Midterm Exam Study Guide

IS 413 – Hilton

Here are the details for our Midterm Exam:

• Exam will be administered via Learning Suite

• You may take it anytime during the allotted time period from wherever you are

• There are 50 questions

• Questions will consist of True/False, Multiple Choice and Short Response types

• You have 60 minutes to complete the exam

• You may use your Notes, Textbook, and Past Programs on the exam

• You may NOT use any other resources (Internet search, other individuals, etc.)

Topics that may be covered on the Midterm Exam are:

• Client-Server Model

The **Client-Server Model** is a fundamental computing architecture that divides a system into two main entities: **clients** and **servers**. It is widely used in web applications, networking, and distributed computing.

In this model:

* A **client** requests services, resources, or data.
* A **server** processes the request and sends back the appropriate response.

This model enables efficient **resource sharing, centralized control, and scalability** in applications ranging from websites to database systems.

**Example:**

When you visit www.google.com:

1. Your web browser (client) sends a request to Google's web server.
2. The server processes the request and sends back an HTML page.
3. Your browser displays the page.

**Client**

* A client is a device, application, or process that requests services from a server.
* Clients **do not share resources** with other clients but depend on servers for processing.

**Examples of Clients:**

* **Web Browsers** (Google Chrome, Firefox, Edge)
* **Mobile Apps** (Instagram, Uber, Gmail)
* **Desktop Applications** (Outlook, Microsoft Teams)

**Server**

* A server is a machine or software that provides **services, processes requests, and manages data**.
* It can serve multiple clients simultaneously.

**Examples of Servers:**

* **Web Server** (Apache, Nginx, IIS) → Hosts websites.
* **Database Server** (MySQL, PostgreSQL, SQL Server) → Manages databases.
* **FTP Server** → Handles file transfers.
* **Game Server** → Hosts multiplayer games.

**Types of Servers:**

* **Dedicated Server:** Exclusively serves requests (e.g., a database server).
* **Shared Server:** Handles multiple services or applications (e.g., a web hosting server).

**How the Client-Server Model Works (Request-Response Cycle)**

The interaction follows these steps:

1. **Client Sends a Request**
   * A user takes action (e.g., clicks a link, submits a form).
   * The client generates an HTTP request and sends it to the server.
2. **Server Processes the Request**
   * The server **retrieves data**, **processes logic**, or **performs an action**.
   * If necessary, it communicates with a **database or another service**.
3. **Server Sends a Response**
   * The server returns a response to the client (HTML, JSON, XML, etc.).
   * If an error occurs (e.g., page not found), the server sends an error response (404 Not Found).
4. **Client Renders the Response**
   * The client (browser/app) processes and displays the response to the user.

**Advantages of the Client-Server Model**

**Scalability:** Servers can handle many clients at once.  
**Centralized Control:** Data and security are managed in one place (the server).  
**Resource Sharing:** Clients don’t need to store all data—they request it when needed.  
**Security:** Centralized security policies can be applied to protect data.

**Disadvantages of the Client-Server Model**

❌ **Server Overload:** Too many clients can slow down or crash the server.  
❌ **Single Point of Failure:** If the server fails, clients cannot function.  
❌ **Higher Cost:** Requires powerful hardware and maintenance.

• URLs

A **Uniform Resource Locator (URL)** is the address used to access resources on the internet. URLs are essential for web browsing, APIs, file access, and more.

A **URL is a string that specifies the location of a resource** on the web and how to access it. It allows browsers, servers, and applications to retrieve web pages, files, images, and other data.

Example of a URL:

<https://www.example.com:443/path/to/resource?query=123#section1>

In this example, the URL tells the browser to:

* Use **HTTPS** for communication.
* Contact the [**www.example.com**](http://www.example.com) server.
* Access a specific **path** (/path/to/resource).
* Include **query parameters** (?query=123).
* Jump to a specific **section** (#section1).

URL Components and Their Roles

A screenshot of a black box

Description automatically generated

|  |  |  |
| --- | --- | --- |
| **Scheme (Protocol)**  **The scheme defines the protocol used to access the resource. Common schemes include:**   * **http:// → Standard web communication (not secure).** * **https:// → Secure communication with encryption (SSL/TLS).** * **ftp:// → File Transfer Protocol for downloading/uploading files.** * **mailto: → Opens an email client (mailto:someone@example.com).** * **tel: → Initiates a phone call on supported devices (tel:+18001234567).**   **🔹 Why Use HTTPS?**   * **Encrypts data (prevents hackers from stealing information).** * **Protects against man-in-the-middle attacks.** * **Required for most modern websites (Google ranks HTTPS sites higher)**   **Host (Domain Name or IP Address)**  **The host identifies the web server that stores the requested resource.**   * **Domain Name (e.g., www.google.com) is human-readable.** * **IP Address (e.g., 192.168.1.1) is used for direct server communication.**   **🔹 DNS (Domain Name System) translates domain names into IP addresses.**  **Port (Optional)**   * **Defines the entry point for communication between client and server.** * **Common ports:**   + **80 (HTTP)**   + **443 (HTTPS)**   + **21 (FTP)**   + **3306 (MySQL database)** * **If omitted, browsers assume default values (80 for HTTP, 443 for HTTPS).**   **🔹 Example: http://localhost:5000/ → Uses port 5000**  **Query String (Optional)**   * **A query string sends data to the server using key-value pairs.** * **Starts with a ? and separates values with &.** * **Used for searches, filters, tracking, and API requests.**   **A black screen with white text  Description automatically generated** |  |  |

• Coding by Convention

**Coding by Convention (CbC)** is a software design paradigm that reduces the number of decisions developers need to make by establishing **default configurations, file structures, and naming conventions**.

Instead of requiring developers to specify configurations explicitly, a framework or system **assumes sensible defaults**, allowing developers to focus on writing application logic rather than boilerplate code.

**Example:**

In Ruby on Rails, if you create a model named User, Rails **automatically** assumes:

* The corresponding **database table** is named users.
* The table's **primary key** is id.
* The model's **file location** is app/models/user.rb

**Why is Coding by Convention Important?**

* **Reduces Configuration Overhead**

Without CbC, developers must explicitly define paths, relationships, and settings. CbC eliminates redundant configurations.

* **Improves Developer Productivity**

Developers can **focus on writing business logic** instead of setting up project structures and configurations.

* **Enhances Readability & Maintainability**

Projects following conventions are **easier to understand** because developers can predict file locations and class names.

* **Standardization Across Teams**

Teams using a convention-based approach can **collaborate more efficiently** because everyone follows the same structure.

**Spring Boot (Java)**

Spring Boot follows **convention-over-configuration** for web apps.

* If you create a class with @RestController, Spring **assumes** it handles web requests.
* **Default properties** like application.properties eliminate extra configurations.

**Example:**

**@RestController**

**@RequestMapping("/api/users")**

**public class UserController {**

**@GetMapping("/{id}")**

**public String getUser(@PathVariable Long id) {**

**return "User " + id;**

**}**

**}**

🔹 No need to manually configure **routing, HTTP methods, or request handling**.

• The MVC Pattern

**MVC (Model-View-Controller)** is a software design pattern used in web and application development. It separates an application into three interconnected components. By following MVC, applications become **more organized, maintainable, and scalable**.

* Models

The **Model** represents **data** and **business rules**. It interacts with the database and handles the application’s core logic.

