FHV
Vorarlberg University
of Applied Sciences

Application Integration and Security

Valmir Bekiri

Philipp Scambor

Web application development with .NET 8

About .NET

- What is .NET
  - **Cross-platform**, **open-source** development framework by Microsoft.
  - Supports **Windows, Linux, macOS.** Originally **.NET Framework** was Windows only. **.NET Core** is the modern, open-source, cross-plattform implementation of .NET
  - Unifies **desktop, web, cloud, mobile, gaming, IoT, AI** applications

Source: https://dotnet.microsoft.com/en-us/

- Key Features
  - **Performance & Scalability** – High-performance runtime & optimizations
  - **Modular & Lightweight** – Flexible deployment with NuGet packages
  - **Interoperability** – Works with multiple languages: **C#**, F#, VB.NET
  - **Unified Development** – One framework for various platforms

https://learn.microsoft.com/en-us/dotnet/core/introduction

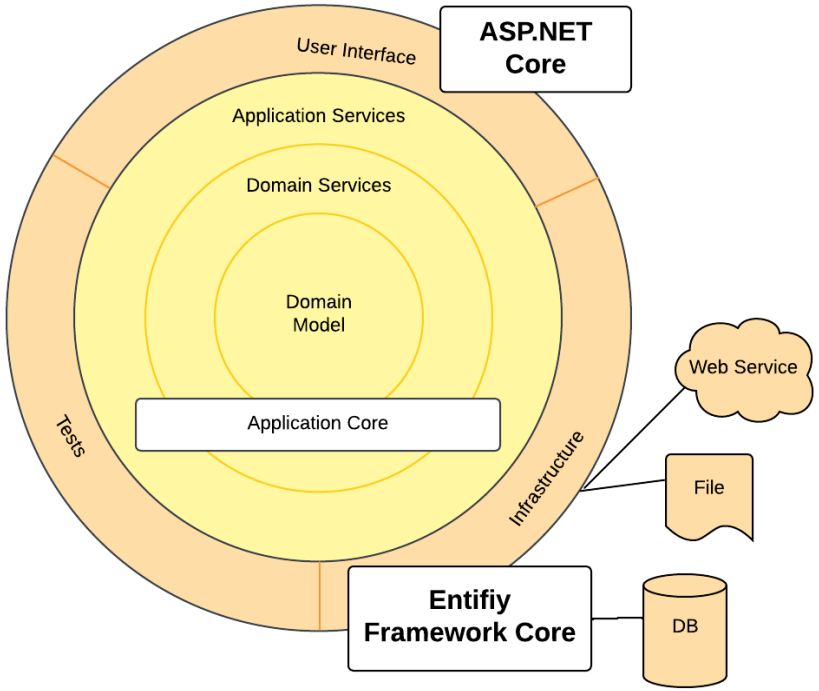
About .NET

- .NET includes the following components:
  - **Runtime** -- executes application code.
  - **Libraries** -- provides utility functionality like JSON parsing.
  - **Compiler** -- compiles C# (and other languages) source code into (runtime) executable code.
  - **SDK** and other tools -- enable building and monitoring apps with modern workflows.
  - **App stacks** -- like **ASP.NET Core**, that enable writing apps.

Source: https://dotnet.microsoft.com/en-us/

What we will focus on

- Basics to build a (stateful) Web API with .NET
  - **ASP.NET Core** => Presentation
  - **Entity Framework Core (EF Core)** => Persistence
- Respecting **Onion Architecture** principles
- Understanding some key **application architecture concepts and patterns** that are commonly applied while developing an application (Dependency Injection, IoC, Repository pattern, etc.)
  - We will introduce those concepts at time (annotated with "*Application architecture*")



Onion Architecture visualisation with ASP.NET and EF Core in context
Source: https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/

