // Implementing Clone method
    public ICloneablePrototype Clone()
    {
        // Shallow Copy
        return (PdfDocument)this.MemberwiseClone();
    }

    public override string ToString()
    {
        return $"PdfDocument: Title = {Title}, Pages = {Pages}";
    }
}


public class Program
{
    public static void Main(string[] args)
    {
        // Create a Word Document Prototype
        var originalWordDoc = new WordDocument
        {
            Title = "Original Word Document",
            Content = "This is the content of the original
               document.",
            Authors = new List<string> { "Author1", "Author2" }
        };

        // Clone the Word Document
```

```
        var clonedWordDoc = (WordDocument)originalWordDoc.Clone();
        clonedWordDoc.Title = "Cloned Word Document";
        clonedWordDoc.Authors.Add("Author3");

        Console.WriteLine("Original Word Document:");
        Console.WriteLine(originalWordDoc);
        Console.WriteLine("\nCloned Word Document:");
        Console.WriteLine(clonedWordDoc);

        // Create a PDF Document Prototype
        var originalPdfDoc = new PdfDocument
        {
            Title = "Original PDF Document",
            Pages = 10
        };

        // Clone the PDF Document
        var clonedPdfDoc = (PdfDocument)originalPdfDoc.Clone();
        clonedPdfDoc.Title = "Cloned PDF Document";

        Console.WriteLine("\nOriginal PDF Document:");
        Console.WriteLine(originalPdfDoc);
        Console.WriteLine("\nCloned PDF Document:");
        Console.WriteLine(clonedPdfDoc);
    }
}
```

Adapter

The Adapter Pattern is a structural design pattern that acts as a bridge between two incompatible interfaces. It allows objects with incompatible interfaces to work together by wrapping an interface around an existing class. The Adapter translates the interface of a class into another interface clients expect.

This pattern is particularly useful in scenarios where you want to integrate an existing class into a system but cannot modify its interface. By using the Adapter, you decouple the implementation details of the existing class from the system.

When to Use:

- When you want to use an existing class but its interface does not match the requirements.
- To create a reusable wrapper that can work with different classes implementing similar functionality.

- When integrating third-party libraries into your system.

Key concepts:

- Target Interface: Defines the domain-specific interface that the client uses.
- Adapter: A class that implements the Target interface and translates requests to the Adaptee.
- Adaptee: An existing class with an incompatible interface.
- Client: The entity that interacts with the Target interface.

Example.

```
// This interface represents the desired functionality expected by
   the client.
public interface IPaymentProcessor
{
    void ExecuteTransaction(decimal amount);
}

// This is the class with the incompatible interface.
public class ThirdPartyPaymentService
{
    public void ProcessPayment(decimal paymentAmount)
    {
        Console.WriteLine($"Payment of {paymentAmount:C} processed
           using ThirdPartyPaymentService.");
    }
}

// The adapter implements the IPaymentProcessor interface and
   translates requests to the ThirdPartyPaymentService.
public class PaymentAdapter : IPaymentProcessor
{
    private readonly ThirdPartyPaymentService
       _thirdPartyPaymentService;

    public PaymentAdapter(ThirdPartyPaymentService
       thirdPartyPaymentService)
    {
        _thirdPartyPaymentService = thirdPartyPaymentService;
    }
```

```
    public void ExecuteTransaction(decimal amount)
    {
        // Translate the method call to the Adaptee's method
        _thirdPartyPaymentService.ProcessPayment(amount);
    }
}

// HereпїЅs how a client interacts with the adapter:
class Program
{
    static void Main(string[] args)
    {
        // Existing third-party service
        ThirdPartyPaymentService thirdPartyService = new
           ThirdPartyPaymentService();

        // Create an adapter to bridge the incompatibility
        IPaymentProcessor paymentProcessor = new
           PaymentAdapter(thirdPartyService);

        // Client interacts with the adapter, unaware of the
           adaptee's implementation
        paymentProcessor.ExecuteTransaction(150.75M);

        Console.ReadKey();
    }
}
```

Bridge

The Bridge Pattern is a structural design pattern that decouples an abstraction from its implementation, allowing the two to vary independently. This pattern is particularly useful when you want to avoid a permanent binding between an abstraction and its implementation, enabling flexibility and extensibility in your code.

The Bridge Pattern involves splitting a class into two parts: Abstraction and Implementation. These parts are connected through a "bridge" and can evolve independently. This is especially helpful in scenarios where a system has multiple dimensions of variability.

Key Components

- Abstraction: The high-level interface or abstract class that defines the core functionality. It contains a reference to the implementer.
- Refined Abstraction: A more specific implementation of the abstraction.
- Implementer: The interface that defines the implementation's low-level operations.
- Concrete Implementer: Specific implementations of the implementer interface.

When to Use the Bridge Pattern

- When you have multiple dimensions of variation in your system and want to avoid an explosion of classes due to inheritance.
- To decouple high-level abstractions from low-level implementations.
- To allow both the abstraction and the implementation to evolve independently.

Example.

```
public interface IRenderer
{
    void Render(string shape);
}

public class VectorRenderer : IRenderer
{
    public void Render(string shape)
    {
        Console.WriteLine($"Drawing {shape} using Vector
           Renderer.");
    }
}

public class RasterRenderer : IRenderer
{
    public void Render(string shape)
    {
        Console.WriteLine($"Drawing {shape} using Raster
           Renderer.");
    }
}

public abstract class Shape
```

```
{
    protected IRenderer Renderer;

    protected Shape(IRenderer renderer)
    {
        Renderer = renderer;
    }

    public abstract void Draw();
}


public class Circle : Shape
{
    private readonly string _name;

    public Circle(IRenderer renderer, string name) : base(renderer)
    {
        _name = name;
    }

    public override void Draw()
    {
        Renderer.Render($"Circle: {_name}");
    }
}

public class Square : Shape
{
    private readonly string _name;

    public Square(IRenderer renderer, string name) : base(renderer)
    {
        _name = name;
    }

    public override void Draw()
    {
        Renderer.Render($"Square: {_name}");
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        // Use vector renderer for the shapes
        IRenderer vectorRenderer = new VectorRenderer();
        Shape circle = new Circle(vectorRenderer, "MyCircle");
        Shape square = new Square(vectorRenderer, "MySquare");

        circle.Draw(); // Drawing Circle: MyCircle using Vector
           Renderer.
        square.Draw(); // Drawing Square: MySquare using Vector
           Renderer.

        // Switch to raster renderer
        IRenderer rasterRenderer = new RasterRenderer();
        circle = new Circle(rasterRenderer, "MyCircle");
        square = new Square(rasterRenderer, "MySquare");

        circle.Draw(); // Drawing Circle: MyCircle using Raster
           Renderer.
        square.Draw(); // Drawing Square: MySquare using Raster
           Renderer.

        Console.ReadKey();
    }
}
```

Composite

The Composite Pattern is a structural design pattern that allows you to treat individual objects and compositions of objects uniformly. This pattern is particularly useful for representing part-whole hierarchies, such as a tree structure, where objects can be composed recursively. The Composite Pattern simplifies client code by allowing it to interact with single objects and groups of objects in the same way.

Key Components

- Component: An abstract class or interface that declares the common operations for both individual objects and compositions.

- Leaf: Represents individual objects in the hierarchy. A leaf has no children.
- Composite: Represents a group of objects (leaves or other composites). It implements the component interface and can store child components.
- Client: Interacts with objects in the composition through the component interface.

Example.