**Key Responsibilities of Models:**

Define the **structure of data** (e.g., Task, User).  
Interact with the **database** (CRUD operations: Create, Read, Update, Delete).  
Enforce **business rules** (e.g., a user cannot register with the same email twice).

* Views

The **View** is responsible for **presenting data** to the user. It is the **UI layer** of the application.

**Key Responsibilities of Views:**

Display **data** received from the Model.  
Use **HTML, CSS, JavaScript** for UI design.  
Receive **dynamic content** from the Controller.

* Controllers

The **Controller** acts as the **middleman** between the **Model** and **View**. It processes user input and decides what data to send to the View.

**Key Responsibilities of Controllers:**

Receive **requests** from the user.  
Call the **appropriate Model** to fetch/update data.  
Pass data to the **View** for rendering.

* File Structure (What Goes Where)

A screenshot of a computer

Description automatically generated

* Purposes of Individual Files

**A. Controllers (Inside /Controllers/ Folder)**

* Each controller handles a specific type of request.
* Example: TaskController.cs manages tasks.

**B. Models (Inside /Models/ Folder)**

* Define the structure of application data.
* Example: TaskModel.cs represents a **Task** entity.

**C. Views (Inside /Views/ Folder)**

* Contains UI templates written in **Razor (.cshtml)**.
* Example: Task/List.cshtml displays a list of tasks.

**D. Static Files (Inside /wwwroot/ Folder)**

* **CSS** for styling.
* **JavaScript** for interactivity.
* **Images** for graphical content.

**E. Configuration Files**

* **appsettings.json** – Stores database connection strings, app settings.
* **Program.cs & Startup.cs** – Configure the app’s behavior.

• CSS files

CSS (**Cascading Style Sheets**) is a **stylesheet language** used to describe the **presentation of web pages**, including colors, layouts, fonts, and animations. It enables developers to separate the **content (HTML)** from the **design**, making websites more **flexible and maintainable**.

**A. Inline CSS (Applied directly to an HTML element)**

* Used for quick styling of individual elements.
* Not recommended for large projects due to poor maintainability.

🔹 **Pros**: Quick to implement.  
🔹 **Cons**: Hard to maintain, difficult to apply changes across multiple pages.

**B. Internal CSS (Defined inside <style> tags within the <head> section)**

* Used when styling a **single HTML document**.
* Useful for smaller projects.

🔹 **Pros**: No need for external files.  
🔹 **Cons**: Doesn't scale well for larger projects.

**C. External CSS (Stored in a separate .css file)**

* The best practice for professional web development.
* A single CSS file can be applied to **multiple pages**, ensuring **consistency**.

🔹 **Pros**: Easier to maintain, reusable across multiple pages.  
🔹 **Cons**: Requires an extra HTTP request to load the CSS file.

CSS follows a **selector { property: value; }** structure.

p {

color: red; /\* Text color \*/

font-size: 16px; /\* Font size \*/

}

**Breakdown:**

* **p** → Selector (targets all <p> elements)
* **color** → Property (what is being changed)
* **red** → Value (the new style for the property)

CSS (Cascading Style Sheets) is used in ASP.NET Core applications to style HTML elements and enhance the user experience. ASP.NET Core **follows a structured approach** for organizing CSS files to ensure modularity and maintainability.

**How to Link CSS Files in ASP.NET Core?**

CSS files are included in **Razor Views** (.cshtml files) using the <link> tag.

**A. Adding CSS in \_Layout.cshtml (Global Styles)**

The **\_Layout.cshtml** file defines the layout for multiple pages and is commonly used to include global CSS files.

Example:

<!-- Link to CSS files -->

<link rel="stylesheet" href="~/css/site.css" />

<link rel="stylesheet" href="~/css/custom.css" />

🔹 **~/css/site.css** uses the tilde (~) notation, which represents the root path of wwwroot.

**B. Adding CSS in a Specific View (Index.cshtml)**

If you want to include a CSS file **only for a specific page**, you can do this in the individual view.

**Using Tag Helpers to Load CSS Files**

ASP.NET Core provides **Tag Helpers**, which simplify the process of including CSS files dynamically.

**Using the asp-append-version Attribute**

To **prevent caching issues**, you can use asp-append-version="true". This ensures browsers always load the latest CSS version.

<link rel="stylesheet" href="~/css/site.css" asp-append-version="true" />

🔹 This appends a version query string (site.css?v=xyz123) to the file, forcing the browser to load the newest version.

**Using External CSS Frameworks (e.g., Bootstrap, Tailwind)**

You can **integrate Bootstrap, Tailwind, or other CSS frameworks** in ASP.NET Core by:

**A. Using CDN (Content Delivery Network)**

This loads the CSS from an external server without storing it locally.

<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css">

**B. Installing Bootstrap via LibMan (Library Manager)**

LibMan allows you to install and manage front-end libraries in ASP.NET Core.

1. **Right-click the project** → Select **Manage Client-Side Libraries**
2. Install Bootstrap
3. Add it to \_Layout.cshtml:

<link rel="stylesheet" href="~/lib/bootstrap/css/bootstrap.min.css" />

• Middleware

Middleware in **ASP.NET Core** is a **pipeline component** that processes HTTP requests and responses. Each middleware component in the pipeline:

* Can **handle** a request.
* Can **modify** the request before passing it to the next middleware.
* Can **modify** the response before sending it back to the client.

Middleware is **executed in order**, making the request processing flow customizable.

When an HTTP request reaches an **ASP.NET Core application**, it travels through a **pipeline** of middleware components **before reaching the endpoint** (such as a controller).

🛠 **Example of a middleware pipeline:**

1. **Logging Middleware** → Logs request details.
2. **Authentication Middleware** → Validates user credentials.
3. **Routing Middleware** → Determines the correct endpoint.
4. **MVC Middleware** → Executes the controller action.
5. **Response Middleware** → Modifies the response before sending it back.

**Registering Middleware in Program.cs**

Middleware is added inside the Program.cs file

🔹 **Breakdown:**

* app.Use(...) → Registers middleware.
* await next.Invoke(); → Passes the request to the next middleware.
* app.UseRouting(); → Enables routing.
* app.UseEndpoints(...); → Maps endpoints for requests.

A screenshot of a computer program

Description automatically generated

Besides built-in middleware, **custom middleware** can be created for specific logic.

**Use, Run, and Map Middleware Methods**

ASP.NET Core provides three methods to define middleware:

**A. Use() - Passes request to the next middleware**

**Example:**

app.Use(async (context, next) =>

{

Console.WriteLine("Middleware 1: Before");

await next.Invoke(); // Calls the next middleware

Console.WriteLine("Middleware 1: After");

});

B. Run() - Ends the middleware pipeline

Example:

app.Run(async context =>

{

await context.Response.WriteAsync("Final Middleware - No Next Call");

});

**Note:** Run() is a terminal middleware (does not call next)

C. Map() - Branches the pipeline

Example:

app.Map("/admin", adminApp =>

{

adminApp.Run(async context =>

{

await context.Response.WriteAsync("Admin Dashboard");

});

});

If a request starts with /admin, it executes this middleware.

The order of middleware **matters** because requests **pass through each middleware in sequence**.

In **ASP.NET Core**, the **pipeline** refers to the sequence of **middleware components** that **process an HTTP request and response**.