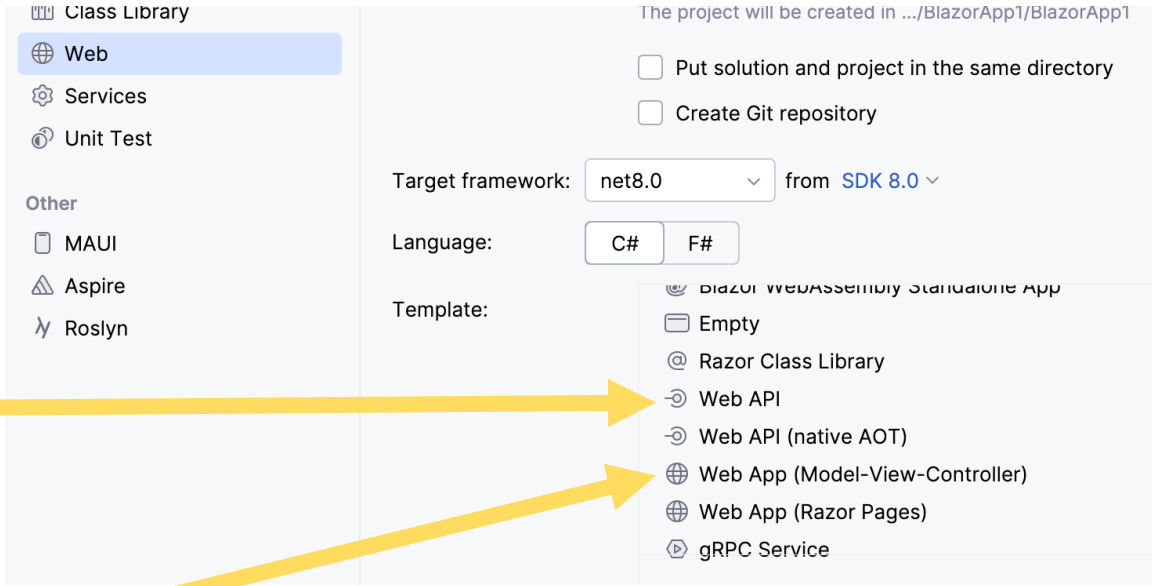
ASP.NET Core Web-API

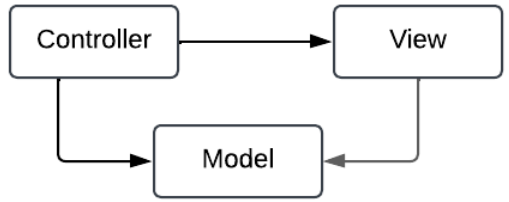
ASP.NET Core

Ultimate learning path

https://roadmap.sh/aspnet-core

- ASP.NET Core supports different approaches
  - client side rendering (CSR) or web service
    = ASP.NET Core-Web-**API**
  - Model View Controller (MVC) with server side rendering (SSR)
    = ASP.NET Core-Web-**App (MVC)**