```
public interface IOrganizationComponent
{
    void DisplayDetails();
}

public class Employee : IOrganizationComponent
{
    public string Name { get; }
    public string Position { get; }

    public Employee(string name, string position)
    {
        Name = name;
        Position = position;
    }

    public void DisplayDetails()
    {
        Console.WriteLine($"{Position}: {Name}");
    }
}

using System.Collections.Generic;

public class Department : IOrganizationComponent
{
    private readonly List<IOrganizationComponent> _components = new
       List<IOrganizationComponent>();
    public string Name { get; }

    public Department(string name)
    {
        Name = name;
```

```
    }

    public void Add(IOrganizationComponent component)
    {
        _components.Add(component);
    }

    public void Remove(IOrganizationComponent component)
    {
        _components.Remove(component);
    }

    public void DisplayDetails()
    {
        Console.WriteLine($"Department: {Name}");
        foreach (var component in _components)
        {
            component.DisplayDetails();
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Create individual employees
        var emp1 = new Employee("Alice", "Developer");
        var emp2 = new Employee("Bob", "Designer");
        var emp3 = new Employee("Charlie", "Manager");

        // Create a department and add employees to it
        var devDepartment = new Department("Development");
        devDepartment.Add(emp1);
        devDepartment.Add(emp2);

        // Create another department and add a sub-department and
           employee to it
        var mainDepartment = new Department("Head Office");
        mainDepartment.Add(devDepartment);
        mainDepartment.Add(emp3);
```

```
        // Display the details of the entire organization
        mainDepartment.DisplayDetails();

        Console.ReadKey();
    }
}
```

Decorator

The Decorator Pattern is a structural design pattern that allows you to dynamically add or modify the functionality of an object at runtime without altering its structure. This pattern uses composition rather than inheritance, making it flexible and reusable.

The Decorator Pattern is particularly useful when you want to add responsibilities to objects without affecting other objects of the same class. It is commonly used in scenarios where there are multiple behaviors that can be combined in various configurations.

Key Components:

- Component: An interface or abstract class that defines the core functionality.
- Concrete Component: A class that implements the Component interface and provides the default behavior.
- Decorator: An abstract class that implements the Component interface and holds a reference to a Component object. It delegates operations to the wrapped object.
- Concrete Decorator: A class that extends the functionality of the Decorator.

Example. Consider a text editor where users can format text with additional features such as bold, italic, and underline. Instead of creating separate subclasses for every combination (e.g., BoldText, ItalicText, BoldItalicText), we can use the Decorator Pattern to dynamically add these features.

```
// The Component:
public interface IText
{
    string GetContent();
}

// The Concrete Component:
public class PlainText : IText
```

```
{
    private readonly string _content;

    public PlainText(string content)
    {
        _content = content;
    }

    public string GetContent()
    {
        return _content;
    }
}

// The Decorator:
public abstract class TextDecorator : IText
{
    protected IText Text;

    protected TextDecorator(IText text)
    {
        Text = text;
    }

    public abstract string GetContent();
}

// Decorators Implementation:
public class BoldDecorator : TextDecorator
{
    public BoldDecorator(IText text) : base(text) { }

    public override string GetContent()
    {
        return $"<b>{Text.GetContent()}</b>";
    }
}

public class ItalicDecorator : TextDecorator
{
    public ItalicDecorator(IText text) : base(text) { }
```

```
    public override string GetContent()
    {
        return $"<i>{Text.GetContent()}</i>";
    }
}

public class UnderlineDecorator : TextDecorator
{
    public UnderlineDecorator(IText text) : base(text) { }

    public override string GetContent()
    {
        return $"<u>{Text.GetContent()}</u>";
    }
}

// Usage:
class Program
{
    static void Main(string[] args)
    {
        // Base text
        IText plainText = new PlainText("Hello, World!");

        // Decorate the text with bold
        IText boldText = new BoldDecorator(plainText);

        // Decorate the bold text with italic
        IText boldItalicText = new ItalicDecorator(boldText);

        // Decorate the bold-italic text with underline
        IText fullyDecoratedText = new
           UnderlineDecorator(boldItalicText);

        Console.WriteLine("Plain Text: " + plainText.GetContent());
        Console.WriteLine("Bold Text: " + boldText.GetContent());
        Console.WriteLine("Bold Italic Text: " +
           boldItalicText.GetContent());
        Console.WriteLine("Fully Decorated Text: " +
           fullyDecoratedText.GetContent());
```

```
    }
}
```

Facade

The Facade Pattern is a structural design pattern that provides a simplified interface to a complex subsystem. It acts as a "front-facing" interface, masking the complexity of the underlying components and making the system easier to use for clients.

The Facade Pattern is particularly useful when you have a system with multiple subsystems and want to provide a unified, easy-to-use interface for clients while preserving flexibility to interact with individual subsystems when needed.

Key Components:

- Subsystem Classes: These are the components of the system that perform various tasks. They are usually complex and require intricate interactions.
- Facade: The class that provides a simplified interface to the subsystem. It delegates client requests to appropriate subsystems.
- Client: The consumer of the facade. It interacts with the facade rather than directly with the subsystems.

Example. Imagine a home automation system with various subsystems such as lights, air conditioning, and entertainment systems. Using the Facade Pattern, you can create a unified interface to control these systems, such as a "HomeController."

```
// Subsystem Classes:
public class LightingSystem
{
    public void TurnOnLights() => Console.WriteLine("Lights are
       turned on.");
    public void TurnOffLights() => Console.WriteLine("Lights are
       turned off.");
}

public class AirConditioningSystem
{
    public void SetTemperature(int temperature) =>
        Console.WriteLine($"Air conditioning set to
           {temperature}пїSC.");
    public void TurnOffAC() => Console.WriteLine("Air conditioning
       is turned off.");
```

```
}

public class EntertainmentSystem
{
    public void PlayMusic() => Console.WriteLine("Music is
       playing.");
    public void StopMusic() => Console.WriteLine("Music is
       stopped.");
}

// Facade:
public class HomeController
{
    private readonly LightingSystem _lightingSystem;
    private readonly AirConditioningSystem _airConditioningSystem;
    private readonly EntertainmentSystem _entertainmentSystem;

    public HomeController()
    {
        _lightingSystem = new LightingSystem();
        _airConditioningSystem = new AirConditioningSystem();
        _entertainmentSystem = new EntertainmentSystem();
    }

    public void StartEveningMode()
    {
        Console.WriteLine("Starting Evening Mode...");
        _lightingSystem.TurnOnLights();
        _airConditioningSystem.SetTemperature(22);
        _entertainmentSystem.PlayMusic();
    }

    public void StopEveningMode()
    {
        Console.WriteLine("Stopping Evening Mode...");
        _lightingSystem.TurnOffLights();
        _airConditioningSystem.TurnOffAC();
        _entertainmentSystem.StopMusic();
    }
}
```