Think of the **pipeline** as an **assembly line**:

* A **request** (e.g., user visits a webpage) enters the pipeline.
* Each middleware component **processes** the request or **modifies** it before passing it to the next middleware.
* Eventually, the request **reaches the final destination** (like an MVC controller or Razor page).
* The **response** then follows the **same path back**, allowing middleware to modify it before it’s sent to the user.

**How the Pipeline Works:**

1. The user **sends a request** (e.g., visiting www.example.com/home).
2. The request **enters the pipeline**.
3. Middleware components **process** the request.
4. The **final destination (controller/view)** is reached.
5. The **response** is generated.
6. Middleware **modifies the response** (if needed).
7. The response is **sent back to the user**.

Middleware is a **software component** that handles **requests and responses** in the pipeline. Each middleware **performs a specific task** before passing the request to the next middleware.

**Middleware Functions:**

**Logging** (e.g., tracking incoming requests)  
**Authentication & Authorization** (e.g., checking if a user is logged in)  
**Routing** (e.g., deciding which controller should handle the request)  
**Static Files** (e.g., serving CSS, JavaScript, images)  
**Error Handling** (e.g., showing an error page for broken links)

Each middleware can:

* **Process the request** before passing it to the next middleware.
* **Modify the response** before it is sent back to the client.

• Razor Code / Razor Commands

**Razor Code**

* **Definition**: Razor Code refers to the **C# logic written inside Razor views (.cshtml files)**.
* **Purpose**: Used to **process data**, **perform logic**, and **generate dynamic HTML**.
* **Syntax**: Written inside @{ } or using inline expressions with @.
* **Example Use Cases**:
  + Declaring variables
  + Performing calculations
  + Using loops and conditions

Example:

@{

var currentTime = DateTime.Now;

}

<p>The current time is @currentTime</p>

@for (int i = 1; i <= 5; i++)

{

<p>Item @i</p>

}

* Declares a variable currentTime and displays it in HTML.
* Uses a for loop to generate <p> elements dynamically.

**Razor Commands (Directives)**

* **Definition**: Razor Commands are **special directives** that control **how Razor processes a view**.
* **Purpose**:
  + Manage **model binding**, **layout settings**, **namespaces**, **dependency injection**, etc.
  + **Do not execute logic** like Razor Code; instead, they **configure how the page behaves**.
* **Syntax**: Commands start with @ followed by a keyword (@model, @section, @inject, etc.).
* **Example Use Cases**:
  + Declaring the **model type** (@model)
  + Importing **namespaces** (@using)
  + Specifying **layout pages** (@layout)

Example:

@model MyApp.Models.User <!-- Declares the Model -->

@using System.Collections.Generic <!-- Imports a namespace -->

<h2>Welcome, @Model.Name</h2>

* @model **binds the view to a specific model** (User class).
* @using **imports namespaces** to use in the view.

A screenshot of a computer

Description automatically generated

Razor is a **lightweight, server-side templating engine** used in **ASP.NET Core** for creating dynamic web pages. It allows developers to **embed C# code inside HTML** to create interactive and data-driven web applications.

**Razor** is a **syntax** used in ASP.NET Core **to mix C# and HTML** inside a view file. It is used in **Razor Pages** and **MVC Views (.cshtml)** to create dynamic content.

✔ **Fast and lightweight**  
✔ **Uses C# syntax** inside HTML  
✔ **Reduces the need for JavaScript in simple scenarios**  
✔ **Supports Model-View-Controller (MVC) pattern**

Example:

@{

var name = "John";

}

<h1>Welcome, @name!</h1>

**Explanation:**

* @{} encloses **C# code**.
* @name **outputs the variable** inside HTML.

Razor code is written inside **@{}** to define variables, loops, and conditions.

**Example: Code Block**

@{

int year = DateTime.Now.Year;

}

<p>The current year is @year.</p>

Example: If-Else Statement

@{

bool isLoggedIn = true;

}

@if (isLoggedIn)

{

<p>Welcome, user!</p>

}

else

{

<p>Please log in.</p>

}

Example: Loop in Razor

@for (int i = 1; i <= 5; i++)

{

<p>Item @i</p>

}

Uses **C# loops** to generate **repeating HTML elements** dynamically.

Razor provides special **commands** for working with **HTML and C# code** inside views.

**A. Output Variables (@variable)**

**@{**

**string message = "Hello, Razor!";**

**}**

**<p>@message</p>**

**B. Displaying Expressions (@(expression))**

**<p>The sum of 5 and 3 is @(5 + 3).</p>**

**C. Using Inline Conditions (@())**

**<p>@(DateTime.Now.Hour < 12 ? "Good morning!" : "Good evening!")</p>**

**D. Working with Strings and HTML**

**Razor automatically escapes HTML to prevent XSS attacks.**

**@{**

**string rawHtml = "<strong>Important</strong>";**

**}**

**<p>@rawHtml</p> <!-- Output: <p>&lt;strong&gt;Important&lt;/strong&gt;</p> -->**

**If you want to render raw HTML, use @Html.Raw()**

**<p>@Html.Raw(rawHtml)</p> <!-- Output: <p><strong>Important</strong></p> -->**

**Razor directives start with @ and control how Razor behaves.**

**A. @model (Model Binding)**

**Used to bind a model to the view.**

**@model MyApp.Models.User**

**<h2>Welcome, @Model.Name</h2>**

**Model.Name accesses properties of the model passed from the controller.**

**B. @using (Import Namespace)**

**Used to import namespaces in a Razor view.**

**@using System.Collections.Generic**

**@using MyApp.Models**

**C. @inject (Dependency Injection)**

**Injects services into a view.**

**@inject MyApp.Services.IUserService UserService**

**<p>User Count: @UserService.GetUserCount()</p>**

**D. @section (Define Layout Sections)**

**Used for custom sections inside a layout file (\_Layout.cshtml).**

**@section Scripts {**

**<script>alert("This is a custom script!");</script>**

**}**

**E. @RenderBody() (Render Main Content)**

**Used in layout pages to render the child view’s content.**

**F. @RenderSection() (Render Optional Sections)**

**Used in layout files to insert optional sections from child views.**

• Controllers

* A **Controller** is a class that handles HTTP requests and returns responses.
* It acts as the **middleman** between the **Model (data)** and the **View (UI)**.
* Controllers **contain action methods** (endpoints) that process requests and generate responses.

**Key Features of Controllers**

Receives **requests** from the client (browser, API call, etc.).  
Processes data using **Models** or **Services**.  
Returns **Views (HTML)** or **JSON (for APIs)**.  
Uses **routing** to determine which action to execute.

The HomeController contains two **action methods**: Index() and About().

These methods return **views** that are rendered as HTML.

• Actions

* An **action** is a **method inside a controller** that **handles a specific request**.
* It determines what response to send back to the client.
* Actions **must be public** and **return an IActionResult or another result type**.

**Types of Actions**

**Return a View (for UI-based apps) – ex) return View();**  
**Return JSON (for APIs)**  
**Redirect to another action – ex) return RedirectToAction("Index", "Home"); - Redirects to Index() in HomeController**  
**Return an HTTP status code**

• Contstructors

* A **constructor** is a special method in a controller that runs when an instance of the controller is created.
* Used to **initialize dependencies** (like services, repositories, or databases).

**Why Use a Constructor?**

Implements **Dependency Injection (DI)**.  
Initializes required services.  
Improves **testability** of controllers.