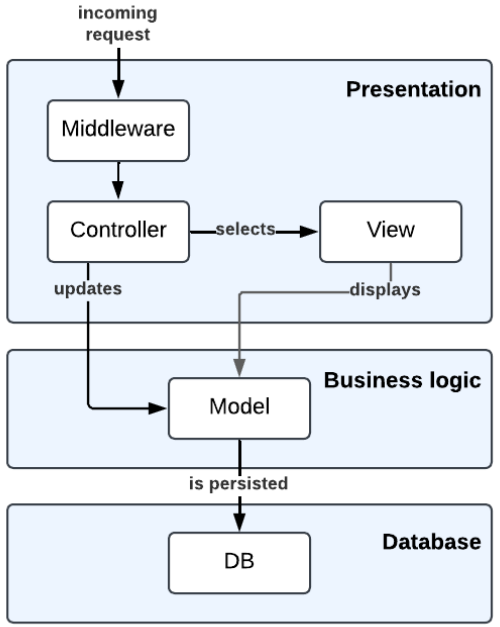


MVC dependency graph

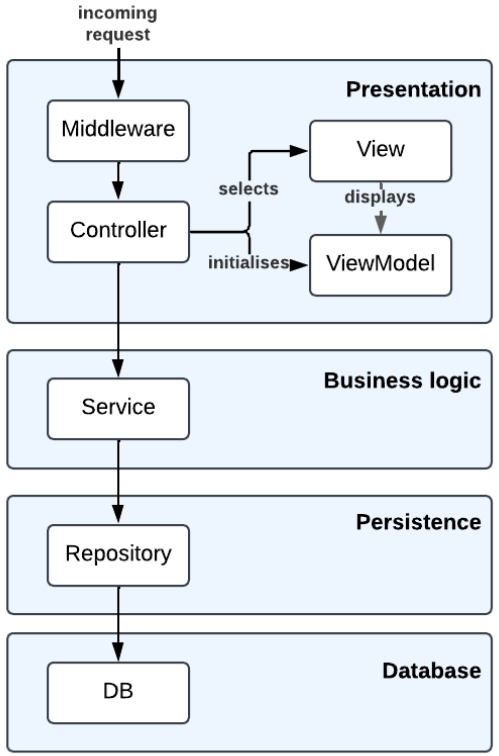
Source: https://learn.microsoft.com/en-gb/aspnet/core/mvc/overview

ASP.NET MVC (SSR)

- Separates application logic (**Model**), UI (**View**), and user interaction (**Controller**).
  - **Routing**: Maps URLs to controller actions.
  - **Models & Data Binding**: Uses model classes and data annotations for validation.
  - **Controllers**: Handle user input, interact with models, and return views or responses.
  - **Views**: SSR templates that display models
- **Middleware & Filters**: Middleware processes requests, filters add cross-cutting behaviour (e.g., authorization, caching).



MVC layered perspective as of definition



MVC layered perspective more recent approach (only presentation layer)

https://learn.microsoft.com/en-gb/aspnet/core/mvc/overview

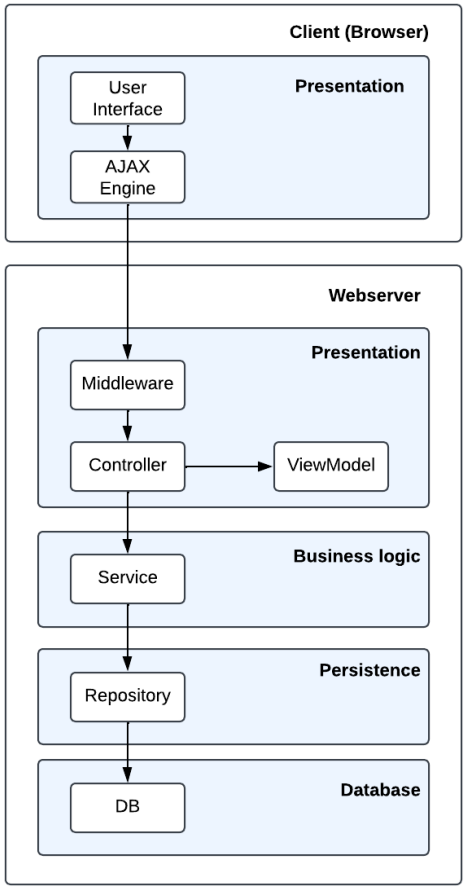
ASP.NET Core Web API (CSR)

Classic CSR

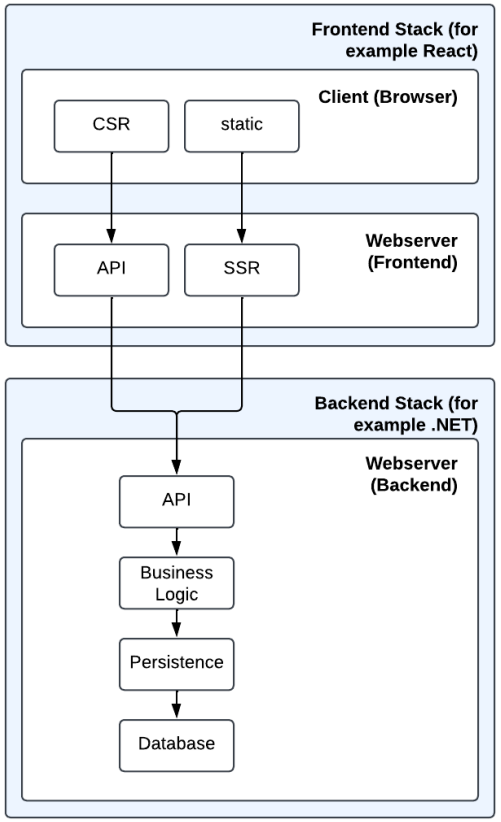
- Client (Browser) calls Web Service asynchrony (AJAX)
- Server is processing the request and sends data back to the client (JSON, XML, …)
  - Routing engine (Middleware) calls referred controller
  - Controller validates data, prepares data, generates response
- Client renders response data and updates UI