```
// Usage:
class Program
{
    static void Main(string[] args)
    {
        var homeController = new HomeController();

        homeController.StartEveningMode();
        Console.WriteLine();

        homeController.StopEveningMode();

        Console.ReadKey();
    }
}
```

Flyweight

The Flyweight Pattern is a structural design pattern that focuses on reducing memory usage by sharing as much data as possible among similar objects. This is particularly useful when dealing with a large number of objects that have identical or nearly identical data.

The pattern achieves this by separating an objectвЂ™s intrinsic state (shared among multiple objects) from its extrinsic state (specific to an instance). The intrinsic state is stored in a single shared object, while the extrinsic state is managed externally.

You can use this pattern, when your application needs to handle a large number of similar objects or memory usage is a concern, and sharing common data can significantly reduce memory consumption.

Key components:

- Flyweight: An interface or abstract class that defines the shared behavior.
- Concrete Flyweight: Implements the Flyweight interface and stores intrinsic state.
- Flyweight Factory: Manages a pool of Flyweight objects and ensures shared objects are reused.
- Client: Uses Flyweight objects and supplies the extrinsic state when needed.

Example. Imagine a drawing application that renders a forest with thousands of trees. Each tree has shared characteristics like species, texture, and color (intrinsic

state) but unique properties such as position and size (extrinsic state). Using the Flyweight Pattern, we can efficiently manage these tree objects.

```
# The Flyweight interface defines the method that accepts the
   extrinsic state.
public interface ITree
{
    void Display(int x, int y, int size);
}

# The concrete flyweight stores the intrinsic state.
public class TreeType : ITree
{
    public string Species { get; }
    public string Color { get; }
    public string Texture { get; }

    public TreeType(string species, string color, string texture)
    {
        Species = species;
        Color = color;
        Texture = texture;
    }

    public void Display(int x, int y, int size)
    {
        Console.WriteLine($"Tree [{Species}, {Color}, {Texture}] at
           ({x}, {y}) with size {size}.");
    }
}

# The factory manages the pool of shared flyweight objects
using System.Collections.Generic;

public class TreeFactory
{
    private readonly Dictionary<string, ITree> _treeTypes = new();

    public ITree GetTreeType(string species, string color, string
       texture)
    {
        var key = $"{species}-{color}-{texture}";
```

```
        if (!_treeTypes.ContainsKey(key))
        {
            _treeTypes[key] = new TreeType(species, color, texture);
        }

        return _treeTypes[key];
    }
}

# Usage:
using System;
using System.Collections.Generic;

class Program
{
    static void Main(string[] args)
    {
        var treeFactory = new TreeFactory();
        var trees = new List<(ITree tree, int x, int y, int
           size)>();

        // Add trees to the forest
        var oakTree = treeFactory.GetTreeType("Oak", "Green",
           "Rough");
        trees.Add((oakTree, 10, 20, 5));
        trees.Add((oakTree, 15, 25, 6));

        var pineTree = treeFactory.GetTreeType("Pine", "Dark
           Green", "Smooth");
        trees.Add((pineTree, 30, 40, 8));
        trees.Add((pineTree, 35, 45, 10));

        // Render the trees
        foreach (var (tree, x, y, size) in trees)
        {
            tree.Display(x, y, size);
        }

        Console.ReadKey();
    }
}
```

Proxy

The Proxy pattern is a structural design pattern that acts as an intermediary between a client and an object, controlling access to the actual implementation. It can be used for caching, logging, security, or lazy initialization.

A common use case in web applications is implementing an API request proxy to cache responses and reduce the load on external services.

Example.

```
public interface IProductService
{
    Task<string> GetProductsAsync();
}

public class ProductService : IProductService
{
    private readonly HttpClient _httpClient;
    public ProductService(HttpClient httpClient)
    {
        _httpClient = httpClient;
    }

    public async Task<string> GetProductsAsync()
    {
        return await
           _httpClient.GetStringAsync("https://api.example.com/products");
    }
}

// Implementing a Proxy for Caching
public class ProductServiceProxy : IProductService
{
    private readonly IProductService _realService;
    private string? _cachedResponse;

    public ProductServiceProxy(IProductService realService)
    {
        _realService = realService;
    }

    public async Task<string> GetProductsAsync()
    {
        if (_cachedResponse == null)
```

```
        {
            _cachedResponse = await _realService.GetProductsAsync();
        }
        return _cachedResponse;
    }
}
```

Chain of Responsibility

The *Chain of Responsibility* pattern is a behavioral design pattern that allows multiple objects to process a request sequentially. Each object in the chain either handles the request or passes it to the next handler in line. This pattern promotes loose coupling between senders and receivers of a request.

Application:

- Logging frameworks (different log levels such as Debug, Info, Error)
- Middleware pipelines in web applications
- Request validation and authorization
- Handling UI events (e.g., event bubbling)

Example. A common use case in Web Development is middleware. Middleware components in the request pipeline process HTTP requests in sequence. Let's create a custom middleware that uses the Chain of Responsibility pattern.

```
// handler interface
public abstract class MiddlewareHandler
{
    protected MiddlewareHandler? NextHandler;

    public void SetNext(MiddlewareHandler next)
    {
        NextHandler = next;
    }

    public abstract Task Handle(HttpContext context);
}

// specific handlers

// login middleware
public class LoggingMiddleware : MiddlewareHandler
```

```
{
    public override async Task Handle(HttpContext context)
    {
        Console.WriteLine($"Request received:
           {context.Request.Path}");

        if (NextHandler != null)
        {
            await NextHandler.Handle(context);
        }
    }
}

// authentication middleware
public class AuthenticationMiddleware : MiddlewareHandler
{
    public override async Task Handle(HttpContext context)
    {
        if (!context.Request.Headers.ContainsKey("Authorization"))
        {
            context.Response.StatusCode = 401;
            await context.Response.WriteAsync("Unauthorized");
            return;
        }

        if (NextHandler != null)
        {
            await NextHandler.Handle(context);
        }
    }
}


// response middleware
public class ResponseMiddleware : MiddlewareHandler
{
    public override async Task Handle(HttpContext context)
    {
        context.Response.StatusCode = 200;
        await context.Response.WriteAsync("Request processed
           successfully");
```

```
    }
}
```

Command

The Command pattern is a behavioral design pattern that turns a request into a stand-alone object. This allows you to parameterize methods with different requests, delay execution, queue commands, and support undo/redo operations.

- Implementing undo/redo functionality in applications.
- Queuing and scheduling operations.
- Encapsulating business logic to decouple sender and receiver.
- Using commands in ASP.NET controllers or mediators (e.g., CQRS pattern).

Example. We will implement the Command pattern to handle user actions in an Web Application. The example simulates an order processing system where commands handle different operations.