**What Can Be Injected in a Constructor? – inside ()**

✔ **Services (IService)** – Business logic services  
✔ **Database Context (DbContext)** – For database access  
✔ **Repositories (IRepository)** – To access data sources  
✔ **Logging (ILogger<T>)** – For logging events

A screenshot of a computer program

Description automatically generated

• Static vs. Non-Static Classes

In C#, classes can be **static** or **non-static** (also called instance classes). The choice between them depends on whether you need to create instances of the class or just use it as a utility with shared functionality.

**Static Classes**

A **static class** is a class that **cannot be instantiated** and only contains **static members** (methods, properties, and fields).

**Characteristics of a Static Class**

✅ **Cannot be instantiated** (you can't use new).  
✅ **Only contains static members** (methods, fields, properties).  
✅ **Used for utility/helper methods** (e.g., mathematical operations, logging).  
✅ **Improves performance** (no object creation, directly accesses members).  
✅ **Cannot inherit from another class** (except System.Object implicitly).

Example:

int sum = MathHelper.Add(5, 10); // ✅ Directly calling the method double sqrt = MathHelper.SquareRoot(25);

**Notice**: We don’t create an instance of MathHelper. We directly call its methods using the class name.

**Non-Static (Instance) Classes**

A **non-static class** is a regular class that can be instantiated with the new keyword.

**Characteristics of a Non-Static Class**

✅ **Can be instantiated** (new keyword is used).  
✅ **Can have instance members** (fields, properties, methods).  
✅ **Can support object-oriented features** (inheritance, polymorphism, encapsulation).  
✅ **Can implement interfaces**.

Example:

Car myCar = new Car { Model = "Toyota", Year = 2022 };

myCar.DisplayInfo(); // ✅ Instance method call

**Notice**: We created an instance (myCar) before calling the method.

A screenshot of a black screen

Description automatically generated

**When to Use Static vs. Non-Static Classes**

✅ **Use Static Classes When**:

* You need **utility functions** (e.g., Math, Logging).
* You don’t need to store data in instances.
* You want to **avoid creating multiple instances** (e.g., ConfigurationManager).

✅ **Use Non-Static Classes When**:

* You need to **store and manipulate object state** (e.g., User, Order).
* You need to **implement interfaces or inheritance**.
* You need **dependency injection** (used in ASP.NET Core services).

Use instance classes for Dependency Injection

Dependency Injection (DI) is a **design pattern** that allows **loose coupling** between objects by injecting their dependencies from an external source rather than creating them internally.

Example:

Program.cs

builder.Services.AddScoped<IMyService, MyService>(); // Dependency Injection

**AddScoped<IMyService, MyService>()** registers MyService for DI.

Once registered, you can **inject dependencies** into controllers using **constructor injection**.

// Dependency Injection in Constructor

public HomeController(IMyService myService)

{ \_myService = myService; }

**ASP.NET Core automatically provides an instance** of IMyService.

We define a service (IMyService) and its implementation (MyService).

public class MyService : IMyService

A screenshot of a computer program

Description automatically generated

• Return statements

A return statement in ASP.NET Core **sends a response** from a method back to the caller. It is used in **Controllers**, **Razor Pages**, **APIs**, and other parts of an application.

In ASP.NET, return statements are often used inside **controller actions** to send back **HTML views, JSON data, HTTP status codes, or files**.

Controllers in ASP.NET Core use return statements to specify the response type. Here are common return types:

**Returning a View**

For returning an **HTML page (View)**:

public IActionResult Index()

{

return View(); // Returns the "Index.cshtml" view

}

**If a view name isn't specified,** it defaults to a view with the same name as the action method.

Returning a Specific View:

public IActionResult About()

{

return View("AboutUs"); // Renders the "AboutUs.cshtml" view

}

Returning a View with Data:

public IActionResult Details(int id)

{

var student = \_db.Students.Find(id);

return View(student); // Passes the student data to the view

}

Data is sent to the view using **View Models or the ViewBag/ViewData**

**Returning Redirects**

**Redirects are used to send users to another action, controller, or external URL.**

**Redirect to Another Action:**

**public IActionResult GoToHome()**

**{**

**return RedirectToAction("Index"); // Redirects to Index action**

**}**

**Returning HTTP Status Codes**

You can return different **HTTP status codes** using built-in methods.

**Example:**

**Returning Success (200 OK) - Pk()**

Returning Bad Request (400) – BadRequest()

**Returning Files**

If you need to allow users to **download a file**, use File().

**Returning Content (Plain Text or HTML)**

If you want to return **plain text** or **raw HTML**, use Content().

• .UseDefaultFiles()

**middleware components** that configure how requests are handled in the application. These methods are used in the Program.cs within the Configure method to define the **request pipeline. They are typically added in the Program.cs file inside the app.Use... pipeline.**

**What It Does:**

* Enables serving **default files** (like index.html, default.html) when the user requests the root URL (/).
* Works **before** UseStaticFiles() and is typically used **with** it.
* If enabled, it **automatically loads** the default file without needing to specify it in the URL.

🔹 If index.html exists in wwwroot, a request to / will automatically return index.html.

**How It Works:**

1. When a user visits /, **without specifying a file**, UseDefaultFiles() checks for:
   * index.html
   * default.html
   * index.htm
   * default.htm
2. If found, it **rewrites the request** to that file.
3. The file is then served by UseStaticFiles().

**Without .UseDefaultFiles()**

If you only use UseStaticFiles(), users must type **the full filename** in the URL:

https://example.com/index.html ✅ Works

https://example.com/ ❌ 404 Not Found

**With .UseDefaultFiles()**

The index.html is **automatically served** when visiting /:

https://example.com/ ✅ Loads index.html

.UseDefaultFiles() **only modifies requests**, it does **not serve the file itself**.

• .UseStaticFiles()

**What It Does:**

* Serves **static files** (like HTML, CSS, JavaScript, images, PDFs) from the wwwroot folder.
* Without UseStaticFiles(), requests for static files return **404 Not Found**.

🔹 Now, if a CSS file exists at wwwroot/css/style.css, you can access it via:

https://example.com/css/style.css ✅

You can configure it to serve files from a different folder.

• .UseRouting()

**What It Does:**

* Enables **endpoint routing**, which is needed for MVC controllers, Razor Pages, and minimal APIs.
* Defines how URLs are **mapped to controllers and actions**.

**How It Works in Middleware Order**

1. .UseRouting() **analyzes the request URL** but does **not** execute any controller action yet.
2. After routing is set up, **middleware like authentication or authorization** can be added.
3. .UseEndpoints() executes the **final matching route**.

.UseRouting() defines how requests should be matched, but **it does not execute the matched route**. You must use .UseEndpoints() after it.

Example:

app.UseRouting();

app.UseEndpoints(endpoints =>

{

endpoints.MapControllerRoute(

name: "default",

pattern: "{controller=Home}/{action=Index}/{id?}");

});

🔹 This maps URLs like:

https://example.com/Home/Index ✅ Maps to HomeController.Index()

https://example.com/Products/Details/5 ✅ Maps to ProductsController.Details(5)

Middleware must be **added in the correct order** for it to work properly.

var app = builder.Build();

app.UseRouting(); // Enable endpoint routing

app.UseDefaultFiles(); // Load index.html automatically

app.UseStaticFiles(); // Serve CSS, JS, images

app.UseAuthorization(); // Apply authentication if needed

app.UseEndpoints(endpoints =>

{

endpoints.MapControllers(); // Enable MVC controllers

});

• Forms

Forms in **ASP.NET Core MVC** are used to collect user input and send it to the server for processing. The most common use case is handling data entry in web applications. They are a crucial part of any web application that requires user interaction, such as login, registration, or data entry.