Modern hybrid approach

- Using a modern frontend framework that combines SSR and CSR (like Next.js or similar)
- Backend does only provide an API that is consumed by the frontend (directoly from client or from frontend web server



Classic CSR



Modern hybrid approach with CSR and SSR in the frontend stack and separated backend stack

Create an ASP.NET Core Web API

/<project-name>.csproj

- In certain IDEs hidden (file not visible)
- Stores project configuration mostly relevant for the build (.NET framework version, dependencies, etc.)

/appsettings.json

- Stores application configuration (database connection, logging, API keys, etc.)
- Affects runtime behaviour

/Properties/launchSettings.json

- Defines URL where the application can be reached
- Optional launch URL (startup path)
- **ONLY for development**

/Program.cs

- Entry point of the program
- Initialisation and configuration for .NET

```
/MyDotNetApp
├── MyDotNetApp.csproj      # Project configuration
├── appsettings.json        # Application settings
├── appsettings.Development.json  # Environment-specific settings
├── Properties
│   ├── launchSettings.json # Defines how the app runs locally
├── Program.cs              # Entry point of the application
└── bin/                    # Compiled output (DLLs, EXEs)
└── obj/                    # Temporary build files
```

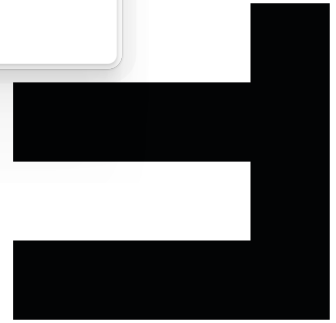
Create an ASP.NET Core Web API

`/Program.cs`

- Uses builder pattern to configure the application
- Configures and registers Services (available across the application via **Dependency Injection**)
- Can be used to directly map routes or to initialise controller based routing (will be explained in the next slides)
- Swagger API documentation is automatically enabled if you start from an ASP.NET Core Web API template

```
1  var builder = WebApplication.CreateBuilder(args);
2
3  // Add services to the container.
4  // Learn more about configuring Swagger/OpenAPI at
   https://aka.ms/aspnetcore/swashbuckle
5  builder.Services.AddEndpointsApiExplorer();
6  builder.Services.AddSwaggerGen();
7
8  var app = builder.Build();
9
10 // Configure the HTTP request pipeline.
11 if (app.Environment.IsDevelopment())
12 {
13     app.UseSwagger();
14     app.UseSwaggerUI();
15 }
16
17 app.UseHttpsRedirection();
18 app.Run();
19
```

Default Program.cs after creating a new ASP.NET Core Web API Project with a template (Rider IDE)



Application architecture
Dependency Injection

Problem

- Class A depends on Class B
- Dependencies have a negative impact on the maintainability of the code

```
1  public class ClassA
2  {
3    private ClassB _obj;
4
5    public ClassA()
6    {
7      _obj = new ClassB();
8      _obj.Operation();
9    }
10 }
11
12 public class ClassB
13 {
14   public void Operation()
15   { /* [...] */ }
16 }
```

DI Solution:

- Class A and Class B are decoupled
- Class A depends on an Interface rather than directly on class B
- Class B can be injected into class A as implementation of the interface
- Injection is done by the IoC container

```
1  public class ClassA
2  {
3    private IOperation _obj;
4
5    public ClassA(IOperation obj)
6    {
7      _obj = obj;
8      _obj.Operation();
9    }
10 }
11
12 public class ClassB : IOperation
13 {
14   public void Operation()
15   { /* [...] */ }
16 }
17
18 public interface IOperation
19 {
20   void Operation();
21 }
```

Application architecture

Inversion of Control & Dependency Injection

- **Inversion of Control (IoC)**
  - Shifts object creation & lifecycle management from the app to a container
  - Promotes **loose coupling** and **better maintainability**
- **Dependency Injection (DI)**
  - A technique to **inject dependencies instead of creating them manually**
  - .NET built-in DI container manages dependencies automatically
  - Three DI lifetimes:
    - **Transient** (new instance per request)
    - **Scoped** (instance per scope, e.g., per HTTP request)
    - **Singleton** (one instance for app lifetime)

Application architecture

.NET IoC & DI example

```cs
public class LectureController
{
  private ILectureService _service;

  public LectureController(ILectureService service)
  {
    _service = service;
  }

  public ActionResult<Lecture> Create(CreateLectureRequest request)
  {
    var lecture = _service.Create(request.Title);

    return Created(lecture);
  }
}
```

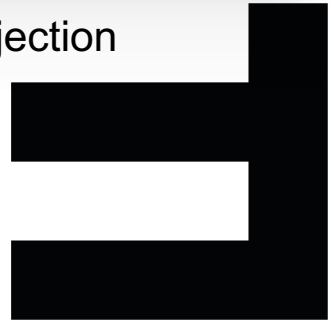
Loose coupling between `LectureController` and `LectureService` through `ILectureService` interface

```cs
public interface ILectureService
{
    public Lecture Create();
}
```

```cs
public class LectureService : ILectureService
{
    public Lecture Create()
    { /* [...] */ }
}
```

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

builder.Services.AddScoped<ILectureService, LectureService>();

...
var app = builder.Build();
app.run();
```

.NET IoC container handles creation and injection of `LectureService` where ever `ILectureService` is used

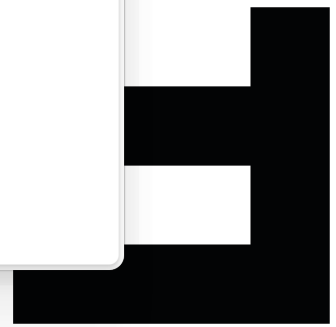


ASP.NET Controllers

- **Controllers inherit from ControllerBase** and their name usually ends with “Controller“ (for standard routing).
  - Public methods in the controller are called **action methods** which can be accessed via **conventions**
  - Action methods process the incoming requests and **return a result**
- As dependency injection is a built-in part of the framework along with configuration, logging can be used, simply by adding the (typed) ILogger as a parameter in the constructor

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

// adds Controller classes to IoC container
builder.Services.AddControllers();
var app = builder.Build();

app.UseHttpsRedirection();

// maps request routes to Controllers
app.MapControllers();
app.Run();
```

```
[ApiController]
[Route("api/[controller]")]
public class UsersController : ControllerBase {
    private readonly ILogger<UsersController> _logger;

    public UsersController(ILogger<UsersController> logger) {
        _logger = logger;
    }

    public IActionResult Get() {
        _logger.LogInformation("Fetching users...");
        return Ok("Default GET");
    }

    public IActionResult Create() {
        _logger.LogInformation("Creating a user...");
        return Created("", "Default POST");
    }
}
```