```
public interface ICommand
{
    Task Execute();
}

public class PlaceOrderCommand : ICommand
{
    private readonly OrderService _orderService;
    private readonly string _orderDetails;

    public PlaceOrderCommand(OrderService orderService, string
       orderDetails)
    {
        _orderService = orderService;
        _orderDetails = orderDetails;
    }

    public async Task Execute()
    {
        await _orderService.PlaceOrder(_orderDetails);
    }
}
```

```
public class CancelOrderCommand : ICommand
{
    private readonly OrderService _orderService;
    private readonly int _orderId;

    public CancelOrderCommand(OrderService orderService, int
       orderId)
    {
        _orderService = orderService;
        _orderId = orderId;
    }

    public async Task Execute()
    {
        await _orderService.CancelOrder(_orderId);
    }
}

public class CommandInvoker
{
    private readonly List<ICommand> _commandQueue = new();

    public void AddCommand(ICommand command)
    {
        _commandQueue.Add(command);
    }

    public async Task ExecuteCommands()
    {
        foreach (var command in _commandQueue)
        {
            await command.Execute();
        }

        _commandQueue.Clear(); // Clear after execution
    }
}
```

Iterator

The Iterator pattern is a behavioral design pattern that provides a way to traverse

a collection without exposing its underlying structure. It abstracts the traversal logic, allowing different types of collections to be iterated in a uniform way.

- Providing a standard way to iterate over complex data structures (e.g., trees, graphs).
- Supporting multiple iteration algorithms without modifying the collection.
- Implementing lazy loading or pagination in web applications.
- Encapsulating traversal logic inside an iterator object.

Example. We can implement Iterator pattern to traverse collection of certain items (for example, products) in the web application:

```
// Iterator interface
public interface IIterator<T>
{
    bool HasNext();
    T Next();
}

// Concrete iterator:
public class ProductIterator : IIterator<Product>
{
    private readonly List<Product> _products;
    private int _position = 0;

    public ProductIterator(List<Product> products)
    {
        _products = products;
    }

    public bool HasNext()
    {
        return _position < _products.Count;
    }

    public Product Next()
    {
        if (!HasNext())
        {
            throw new InvalidOperationException("No more products.");
        }
```

```
        return _products[_position++];
    }
}


// Iterable collection:
public interface IProductCollection
{
    IIterator<Product> CreateIterator();
}

// And its implementation:
public class ProductCollection : IProductCollection
{
    private readonly List<Product> _products = new();

    public void AddProduct(Product product)
    {
        _products.Add(product);
    }

    public IIterator<Product> CreateIterator()
    {
        return new ProductIterator(_products);
    }
}

// Define product model:
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; } = string.Empty;
}

// Example usage of product iterator in web api controller:
[ApiController]
[Route("api/products")]
public class ProductsController : ControllerBase
{
    private readonly ProductCollection _productCollection;
```

```
    public ProductsController()
    {
        _productCollection = new ProductCollection();
        _productCollection.AddProduct(new Product { Id = 1, Name =
           "Laptop" });
        _productCollection.AddProduct(new Product { Id = 2, Name =
           "Smartphone" });
        _productCollection.AddProduct(new Product { Id = 3, Name =
           "Tablet" });
    }

    [HttpGet("next")]
    public IActionResult GetNextProduct()
    {
        var iterator = _productCollection.CreateIterator();

        if (!iterator.HasNext())
        {
            return NotFound("No more products.");
        }

        return Ok(iterator.Next());
    }
}
```

Mediator

The Mediator pattern is a behavioral design pattern that reduces direct dependencies between components by centralizing communication through a mediator object. Instead of components communicating directly with each other, they send messages to a mediator, which handles interactions.

- Decoupling UI elements (e.g., Blazor or WPF applications).
- Managing interactions between multiple services in a web application.
- Implementing CQRS using MediatR in ASP.NET Core.
- Handling event-driven systems where multiple objects interact.

Example.

```
// We will create a query that fetches products.
using MediatR;
```

```
public record GetProductsQuery() : IRequest<List<Product>>;


// This class handles the query and returns a list of products.
public class GetProductsHandler : IRequestHandler<GetProductsQuery,
   List<Product>>
{
    private readonly ProductService _productService;

    public GetProductsHandler(ProductService productService)
    {
        _productService = productService;
    }

    public async Task<List<Product>> Handle(GetProductsQuery
       request, CancellationToken cancellationToken)
    {
        return await _productService.GetProductsAsync();
    }
}


// Define the Product Model and Service
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; } = string.Empty;
}

public class ProductService
{
    private static readonly List<Product> _products = new()
    {
        new Product { Id = 1, Name = "Laptop" },
        new Product { Id = 2, Name = "Smartphone" },
        new Product { Id = 3, Name = "Tablet" }
    };

    public Task<List<Product>> GetProductsAsync()
    {
```

```
        return Task.FromResult(_products);
    }
}


// And controller:
[ApiController]
[Route("api/products")]
public class ProductsController : ControllerBase
{
    private readonly IMediator _mediator;

    public ProductsController(IMediator mediator)
    {
        _mediator = mediator;
    }

    [HttpGet]
    public async Task<IActionResult> GetProducts()
    {
        var products = await _mediator.Send(new GetProductsQuery());
        return Ok(products);
    }
}


// Register mediator in Program.cs
builder.Services.AddMediatR(Assembly.GetExecutingAssembly());
builder.Services.AddSingleton<ProductService>();
builder.Services.AddControllers();
```

Memento

The Memento pattern is a behavioral design pattern that allows you to capture and save an object's internal state without violating encapsulation — so it can be restored to that state later. It's commonly used for implementing undo functionality.

Use cases:

- Undo or redo in forms or editors.
- Snapshots of an entity before performing risky updates.
- Session history in web applications.

- Temporal state tracking (e.g., configuration changes).

Example. Web application with functionality to edit a product and undo the edit using memento pattern.

```
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }

    public ProductMemento SaveState()
    {
        return new ProductMemento(Id, Name);
    }

    public void RestoreState(ProductMemento memento)
    {
        Id = memento.Id;
        Name = memento.Name;
    }
}

public class ProductMemento
{
    public int Id { get; }
    public string Name { get; }

    public ProductMemento(int id, string name)
    {
        Id = id;
        Name = name;
    }
}

public class ProductHistory
{
    private readonly Stack<ProductMemento> _history = new();

    public void Save(Product product)
    {
        _history.Push(product.SaveState());
    }
```

```
    public void Undo(Product product)
    {
        if (_history.Count > 0)
        {
            var previousState = _history.Pop();
            product.RestoreState(previousState);
        }
    }
}

// Usage:
public class ProductService
{
    private Product _product = new Product { Id = 1, Name =
       "Original Product" };
    private readonly ProductHistory _history = new();

    public Product Get() => _product;

    public void Update(string newName)
    {
        _history.Save(_product);
        _product.Name = newName;
    }

    public void Undo()
    {
        _history.Undo(_product);
    }
}
```

Observer

The Observer pattern is a behavioral design pattern that defines a one-to-many dependency between objects so that when one object (the Subject) changes state, all its dependents (called Observers) are notified and updated automatically.

Applications:

- Notifying multiple services of an event (e.g., user registration).
- WebSocket or SignalR-based real-time updates.
- Event-driven architecture / pub-sub messaging.

- Logging, caching, or triggering background jobs on state change.

Example. We'll simulate a user registration system where observers (like logging and email services) are notified when a new user is registered.