A form typically consists of:

* **Input fields** (text, email, password, dropdowns, etc.)
* **A submit button**
* **A method** (GET or POST)
* **Validation rules** (client-side and server-side)

**How Forms Work**

* A **form** in MVC is defined using the <form> HTML tag.
* Forms **send data** using either the **GET** or **POST** method.
* The **controller** processes the form data and takes action (such as saving it to a database).

Example:

<form asp-action="Create" asp-controller="Home" method="post">

asp-action="Create" → Specifies that the form submits to the Create action.

asp-controller="Home" → Specifies the controller handling the request.

method="post" → Uses HTTP POST for submission.

HomeController:

[HttpPost] → Ensures the method only handles **POST** requests.

ModelState.IsValid → Validates the form inputs before processing.

Instead of manually extracting form values, **ASP.NET Core MVC** uses **model binding**.

public class UserModel

{

[Required]

public string Name { get; set; }

[Required]

[EmailAddress]

public string Email { get; set; }

}

* [Required] → Ensures that the field cannot be empty.
* [EmailAddress] → Ensures valid email format.

<form asp-action="SubmitForm" method="post"> <label for="name">Name:</label> <input asp-for="Name" /> <span asp-validation-for="Name" class="text-danger"></span></form>

asp-for="Name" → Binds input field to UserModel.Name.

asp-validation-for="Name" → Displays validation errors.

**Client-Side Validation with jQuery**

To enable client-side validation:

1. Include jQuery and jQuery Validation scripts in \_Layout.cshtml:

<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.6.0/jquery.min.js"></script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery-validate/1.19.3/jquery.validate.min.js"></script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery-validation-unobtrusive/3.2.11/jquery.validate.unobtrusive.min.js"></script>

1. Ensure asp-validation-for is used in the form.

Now, validation messages will **appear instantly** when users enter incorrect data.

**Redirecting After Form Submission**

After form submission, redirect users to another page using RedirectToAction().

This **prevents resubmission** when users refresh the page.

• Navigation (Routing) in MVC

Routing in **ASP.NET Core MVC** is how URLs map to **controllers** and **actions**. The framework uses **routing middleware** to determine how HTTP requests should be handled.

**Types of Routing**

1. **Conventional Routing** (Defined globally)
2. **Attribute Routing** (Defined on controllers or actions)

**1. Conventional Routing**

Defined in Program.cs

app.UseRouting();

app.UseEndpoints(endpoints =>

{

endpoints.MapControllerRoute(

name: "default",

pattern: "{controller=Home}/{action=Index}/{id?}");

});

{controller=Home} → Default controller is HomeController.

{action=Index} → Default action is Index().

{id?} → Optional parameter for ID.

**2. Attribute Routing**

Defined **inside the controller** using [Route] attributes.

[Route("products")]

public class ProductController : Controller

{

[Route("details/{id}")]

public IActionResult Details(int id)

{

return View();

}

}

products/details/5 will route to Details(int id).

• Inheritance

**Inheritance** is a fundamental concept in **Object-Oriented Programming (OOP)** that allows classes to reuse and extend functionality. In ASP.NET Core MVC, inheritance is commonly used for:

1. **Controllers** (Inheriting from Controller)
2. **Models** (Inheriting base classes)
3. **Custom Base Classes** (Reusing code)

Allowing a class (**child/derived class**) to inherit **fields, properties, and methods** from another class (**parent/base class**). In **ASP.NET Core**, inheritance is widely used to **reuse code, maintain consistency, and follow best coding practices**.

**Definition**

Inheritance allows a **child class** to derive from a **parent class**, meaning:

* The child class **inherits** properties and methods from the parent.
* The child class **can override** or extend functionality.
* Code is **reused and structured more efficiently**

**1. Inheritance in Controllers**

All controllers in ASP.NET Core **inherit from Controller**.

public class HomeController : Controller

{

public IActionResult Index()

{

return View();

}

}

The HomeController inherits methods like View(), RedirectToAction(), and Json() from Controller.

**2. Inheritance in Models**

A model can inherit from another class to share common properties.

public class Person

{

public string Name { get; set; }

public int Age { get; set; }

}

public class Student : Person

{

public string StudentID { get; set; }

}

Student **inherits** Name and Age from Person.

**3. Creating a Custom Base Controller**

You can create a **custom base controller** to share logic among multiple controllers.

public class BaseController : Controller

{

protected void Log(string message)

{

Console.WriteLine(message);

}

}

public class HomeController : BaseController

{

public IActionResult Index()

{

Log("Home Page Accessed");

return View();

}

}

HomeController inherits the Log() method from BaseController.

Views in ASP.NET Core **inherit from Layouts**.

Child View Using Layout:

@{ Layout = "\_Layout"; }

\_Layout.cshtml is **inherited** by multiple views, ensuring **consistent headers, footers, and styles**.

A screenshot of a black and white screen

Description automatically generated

Example:

public class Parent

{

public virtual void ShowMessage()

{

Console.WriteLine("Parent Message");

}

}

public class Child : Parent

{

public override void ShowMessage()

{

Console.WriteLine("Child Message");

}

}

**virtual** → Allows overriding in derived classes.

**override** → Modifies the method in the child class.

**abstract** → Forces child classes to implement MakeSound().

Sealed -> class **prevents inheritance**.

In **ASP.NET Core**, interfaces (IService, IRepository, etc.) are often used instead of inheritance.

• Interface vs. Class

In **ASP.NET Core**, both **interfaces** and **classes** are fundamental building blocks for designing maintainable, reusable, and testable code. While **classes** define objects with properties and behaviors, **interfaces** define contracts that classes must follow.

**Class**

**Definition**

A **class** is a **blueprint** for creating objects that contain **fields (data) and methods (behavior)**. A class can have:

* Properties (get, set)
* Methods (functions)
* Constructors (initialize objects)
* Inheritance (extend another class)

Example:

public class Car

{

public string Model { get; set; }

public int Speed { get; set; }

public void Drive()

{

Console.WriteLine($"{Model} is driving at {Speed} mph.");

}

}

A class **can have implementations** (method definitions).  
A class **can inherit from another class** (single inheritance).  
Objects are created from a class using new.

**Interface**

**Definition**

An **interface** is a contract that **defines method signatures but does not provide implementations**. It ensures that classes follow a specific structure.

Example:

public interface IVehicle

{

void Start(); // No implementation

void Stop();

}

Usage:

public class Bike : IVehicle

{

public void Start()

{

Console.WriteLine("Bike started.");

}

public void Stop()

{

Console.WriteLine("Bike stopped.");

}

}

An interface **cannot have implementations** (only method signatures).  
A class **can implement multiple interfaces** (unlike classes, which support only single inheritance).  
Interfaces **ensure consistency** across multiple classes.

A screenshot of a black screen

Description automatically generated  
Interfaces make **Dependency Injection (DI)** possible by allowing us to inject dependencies instead of creating objects inside a class.

public class EmailService : IEmailService

{

public void SendEmail(string recipient, string subject, string body)

{

Console.WriteLine($"Email sent to {recipient} with subject: {subject}");

}

}

Now, we **inject** the service using DI instead of creating a new object:

public class NotificationController

{

private readonly IEmailService \_emailService;

public NotificationController(IEmailService emailService)

{

\_emailService = emailService;

}

public void NotifyUser()

{

\_emailService.SendEmail("user@example.com", "Welcome", "Hello!");

}

}

✅ This makes the code **loosely coupled**, meaning we can easily swap EmailService with another implementation.

• Data Annotations

Data Annotations in **ASP.NET Core** are **attributes** used to define validation rules, display formatting, and behavior for model properties. They help enforce **business rules** directly in the model and integrate seamlessly with **Model Binding and Validation** in ASP.NET Core MVC.

**Validate user input** before saving data to a database.  
**Improve readability** by specifying metadata (like display names).  
**Reduce boilerplate code** by defining rules inside models.  
**Seamlessly integrate** with ASP.NET Core's **Model Binding and Validation system**.

A screenshot of a computer

Description automatically generated

Example:

[Required(ErrorMessage = "Name is required")] [StringLength(50, MinimumLength = 2, ErrorMessage = "Name must be between 2-50 characters.")]

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

Sometimes built-in validation isn't enough. You can **create custom validation attributes** by inheriting from ValidationAttribute.

In **MVC**, data annotations work with the **Model Validation system**.

**How It Works**

1. **User submits form** → ASP.NET Core automatically checks data annotations.
2. **ModelState.IsValid** checks if all validation rules are met.
3. If **valid**, proceed; otherwise, **return form with errors**.

ASP.NET Core automatically integrates **data annotations** with **Tag Helpers**.

• Tag Helpers

**Tag Helpers** in **ASP.NET Core** are a **server-side** feature that allows you to create dynamic HTML elements in Razor views using **C# code**, making them more readable and maintainable. They enable **server-side processing** while keeping HTML syntax intact.

✅ **They provide an alternative to traditional HTML Helpers (@Html syntax) and make Razor views cleaner and more HTML-like.**

**Why Use Tag Helpers?**

✔ **Improves Readability** – Looks like standard HTML, making it easier to work with.  
✔ **Intellisense Support** – Works with Visual Studio’s IntelliSense, unlike HTML Helpers.  
✔ **Encapsulation** – Helps separate concerns between HTML and server-side logic.  
✔ **Reusable** – Can be customized and extended.

ASP.NET Core provides **built-in Tag Helpers** for **forms, inputs, links, images, and more**.

A screenshot of a computer program

Description automatically generated

**B. Anchor (<a>) and Link Tag Helpers**

Helps generate **dynamic links** based on routing.

**Example: Generating a Link to a Controller Action**

<a asp-controller="Home" asp-action="Details" asp-route-id="10">View Details</a>

Generates:

<a href="/Home/Details/10">View Details</a>

**C. Image (<img>) Tag Helper**

Ensures images load correctly by resolving **static file paths**.

<img asp-append-version="true" src="/images/logo.png" alt="Logo" />

**D. Environment Tag Helper**

Used to **conditionally include scripts/styles** based on the **environment** (Development, Staging, Production).

**Example: Load Different Scripts Based on Environment**

<environment include="Development">

<script src="dev-script.js"></script>

</environment>

<environment exclude="Development">

<script src="prod-script.js"></script>

</environment>

**E. Caching Tag Helper**

Used to cache parts of a Razor view to **improve performance**.

**Example: Cache a Section for 1 Hour**

You can create your **own Tag Helpers** for reusable components.

• C# Arrays

An **array** is a **fixed-size collection** of elements of the **same data type**.

* Arrays are **zero-indexed** (first element at index 0).
* Their **size is fixed** once declared.

A. Declare an Array (Fixed Size)

int numbers = new int[5]; // An array with 5 elements

B. Initialize an Array with Values

string[] names = { "Alice", "Bob", "Charlie" };

C. Accessing Array Elements

Console.WriteLine(names[1]); // Output: Bob

D. Modifying an Array Element

names[2] = "David"; // Changes "Charlie" to "David"

| **Operation** | **Example** |
| --- | --- |

|  |  |
| --- | --- |
| **Find Length** | numbers.Length |

|  |  |
| --- | --- |
| **Loop Through** | foreach (var num in numbers) { } |

|  |  |
| --- | --- |
| **Sort** | Array.Sort(numbers); |

|  |  |
| --- | --- |
| **Reverse** | Array.Reverse(numbers); |

**Limitations of Arrays**

* Fixed size (cannot add/remove elements dynamically).
* Use **Lists** if dynamic resizing is required.

• Nullable Values

**What is a Nullable Type?**

By default, **value types (int, double, bool, etc.) cannot be null**.  
A **nullable type** allows assigning null to value types using ?.

Use .HasValue or == null to check if a nullable variable has data.

The ?? operator assigns a **default value** if null.

int finalAge = age ?? 18; // If age is null, assign 18

• List

A **List<T>** in C# is a **dynamic collection** that allows adding, removing, and modifying elements **at runtime**.

* Unlike arrays, **Lists are resizable**.
* Defined in System.Collections.Generic.

**Declaring and Initializing a List**

**List<string> students = new List<string>(); // Empty list**

**Initializing with Values**

**List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };**

| **Operation** | **Example** |
| --- | --- |

|  |  |
| --- | --- |
| **Add Item** | **students.Add("John");** |

|  |  |
| --- | --- |
| **Remove Item** | **students.Remove("John");** |

|  |  |
| --- | --- |
| **Check Count** | **students.Count;** |

|  |  |
| --- | --- |
| **Sort** | **students.Sort();** |

|  |  |
| --- | --- |
| **Find Element** | **students.Contains("Alice");** |

|  |  |
| --- | --- |
| **Clear List** | **students.Clear();** |

• NuGet Package Manager

NuGet is the **package manager** for .NET, including **ASP.NET Core**. It provides a way to install, update, and manage third-party libraries and tools for .NET projects.

* **Think of it like** npm (for JavaScript) or pip (for Python).
* NuGet **packages** contain reusable **.NET code**, dependencies, and configuration files.
* The official package source is **NuGet.org**.

**Why Use NuGet?**

✅ **Saves Time** – No need to manually download and configure libraries.  
✅ **Manages Dependencies** – Automatically resolves and installs required libraries.  
✅ **Keeps Packages Updated** – Allows easy updates for security and performance.  
✅ **Ensures Compatibility** – Installs the correct versions for your .NET framework.

A screenshot of a computer program

Description automatically generated

**Where Are Packages Stored?**

* Locally: **C:\Users\<YourUser>\.nuget\packages\**
* Project-Level: Inside the **packages/** folder or as **PackageReferences** in .csproj
* Globally: Managed by .nuget.config

A screenshot of a black box

Description automatically generated

• “model” vs. “Model”

**"model" (Lowercase 'm') – A Naming Convention in Razor Views**

* In **Razor views (.cshtml files)**, the lowercase model is a **directive** that defines the type of data the view will receive.
* This allows strong typing, meaning the view knows what properties and methods are available on the model.

**Example: Using @model in a Razor View**

@model MyApp.Models.Movie

<h1>@Model.Title</h1>

<p>Release Year: @Model.ReleaseYear</p>

* @model **declares the model type** (MyApp.Models.Movie).
* @Model **references the actual model instance** passed to the view.