```
public interface IUserObserver
{
    void OnUserRegistered(User user);
}

public class UserService
{
    private readonly List<IUserObserver> _observers = new();

    public void RegisterObserver(IUserObserver observer)
    {
        _observers.Add(observer);
    }

    public void UnregisterObserver(IUserObserver observer)
    {
        _observers.Remove(observer);
    }

    public void RegisterUser(User user)
    {
        Console.WriteLine($"User registered: {user.Name}");

        // Notify all observers
        foreach (var observer in _observers)
        {
            observer.OnUserRegistered(user);
        }
    }
}

// User Model
public class User
{
    public string Name { get; set; } = string.Empty;
    public string Email { get; set; } = string.Empty;
}
```

```
// Email notification of the user:
public class EmailNotificationService : IUserObserver
{
    public void OnUserRegistered(User user)
    {
        Console.WriteLine($"[Email] Welcome email sent to
           {user.Email}");
    }
}

// Logging service:
public class LoggingService : IUserObserver
{
    public void OnUserRegistered(User user)
    {
        Console.WriteLine($"[Log] User {user.Name} has been
           registered.");
    }
}
```

State

The State design pattern (GoF) is a behavioral pattern that lets an object alter its behavior when its internal state changes.

Instead of writing big if/else or switch statements everywhere, you encapsulate state-specific behavior in separate classes and let the object delegate work to the current state.

Example. Let's say we're building an e-commerce site. An Order can be in multiple states (Pending, Paid, Shipped), and its behavior changes depending on the state.

```
// State intefrace
public interface IOrderState
{
    void Pay(OrderContext context);
    void Ship(OrderContext context);
}



// Implement concrete states
public class PendingState : IOrderState
```

```
{
    public void Pay(OrderContext context)
    {
        Console.WriteLine("Order is now Paid.");
        context.SetState(new PaidState());
    }

    public void Ship(OrderContext context)
    {
        Console.WriteLine("Cannot ship. Order is still pending
           payment.");
    }
}

public class PaidState : IOrderState
{
    public void Pay(OrderContext context)
    {
        Console.WriteLine("Order already paid.");
    }

    public void Ship(OrderContext context)
    {
        Console.WriteLine("Order is now Shipped.");
        context.SetState(new ShippedState());
    }
}

public class ShippedState : IOrderState
{
    public void Pay(OrderContext context)
    {
        Console.WriteLine("Cannot pay. Order already shipped.");
    }

    public void Ship(OrderContext context)
    {
        Console.WriteLine("Order already shipped.");
    }
}
```

```
// The Context class
public class OrderContext
{
    private IOrderState _state;

    public OrderContext()
    {
        _state = new PendingState(); // default state
    }

    public void SetState(IOrderState state) => _state = state;

    public void Pay() => _state.Pay(this);
    public void Ship() => _state.Ship(this);
}


// Usage example
public class OrdersController : ControllerBase
{
    private readonly OrderContext _orderContext;

    public OrdersController(OrderContext orderContext)
    {
        _orderContext = orderContext;
    }

    [HttpPost("pay")]
    public IActionResult Pay()
    {
        _orderContext.Pay();
        return Ok("Order processed.");
    }

    [HttpPost("ship")]
    public IActionResult Ship()
    {
        _orderContext.Ship();
        return Ok("Order processed.");
    }
```

```
}
```

Strategy

The Strategy Pattern is one of the most widely used behavioral design patterns. It enables selecting an algorithm's behavior at runtime instead of hard-coding it. By encapsulating algorithms in separate classes and making them interchangeable, the Strategy Pattern adheres to the Open/Closed Principle: classes are open for extension but closed for modification.

Example. Imagine a .NET Web API where users can place orders. Discounts vary depending on the customer type (e.g., Regular, Premium, VIP). Instead of hard-coding discount logic into the service, we can apply the Strategy Pattern.

```
// Define the Strategy Interface
public interface IDiscountStrategy
{
    decimal ApplyDiscount(decimal amount);
}



// Implement Concrete Strategies
public class RegularCustomerDiscount : IDiscountStrategy
{
    public decimal ApplyDiscount(decimal amount) => amount; // no
       discount
}

public class PremiumCustomerDiscount : IDiscountStrategy
{
    public decimal ApplyDiscount(decimal amount) => amount * 0.9m;
       // 10% off
}

public class VipCustomerDiscount : IDiscountStrategy
{
    public decimal ApplyDiscount(decimal amount) => amount * 0.8m;
       // 20% off
}
```

```
// Create a Context
public class DiscountService
{
    private readonly IDiscountStrategy _discountStrategy;

    public DiscountService(IDiscountStrategy discountStrategy)
    {
        _discountStrategy = discountStrategy;
    }

    public decimal CalculatePrice(decimal amount)
    {
        return _discountStrategy.ApplyDiscount(amount);
    }
}

// Register Strategies in Dependency Injection, in Program.cs
builder.Services.AddScoped<RegularCustomerDiscount>();
builder.Services.AddScoped<PremiumCustomerDiscount>();
builder.Services.AddScoped<VipCustomerDiscount>();
builder.Services.AddScoped<DiscountService>();


// Usage:
[ApiController]
[Route("api/[controller]")]
public class OrdersController : ControllerBase
{
    private readonly IServiceProvider _serviceProvider;

    public OrdersController(IServiceProvider serviceProvider)
    {
        _serviceProvider = serviceProvider;
    }

    [HttpPost("{customerType}")]
    public IActionResult CreateOrder(string customerType,
       [FromBody] decimal amount)
    {
        IDiscountStrategy strategy = customerType.ToLower() switch
```

```
        {
            "regular" =>
               _serviceProvider.GetRequiredService<RegularCustomerDiscount>(
            "premium" =>
               _serviceProvider.GetRequiredService<PremiumCustomerDiscount>(
            "vip"     =>
               _serviceProvider.GetRequiredService<VipCustomerDiscount>(),
            _         => throw new ArgumentException("Unknown
               customer type")
        };

        var discountService = new DiscountService(strategy);
        var finalPrice = discountService.CalculatePrice(amount);

        return Ok(new { OriginalPrice = amount, FinalPrice =
           finalPrice });
    }
}
```

Template Method

The Template Method pattern is a behavioral design pattern that defines the skeleton of an algorithm in a base class while allowing subclasses to override specific steps without changing the algorithm's overall structure. It is particularly useful when you have a process that always follows the same outline but has variations in some of its steps. By applying this pattern, you promote code reuse and enforce a consistent workflow, while giving flexibility to extend or customize parts of the process.

Example.