🔹 **Key Notes:**

* Always use @model **at the top of a Razor view**.
* The **first letter is lowercase** (@model).
* It defines **the type of model** the view will use.

**"Model" (Uppercase 'M') – Refers to the Data Model in MVC**

* The **Model (uppercase 'M')** is a class representing the **data structure** used in the application.
* It often maps to a **database table** (if using **Entity Framework**).
* It is placed inside the **Models** folder in an ASP.NET project.

**Example: A Model Class (Movie.cs)**

namespace MyApp.Models

{

public class Movie

{

public int Id { get; set; }

public string Title { get; set; }

public int ReleaseYear { get; set; }

}

}

This **Movie** class represents a **database entity**.

It is used for **data processing, validation, and business logic**.

A screenshot of a computer

Description automatically generated

• Model-First (Code-First) vs. DatabaseFirst

In **ASP.NET Core with Entity Framework (EF Core)**, there are two main approaches to designing and managing the database:

1. **Model-First (Code-First)**
2. **Database-First**

Each approach has its own use case depending on project requirements.

**1. Model-First (Code-First) Approach**

**Definition:**

* Developers **define database tables using C# classes** (models) instead of manually creating tables in a database.
* Entity Framework Core **automatically generates the database schema** based on these model classes.
* Migrations are used to **apply changes** to the database as the models evolve.

**When to Use Code-First?**

✅ When you want **full control over your models** and prefer writing C# code over SQL.  
✅ When you **start a project from scratch** and don't have an existing database.  
✅ When you want to use **migrations** to manage database updates.

1. Define the Model (Entity Class)

public class Movie

{

public int Id { get; set; }

public string Title { get; set; }

public int ReleaseYear { get; set; }

}

2. Create a DbContext Class

using Microsoft.EntityFrameworkCore;

public class ApplicationDbContext : DbContext

{

public DbSet<Movie> Movies { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder options)

{

options.UseSqlite("Data Source=movies.db"); // SQLite database

}

}

* migrations add InitialCreate → Generates a migration file based on the Movie model.
* database update → Applies the migration and creates the database schema.

3. Apply Migrations to Generate the Database

dotnet ef migrations add InitialCreate

dotnet ef database update

**2. Database-First Approach**

**Definition:**

* Developers **start with an existing database** and then generate C# classes based on the database schema.
* Entity Framework Core **reverse engineers** the database into entity classes and a DbContext.

**When to Use Database-First?**

✅ When you already have a **pre-existing database**.  
✅ When the **database schema is designed by a database administrator (DBA)**.  
✅ When working with **legacy systems** that already have a structured database.

**1. Scaffold the Database Context and Models**

Run this command to generate models from an existing database:

dotnet ef dbcontext scaffold "Server=myserver;Database=mydb;User=myuser;Password=mypassword;" Microsoft.EntityFrameworkCore.SqlServer -o Models

* This command **automatically creates C# entity classes** inside the Models folder.
* It also generates a DbContext class to interact with the database.

2. Example of Auto-Generated DbContext

public class ApplicationDbContext : DbContext

{

public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options) { }

public DbSet<Movie> Movies { get; set; }

}

A screenshot of a computer program

Description automatically generated

• Migrations

ASP.NET Core uses **Entity Framework Core (EF Core)** as an **Object-Relational Mapper (ORM)** to interact with databases.  
Three key concepts in EF Core are:

1. **Migrations** – Managing database schema changes.
2. **Scaffolding** – Auto-generating code based on existing structures.
3. **Entity Framework** – The ORM that bridges C# and SQL databases.

**Migrations** are used to keep the **database schema in sync** with C# models.  
Instead of manually updating tables, **EF Core Migrations** allow developers to apply database changes **incrementally**.

**How Migrations Work?**

1. **Define a C# model** (Class → Table)
2. **Run migration commands** to update the database
3. **EF Core generates SQL scripts** behind the scenes

1. Define a Model

public class Movie

{

public int Id { get; set; }

public string Title { get; set; }

public int ReleaseYear { get; set; }

}

2. Create a DbContext Class

public class ApplicationDbContext : DbContext

{

public DbSet<Movie> Movies { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder options)

{

options.UseSqlite("Data Source=movies.db"); // SQLite database

}

}

3. Run Migration Commands in Terminal

dotnet ef migrations add InitialCreate

dotnet ef database update

- migrations add InitialCreate → Generates a migration file with schema changes.

- database update → Applies the migration to the actual database.

**Up()** → Defines changes to apply.

**Down()** → Defines rollback actions.

• Scaffolding

**Scaffolding** is used to **automatically generate** CRUD (Create, Read, Update, Delete) operations based on a model.

**When to Use Scaffolding?**

✅ To quickly generate **controllers and views** for a database table.  
✅ Saves development time by **auto-generating CRUD code**.  
✅ Commonly used in **Database-First** projects.

Example: Generate a Controller and Views

dotnet aspnet-codegenerator controller -name MoviesController -m Movie -dc ApplicationDbContext --relativeFolderPath Controllers –useDefaultLayout

This will generate:

* A **MoviesController.cs** file
* Views for **Create, Edit, Delete, Details, and Index**

Auto-Generated Controller:

public class MoviesController : Controller

{

private readonly ApplicationDbContext \_context;

public MoviesController(ApplicationDbContext context)

{

\_context = context;

}

public async Task<IActionResult> Index()

{

return View(await \_context.Movies.ToListAsync());

}

}

* Uses **Dependency Injection** to get the DbContext.
* The Index() action retrieves all **Movies** from the database.

• Entity Framework

**Entity Framework Core (EF Core)** is the **ORM (Object-Relational Mapper)** that connects C# classes with database tables.

**Why Use EF Core?**

✅ Eliminates the need for **raw SQL queries**.  
✅ Works with **multiple databases** (SQL Server, MySQL, PostgreSQL, SQLite, etc.).  
✅ Allows **LINQ queries** instead of SQL.

**How EF Core Works?**

1. Define a **model class** → Becomes a table in the database.
2. Use DbContext → Handles communication between the application and database.
3. Use **LINQ** queries to interact with data.

Example: Querying Data with LINQ

var movies = \_context.Movies.Where(m => m.ReleaseYear > 2000).ToList();

EF Core converts this into SQL:

SELECT \* FROM Movies WHERE ReleaseYear > 2000;

• DbSet

Entity Framework Core (**EF Core**) is an **Object-Relational Mapper (ORM)** used to interact with a database in ASP.NET Core.  
Two fundamental concepts in EF Core are **DbContext** and **DbSet**, which help developers manage database operations using **C# classes instead of raw SQL queries**.

A **DbSet<T>** represents a **table** in the database, where T is a model class.  
It allows you to **query, insert, update, and delete** data.

**Example: Defining a Model and Using DbSet**

public class Movie

{

public int Id { get; set; }

public string Title { get; set; }

public int ReleaseYear { get; set; }

}

Now, we define DbSet<Movie> inside DbContext:

public DbSet<Movie> Movies { get; set; } // This represents the Movies table

This means:

* The **Movies table** in the database is mapped to **DbSet<Movie>**.
* We can **query** and **modify** the Movies table using LINQ.

• DbContext

DbContext is a class in EF Core that acts as a **bridge** between C# objects and the database.  
It is responsible for:

✅ Managing **database connections**  
✅ Tracking **changes** to entities  
✅ Performing **CRUD operations** (Create, Read, Update, Delete)  
✅ Running **migrations**

**Creating a Custom DbContext**

You need to create a class that inherits from DbContext and configure the database connection.

You need to create a class that inherits from DbContext and configure the database connection.

**Example: Define a DbContext Class**

public class ApplicationDbContext : DbContext

{

public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)

: base(options) { }

public DbSet<Movie> Movies { get; set; } // Represents the Movies table

protected override void OnConfiguring(DbContextOptionsBuilder options)

{

options.UseSqlite("Data Source=movies.db"); // Configures SQLite as the database

}

}

ApplicationDbContext **inherits** from DbContext, meaning it has all database-related functionality.

The DbSet<Movie> property represents a table in the database.

The OnConfiguring method defines **which database** to use (e.g., SQLite, SQL Server, PostgreSQL, etc.).

Once you define DbContext and DbSet, you can perform database operations.

**Add a New Record (Create)**

✅ context.Movies.Add(movie) → Adds a new row to the **Movies** table.  
✅ context.SaveChanges() → Saves the change in the database.

Read Data (Retrieve)

✅ context.Movies.ToList() → Retrieves **all** rows from the **Movies** table.  
✅ context.Movies.FirstOrDefault(m => m.Title == "Inception") → Retrieves the **first** movie with the title "Inception".

Update Data

using (var context = new ApplicationDbContext())

{

var movie = context.Movies.FirstOrDefault(m => m.Title == "Inception");

if (movie != null)

{

movie.ReleaseYear = 2012; // Update field

context.SaveChanges(); // Save changes

}

}

✅ Retrieves **Inception** movie.  
✅ Updates **ReleaseYear** to 2012.  
✅ context.SaveChanges() → Saves the update to the database.

Delete Data

✅ context.Movies.Remove(movie) → Deletes the row from **Movies** table.  
✅ context.SaveChanges() → Applies the deletion.

In ASP.NET Core, DbContext is registered in Program.cs for **Dependency Injection (DI)**.

AddDbContext<ApplicationDbContext>() registers **EF Core** with the **Dependency Injection** container.

UseSqlServer() configures **SQL Server** as the database.

A screenshot of a computer program

Description automatically generated

• ModelState.IsValid

ModelState.IsValid is used in ASP.NET Core MVC to **validate user input** when working with forms and models.

**How It Works:**

1. When a form is submitted, ASP.NET Core automatically checks if the input matches the data annotations defined in the model.
2. ModelState.IsValid returns true if **all validation rules are met**, otherwise it returns false.

Example: Checking ModelState.IsValid in a Controller

[HttpPost]

public IActionResult Create(Movie model)

{

if (ModelState.IsValid) // Checks if the model is valid

{

\_context.Movies.Add(model);

\_context.SaveChanges();

return RedirectToAction("Index");

}

return View(model); // If invalid, return the same view with validation errors

}

**When ModelState.IsValid Returns false:**

* If **required fields are missing**.
* If the **data format is incorrect** (e.g., entering text in a numeric field).
* If **custom validation rules** fail.

• Linq

**LINQ (Language Integrated Query)** is a powerful way to query collections and databases in C#.  
It allows developers to retrieve, filter, and manipulate data **using C# syntax instead of SQL**.

**Types of LINQ Queries**

1. **LINQ to Objects** → Querying in-memory collections (List<T>, Array, etc.).
2. **LINQ to Entities** → Querying databases via **Entity Framework Core**.
3. **LINQ to XML** → Querying XML documents.

Example: LINQ Query on a List

✅ Uses Where() to filter movies **after 2010**.

// Find movies released after 2010

var recentMovies = movies.Where(m => m.ReleaseYear > 2010).ToList();

Example: LINQ Query on a Database (DbContext)

using (var context = new ApplicationDbContext())

{

var movies = context.Movies

.Where(m => m.ReleaseYear > 2010)

.OrderBy(m => m.Title)

.ToList();

}

• IQueryable

**IQueryable<T>** is an interface in **Entity Framework Core** that allows **query execution on the database level**, meaning **filters are applied before retrieving data from the database**.

**Why Use IQueryable<T>?**

* **Efficient** → Queries are executed **on the database**, reducing memory usage.
* **Supports Lazy Loading** → Data is fetched **only when needed**.

Example:

public IQueryable<Movie> GetMovies()

{

return \_context.Movies.Where(m => m.ReleaseYear > 2010);

}

The query is NOT executed until it is converted to a list (ToList()).

A screenshot of a computer screen

Description automatically generated

• Git Commands

Git is a version control system used to track changes in source code during software development. Here are some essential Git commands:

1. **git init**: Initializes a new Git repository in the current directory.
2. **git clone <repository\_url>**: Clones an existing remote repository to your local machine.
3. **git status**: Shows the status of changes in the working directory, such as new, modified, or deleted files.
4. **git add <file\_name>**: Stages a file or changes to be committed. Use git add . to stage all changes.
5. **git commit -m "message"**: Commits staged changes with a descriptive message.
6. **git pull**: Fetches changes from a remote repository and merges them into your local branch.
7. **git push**: Pushes local commits to a remote repository.
8. **git branch**: Lists all branches or creates a new branch if followed by a branch name, e.g., git branch new-feature.
9. **git checkout <branch\_name>**: Switches to another branch.
10. **git merge <branch\_name>**: Merges changes from one branch into another.
11. **git log**: Shows the commit history of the current branch.
12. **git reset <commit>**: Undoes changes by resetting the current branch to a previous commit.

• Bootstrap Grid System

Bootstrap is a popular front-end framework that makes web development easier. Its grid system helps create responsive layouts. Here’s how it works, especially in the context of an ASP.NET web app:

1. **Basic Structure**: Bootstrap's grid system is based on a 12-column layout. You can divide a row into up to 12 columns, and the width of the columns can be adjusted based on screen size. You define this using row and col-\* classes.

<div class="row">

<div class="col-md-4">Column 1</div>

<div class="col-md-4">Column 2</div>

<div class="col-md-4">Column 3</div>

</div>

1. **Responsive Design**: You can specify different column sizes for different screen sizes using classes like col-sm-\*, col-md-\*, and col-lg-\*. This ensures that your layout is responsive across mobile, tablet, and desktop screens.

<div class="row">

<div class="col-sm-12 col-md-6">Column 1</div>

<div class="col-sm-12 col-md-6">Column 2</div>

</div>

**col-sm-12**: The column takes up the full width (12 columns) on small screens.

**col-md-6**: The column takes up 6 columns (half the width) on medium and larger screens.

1. **Offsetting Columns**: You can shift columns to the right by using the offset classes. For example, col-md-offset-4 would push the column by 4 grid spaces.

<div class="row">

<div class="col-md-4 col-md-offset-4">Centered Column</div>

</div>

1. **Nesting Columns**: You can also nest columns within columns for more complex layouts. Nesting requires an additional row class.

<div class="row">

<div class="col-md-6">

<div class="row">

<div class="col-md-6">Nested Column 1</div>

<div class="col-md-6">Nested Column 2</div>

</div>

</div>

</div>

In an ASP.NET web app, you can use these Bootstrap grid classes directly in your Razor views (.cshtml files) to structure the layout of your pages. Just ensure you've linked the Bootstrap CSS file in your \_Layout.cshtml file.