```
public abstract class DataImportTemplate
{
    // The Template Method defines the skeleton of the workflow
    public void ImportData(IEnumerable<string> records)
    {
        Validate(records);
        Process(records);
        Log(records);
    }

    // Steps that can be overridden
    protected abstract void Validate(IEnumerable<string> records);
```

```
    protected abstract void Process(IEnumerable<string> records);

    // Common step shared across all subclasses
    protected virtual void Log(IEnumerable<string> records)
    {
        Console.WriteLine($"{records.Count()} records have been
            imported at {DateTime.UtcNow}");
    }
}

public class CustomerImportService : DataImportTemplate
{
    protected override void Validate(IEnumerable<string> records)
    {
        if (!records.All(r => r.Contains("@")))
            throw new InvalidOperationException("All customer
                records must contain an email.");
    }

    protected override void Process(IEnumerable<string> records)
    {
        foreach (var record in records)
        {
            Console.WriteLine($"Processing customer record:
                {record}");
            // Insert into database
        }
    }
}

public class ProductImportService : DataImportTemplate
{
    protected override void Validate(IEnumerable<string> records)
    {
        if (!records.All(r => r.Split(',').Length >= 2))
            throw new InvalidOperationException("Product records
                must contain at least name and price.");
    }

    protected override void Process(IEnumerable<string> records)
    {
```

```
        foreach (var record in records)
        {
            Console.WriteLine($"Processing product record:
               {record}");
            // Insert into database
        }
    }
}
```

In your Web API controller, you can now reuse the workflow for different imports:

```
[ApiController]
[Route("api/[controller]")]
public class ImportController : ControllerBase
{
    [HttpPost("customers")]
    public IActionResult ImportCustomers([FromBody] List<string>
       records)
    {
        var service = new CustomerImportService();
        service.ImportData(records);
        return Ok("Customer data imported successfully");
    }

    [HttpPost("products")]
    public IActionResult ImportProducts([FromBody] List<string>
       records)
    {
        var service = new ProductImportService();
        service.ImportData(records);
        return Ok("Product data imported successfully");
    }
}
```

Visitor

The Visitor pattern is a behavioral design pattern that lets you separate an algorithm from the objects it operates on. Instead of putting logic inside your data models, you "visit" them with a separate object (the visitor), which encapsulates the operation.

The main benefit is that you can add new operations without changing the classes of the elements you're working with. This is useful when you have a hierarchy of

objects and want to perform different actions depending on the object type, but don't want to clutter the model classes with many methods.

Example. Suppose you have an API that handles different notifications: Email and SMS. Each notification needs to be processed differently (e.g., send via SMTP vs. send via Twilio). The Visitor pattern can help keep this clean.

```
// Step 1. Define the element interface (accepts a visitor):
public interface INotification
{
    void Accept(INotificationVisitor visitor);
}


// Step 2. Implement concrete elements:
public class EmailNotification : INotification
{
    public string Subject { get; set; }
    public string Body { get; set; }

    public void Accept(INotificationVisitor visitor)
    {
        visitor.Visit(this);
    }
}

public class SmsNotification : INotification
{
    public string PhoneNumber { get; set; }
    public string Message { get; set; }

    public void Accept(INotificationVisitor visitor)
    {
        visitor.Visit(this);
    }
}



// Step 3. Define the visitor interface:
public interface INotificationVisitor
{
    void Visit(EmailNotification email);
```

```
    void Visit(SmsNotification sms);
}



// Step 4. Create concrete visitors (operations):
public class SendNotificationVisitor : INotificationVisitor
{
    public void Visit(EmailNotification email)
    {
        Console.WriteLine($"Sending Email: {email.Subject} -
           {email.Body}");
        // logic: SMTP send
    }

    public void Visit(SmsNotification sms)
    {
        Console.WriteLine($"Sending SMS to {sms.PhoneNumber}:
           {sms.Message}");
        // logic: SMS gateway
    }
}



// Step 5. Use in a Web API controller:
[ApiController]
[Route("api/[controller]")]
public class NotificationsController : ControllerBase
{
    [HttpPost("send")]
    public IActionResult SendNotifications([FromBody]
       List<INotification> notifications)
    {
        var visitor = new SendNotificationVisitor();

        foreach (var notification in notifications)
        {
            notification.Accept(visitor);
        }
```

```
        return Ok("Notifications sent.");
    }
}
```

Chapter 11

Unit and integration testing. Tools for unit testing web applications.

Unit testing is a process in programming that allows you to check the correctness of individual modules of the source code of the program.

The idea is to write tests for each non-trivial function or method. This allows you to quickly check whether the next code change has led to regression, that is, to the appearance of errors in already tested areas of the program, and also facilitates the detection and elimination of such errors.

Key Features of Unit Tests:

- Isolation: Tests are executed independently of other parts of the system.
- Automation: Tests can be run automatically as part of the development process.
- Repeatability: Once written, unit tests can be run as many times as needed to verify code changes.
- Fast Feedback: Unit tests are usually quick to execute, providing immediate feedback on code correctness.

Writing unit tests encourages developers to think through edge cases and potential errors, resulting in more robust code. Also, unit tests help identify and fix bugs early in the development process when they are easier and less costly to address. With a solid suite of unit tests, developers can confidently refactor or enhance code without the risk of introducing regressions.

Unit tests serve as a form of living documentation, demonstrating how specific functions or methods are expected to behave. Besides these advantages, by catching issues early, unit tests reduce the time and cost associated with debugging and fixing defects in later stages of development. By making unit testing an integral part of the development lifecycle, teams can build more reliable and maintainable software systems.

Unit tests implementation in .NET

Create new project on your solution via Visual Studio, choosing NUnit Test Project out of the available options.

NUnit is a popular open-source unit testing framework for .NET applications. It provides a rich set of features to write and execute unit tests, making it easier for developers to ensure the correctness and reliability of their code. With NUnit, developers can create test cases, organize them into test suites, and automate the process of verifying application behavior.

After new test project is loaded into your solution, you should add reference to the project you want to cover with unit tests to your NUnit tests project.

Suppose, we want to cover with unit tests the following service, that was defined in one of the previous chapters:

```
using demo_rest.bll.Interfaces;
using demo_rest.dal;
using demo_rest.dal.Entities;
using Microsoft.EntityFrameworkCore;

namespace demo_rest.bll.Services;

public class StudentService : IStudentService
{
    private DemoDbContext _demoDbContext;

    public StudentService(DemoDbContext demoDbContext)
    {
        _demoDbContext = demoDbContext;
    }

    public async Task<Student?> GetStudentAsync(int id)
    {
        return await _demoDbContext.Students
            .FirstOrDefaultAsync(s => s.Id == id);
    }

    public async Task<IEnumerable<Student>> GetStudentsAsync(int
       skip, int take)
    {
        return await _demoDbContext.Students
            .Skip(skip)
            .Take(take)
            .ToListAsync();
```

```
    }

    public async Task<Student> AddStudentAsync(Student student)
    {
        var studentAdded = await _demoDbContext.AddAsync(student);
        await _demoDbContext.SaveChangesAsync();
        return studentAdded.Entity;
    }

    public async Task<Student> UpdateStudentAsync(Student student)
    {
        var studentModified = _demoDbContext.Update(student);
        await _demoDbContext.SaveChangesAsync();
        return studentModified.Entity;
    }

    public async Task DeleteStudentAsync(int id)
    {
        var student = await _demoDbContext.Students
            .FirstOrDefaultAsync(s => s.Id == id);
        if (student == null)
        {
            throw new KeyNotFoundException();
        }
        _demoDbContext.Remove(student);
        await _demoDbContext.SaveChangesAsync();
    }
}
```

This service uses DemoDbContext for retrieval of data from the database, and provides functionality for adding, updating, retrieving and deleting entity, encapsulating student data.

Let's define the following service, that would contain all the test cases for this service, and name it StudentServiceShould.cs:

```
namespace demo_tests.Services;

public class StudentServiceShould
{
    [SetUp]
    public void Setup()
    {
```

```
    }

    [Test]
    public void Test1()
    {
        Assert.Pass();
    }
}
```

For now, we do not have any tests defined, but this structure already provides some key concepts, that you might want to familiarize yourself with. First of all, attribute `[SetUp]` defines a method, that sets up all the necessary tools for unit tests, and it runs automatically before test execution every time, you run the tests. Attribute `[Test]` defines a specific test case.

Before adding tests, we are going to install the package Microsoft.EntityFrameworkCore.I to the application, that we are covering with unit tests. It will allow us to use in-memory database for unit tests, instead of using real database. An in-memory database is a purpose-built database that relies primarily on internal memory for data storage.

After we added this NuGet package, we can now set up our database in tests in the following way:

```
using demo_rest.dal;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Diagnostics;

namespace demo_tests.Services;

public class StudentServiceShould
{
    private DemoDbContext _demoDbContext;

    [SetUp]
    public void Setup()
    {
        CreateDemoDbContextContext();
    }

    private void CreateDemoDbContextContext()
    {
        var dbContextOptions = new
           DbContextOptionsBuilder<DemoDbContext>()
```

```
            .UseInMemoryDatabase("EventsTestDB", b =>
              b.EnableNullChecks(false))
            .ConfigureWarnings(b =>
              b.Ignore(InMemoryEventId.TransactionIgnoredWarning))
            .Options;

        _demoDbContext = new DemoDbContext(
            dbContextOptions
        );

        _demoDbContext.Database.EnsureDeleted();
        _demoDbContext.Database.EnsureCreated();
    }

    [Test]
    public void Test1()
    {
        Assert.Pass();
    }
}
```

We can now also define method for adding entries to our Students table for test purposes:

```
private async Task AddStudent(Student student)
{
    await _demoDbContext.AddAsync(student);
    await _demoDbContext.SaveChangesAsync();
}
```

In order for unit testing method GetStudentsAsync, we should first add student entry to the database, and then verify that we retrieve exactly one student entry from the database with this method, using the Assert.That NUnit method:

```
[Test]
public async Task GetStudentsAsync_Should_Return()
{
    await AddStudent(new Student()
    {
        Name = "Jack",
        BirthDate = DateTime.Now.AddYears(-19)
    });
    var students = await _studentService.GetStudentsAsync(0, 10);
```

```
    Assert.That(students.Count(), Is.EqualTo(1));
}
```

Similarly, we can verify, that updating student with new name via UpdateStudentAsync method works as expected:

```
[Test]
public async Task UpdateStudentAsync_Should_Return()
{
    await AddStudent(new Student()
    {
        Name = "Jack",
        BirthDate = DateTime.Now.AddYears(-19)
    });
    var newName = "John Smith";
    var student = _demoDbContext.Students.First();
    student.Name = newName;
    var studentUpdated = await
       _studentService.UpdateStudentAsync(student);

    Assert.That(studentUpdated.Name, Is.EqualTo(newName));
}
```

In the same way, we can verify, that method AddStudentAsync adds student to the database, and returns newly added entries id:

```
[Test]
public async Task AddStudentAsync_Should_Return()
{
    var student = new Student()
    {
        Name = "Jack",
        BirthDate = DateTime.Now.AddYears(-19)
    };
    var studentAdded = await
       _studentService.AddStudentAsync(student);
    Assert.NotNull(studentAdded.Id);
}
```

Test case for the method GetStudentAsync looks like that:

```
[Test]
public async Task GetStudentAsync_Should_Return()
{
```

```
    var studentName = "Jack";
    await AddStudent(new Student()
    {
        Name = studentName,
        BirthDate = DateTime.Now.AddYears(-19)
    });
    var studentId = _demoDbContext.Students.First().Id;
    var student = await _studentService.GetStudentAsync(studentId);
    Assert.NotNull(student);
    Assert.That(student.Name, Is.EqualTo(studentName));
}
```

Method DeleteStudentAsync does not return any data, so we just verify that it gets executed without throwing any exceptions:

```
[Test]
public async Task DeleteStudentAsync_Should_Return()
{
    await AddStudent(new Student()
    {
        Name = "Name",
        BirthDate = DateTime.Now.AddYears(-19)
    });
    var student = _demoDbContext.Students.First();
    Assert.DoesNotThrowAsync(async () =>
        await _studentService.DeleteStudentAsync(student.Id), "");
}
```

It is also important, that our method throws exceptions when we expect it to. We can verify, for example, that method DeleteStudentAsync throws KeyNotFound exception, when we try do delete the entry, that does not exist in the database:

```
[Test]
public void DeleteStudentAsync_Should_Throw()
{
    var students = _demoDbContext.Students.ToList();
    _demoDbContext.Students.RemoveRange(students);
    Assert.ThrowsAsync<KeyNotFoundException>(async () =>
        await _studentService.DeleteStudentAsync(100), "");
}
```

All together, our test bundle looks like this:

```
using demo_rest.bll.Services;
```

```
using demo_rest.dal;
using demo_rest.dal.Entities;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Diagnostics;

namespace demo_tests.Services;

public class StudentServiceShould
{
    private DemoDbContext _demoDbContext;
    private StudentService _studentService;

    [SetUp]
    public void Setup()
    {
        CreateDemoDbContextContext();
        _studentService = new StudentService(_demoDbContext);
    }

    private void CreateDemoDbContextContext()
    {
        var dbContextOptions = new
           DbContextOptionsBuilder<DemoDbContext>()
            .UseInMemoryDatabase("DemoTestDB", b =>
               b.EnableNullChecks(false))
            .ConfigureWarnings(b =>
               b.Ignore(InMemoryEventId.TransactionIgnoredWarning))
            .Options;

        _demoDbContext = new DemoDbContext(
            dbContextOptions
        );

        _demoDbContext.Database.EnsureDeleted();
        _demoDbContext.Database.EnsureCreated();
    }

    [Test]
    public async Task GetStudentsAsync_Should_Return()
    {
        await AddStudent(new Student()
```

```
    {
        Name = "Jack",
        BirthDate = DateTime.Now.AddYears(-19)
    });
    var students = await _studentService.GetStudentsAsync(0,
       10);
    Assert.That(students.Count(), Is.EqualTo(1));
}

[Test]
public async Task UpdateStudentAsync_Should_Return()
{
    await AddStudent(new Student()
    {
        Name = "Jack",
        BirthDate = DateTime.Now.AddYears(-19)
    });
    var newName = "John Smith";
    var student = _demoDbContext.Students.First();
    student.Name = newName;
    var studentUpdated = await
       _studentService.UpdateStudentAsync(student);

    Assert.That(studentUpdated.Name, Is.EqualTo(newName));
}

[Test]
public async Task AddStudentAsync_Should_Return()
{
    var student = new Student()
    {
        Name = "Jack",
        BirthDate = DateTime.Now.AddYears(-19)
    };
    var studentAdded = await
       _studentService.AddStudentAsync(student);
    Assert.NotNull(studentAdded.Id);
}

[Test]
public async Task GetStudentAsync_Should_Return()
```

```
{
    var studentName = "Jack";
    await AddStudent(new Student()
    {
        Name = studentName,
        BirthDate = DateTime.Now.AddYears(-19)
    });
    var studentId = _demoDbContext.Students.First().Id;
    var student = await
       _studentService.GetStudentAsync(studentId);
    Assert.NotNull(student);
    Assert.That(student.Name, Is.EqualTo(studentName));
}

[Test]
public async Task DeleteStudentAsync_Should_Return()
{
    await AddStudent(new Student()
    {
        Name = "Name",
        BirthDate = DateTime.Now.AddYears(-19)
    });
    var student = _demoDbContext.Students.First();
    Assert.DoesNotThrowAsync(async () =>
        await _studentService.DeleteStudentAsync(student.Id),
           "");
}

[Test]
public void DeleteStudentAsync_Should_Throw()
{
    var students = _demoDbContext.Students.ToList();
    _demoDbContext.Students.RemoveRange(students);
    Assert.ThrowsAsync<KeyNotFoundException>(async () =>
        await _studentService.DeleteStudentAsync(100), "");
}

private async Task AddStudent(Student student)
{
    await _demoDbContext.AddAsync(student);
    await _demoDbContext.SaveChangesAsync();
```

```
    }
}
```

After you have completed creating this test collection, you can run them to verify, that your code works as expected. For that, in Visual Studio, go to View -> after that, click on Test Explorer. There, you will see the list of test cases, that you have created. Right click on them and choose option Run. If tests are executed without fail and the results of their execution is as expected by test case, you will see the test highlighted in green, else - in red.

Chapter 12

Logging.

Logging is an important component for the development and maintenance of the software applications. It serves as the backbone for monitoring, diagnosing, and understanding application behavior in both development and production environments.

Logging is first of all important for debugging and diagnostics: logs provide insights into the execution flow and state of an application. When an issue arises, logs act as breadcrumbs to help developers trace the root cause without needing to replicate the problem.

Logs are also important for performance monitoring, as by logging key metrics, such as response times and resource usage, developers can identify performance bottlenecks and optimize application behavior.

Logs can detect and alert on suspicious activity, such as unauthorized access or anomalous behavior, contributing to the overall security of the application.

Log levels categorize messages by their importance and urgency. Most systems use standard levels, such as:

- TRACE: Detailed information, typically of interest only during development or debugging.
- DEBUG: Diagnostic information for debugging. Useful in non-production environments.
- INFO: General informational messages about the applicationвЂ™s normal operations.
- WARN: Indications of potential issues or non-critical problems that might need attention.
- ERROR: Errors that prevent a specific operation from completing but do not crash the application.
- FATAL: Severe errors that lead to application crashes or critical failures.

There are several best practices for introducing logging to your application.

First of all, we should awoid over-logging. Logging everything can clutter log files and make it difficult to find useful information. Log only whatвЂ™s necessary and at the appropriate level.

We should also consider contextual logging, meaning, that we should include relevant context in logs, such as user IDs, request IDs, or session IDs. This makes logs more actionable and easier to correlate. But, we should never log sensitive information, such as passwords or personal identifiable information (PII), unless properly masked or encrypted.

We should also regularly verify that log messages are meaningful, actionable, and formatted correctly.

After you have set up the logging in your application, good practice is to set up alerts, special monitoring tools to trigger alerts for crucial log events, such as fatal errors or security violations.

In .net application logging can be implemented using either built-in tools, or NuGet packages, such as Serilog, NLog or log4net.

Implementation. ASP.NET Core comes with a built-in logging framework (Microsoft.Extensions.Logging) that can log to console, debug, eventSource and third-party providers (Serilog, NLog, Application Insights, etc.)

When you create a new Web API project, logging is already configured in `Program.cs`:

```
var builder = WebApplication.CreateBuilder(args);

// Add logging
builder.Logging.ClearProviders(); // optional: clear default
   providers
builder.Logging.AddConsole();  // log to console
builder.Logging.AddDebug();    // log to debug window

var app = builder.Build();
```

In order to add logging to the controller or service, you can inject `ILogger<T>`, where T is the service:

```
[ApiController]
[Route("api/[controller]")]
public class ProductsController : ControllerBase
{
    private readonly ILogger<ProductsController> _logger;

    public ProductsController(ILogger<ProductsController> logger)
    {
```

```
        _logger = logger;
    }

    [HttpGet("{id}")]
    public IActionResult GetProduct(int id)
    {
        _logger.LogInformation("Fetching product with ID
           {ProductId}", id);

        if (id <= 0)
        {
            _logger.LogWarning("Invalid product id: {ProductId}",
               id);
            return BadRequest("Invalid ID");
        }

        try
        {
            // Simulate fetching product
            var product = new { Id = id, Name = "Test Product" };
            return Ok(product);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error fetching product
               {ProductId}", id);
            return StatusCode(500, "Internal server error");
        }
    }
}
```

The built-in log levels can be used in the following way:

```
_logger.LogTrace("Trace log");
_logger.LogDebug("Debug log");
_logger.LogInformation("Info log");
_logger.LogWarning("Warning log");
_logger.LogError("Error log");
_logger.LogCritical("Critical log");
```

You can control what gets logged without changing code, in the `appsettings.json` file:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "MyAppNamespace": "Debug"
    }
  }
}
```

For advanced logging (structured logs, file sinks, Seq, ElasticSearch), many teams use Serilog.

Install packages:

```
dotnet add package Serilog.AspNetCore
dotnet add package Serilog.Sinks.Console
dotnet add package Serilog.Sinks.File
```

Configure in Program.cs:

```
using Serilog;

var builder = WebApplication.CreateBuilder(args);

// Setup Serilog
Log.Logger = new LoggerConfiguration()
    .ReadFrom.Configuration(builder.Configuration) // read from
       appsettings.json
    .Enrich.FromLogContext()
    .WriteTo.Console()
    .WriteTo.File("logs/log-.txt", rollingInterval:
       RollingInterval.Day)
    .CreateLogger();

builder.Host.UseSerilog();

var app = builder.Build();
```

And in appsettings.json:

```
"Serilog": {
  "MinimumLevel": "Information",
  "WriteTo": [
    { "Name": "Console" },
```

```
    { "Name": "File", "Args": { "path": "logs/webapi-.log",
      "rollingInterval": "Day" } }
  ]
}
```

Bibliography

[1] Martin Fowler, *CQRS*, 2011. (https://martinfowler.com/bliki/CQRS.html)

[2] Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley, 2003.

[3] Robert C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, Prentice Hall, 2017.

[4] Microsoft Documentation, *ASP.NET Core Web API documentation*. Available at: https://learn.microsoft.com/en-us/aspnet/core/web-api

[5] Eve Porcello, Alex Banks, *Learning GraphQL*, O'Reilly Media, 2018.

[6] Peter Himschoot, *Blazor Revealed*, Apress, 2019.

[7] OWASP Foundation, *OWASP Top Ten Web Application Security Risks*, 2021. Available at: https://owasp.org/Top10/

[8] Robert C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, Prentice Hall, 2008.