ASP.NET ControllerBase

- The `ControllerBase` class provides many properties and methods that are useful for handling HTTP requests. For example, CreatedAtAction returns a 201 status code
  - Helper methods for all HTTP response codes (`Ok()`, `NotFound()`, ...)
  - Access to request (to access request headers, cookies, etc.), validation and much more
- For all available properties and methods see: https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.controllerbase

```
1  [ApiController]
2  [Route("api/users")] // Base route for the controller
3  public class UsersController : ControllerBase
4  {
5    [HttpGet("{id}/something")] // Maps to GET api/users/{id}/something
6    [ProducesResponseType(typeof(Something), 200)] // for OpenAPI docs
7    [ProducesResponseType(404)] // for OpenAPI docs
8    public ActionResult<Something> GetSomething(
9        int id, [FromQuery] string color, [FromQuery] int count
10             // GET api/users/{id}/something?color=red&count=4
11   ) {
12     var user = userservice.GetById(id);
13     if (user == null)
14     {
15       return NotFound();
16     }
17
18     var something = userservice.BuildSomething(user, color, count);
19     return Ok(something);
20   }
21 }
```

ASP.NET OpenAPI & Swagger

- The **OpenAPI** specification is a programming language-agnostic standard for **documenting HTTP APIs**
- **Swagger is tooling** that uses the OpenAPI specification. For example SwaggerUI
- ASP.Net Core Web Api Starter template in Visual Studio and Rider already have Swagger tooling installed and configured
  - /swagger route can be accessed to browse the api documentation
  - Swashbuckle.AspNetCore package is already installed
- OpenAPI documentation can largely be automatically **derived from existing conventions and annotations**. However, there are some **annotations like [ProducesResponseType]** that are added **explicitly for documentation** purposes.

```
[ApiController]
[Route("api/users")] // Base route for the controller
public class UsersController : ControllerBase
{
  [HttpGet("{id}/something")] // Maps to GET api/users/{id}/som
  [ProducesResponseType(typeof(Something), 200)] // for OpenAPI
  [ProducesResponseType(404)] // for OpenAPI docs
  public ActionResult<Something> GetSomething(
      int id, [FromQuery] string color, [FromQuery] int count
```

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

builder.Services.AddControllers();
builder.Services.AddSwaggerGen();

var app = builder.Build();

// enable swagger docs while in development
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

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

ASP.NET ApiController and Route

The `[ApiController]` attribute is applied to a controller class to enable the following opinionated, API-specific behaviours:

- **Attribute routing requirement**
  - `[Route(...)]` has to be specified. A route can be statically defined as "`api/users`" or reference the controller name "`api/[controller]`"
- **Automatic HTTP 400 responses**
  - If action parameters validation fails an automatic HTTP 400 Bad Request is returned
- **Binding source parameter inference**
  - Annotations to define the location at which an action parameter's value is found: `[FromQuery]`, `[FromBody]`, `[FromRoute]`, etc.
- **Problem details for error status codes**
  - Methods like `NotFound()` resolve to a result with `ProblemDetails` (Json body of the response). The `ProblemDetails` type is based on the RFC 7807 specification for providing machine-readable error details in an HTTP response.

```
[ApiController]
[Route("api/users")] // Base route for the controller
public class UsersController : ControllerBase
{
  [HttpGet("{id}/something")] // Maps to GET api/users/{id}/something
  [ProducesResponseType(typeof(Something), 200)] // for OpenAPI docs
  [ProducesResponseType(404)] // for OpenAPI docs
  public ActionResult<Something> GetSomething(
      int id, [FromQuery] string color, [FromQuery] int count
          // GET api/users/{id}/something?color=red&count=4
  ) {
    var user = userservice.GetById(id);
    if (user == null)
    {
      return NotFound();
    }

    var something = userservice.BuildSomething(user, color, count);
    return Ok(something);
  }
}
```

```
{
  type: "https://tools.ietf.org/html/rfc7231#section-6.5.4",
  title: "Not Found",
  status: 404,
  traceId: "0HLHLV31KRN83:00000001"
}
```

ASP.NET Controllers: Methods & routes

Method Name Convention	Allowed HTTP Methods
`Get(), GetAll(), GetById()`	`GET`
`Post(), Create()`	`POST`
`Put(), Update()`	`PUT`
`Patch(), Modify()`	`PATCH`
`Delete(), Remove()`	`DELETE`

Controller action method conventions (usually work out of the box) -- No need to annotate [HttpGet] if the method is named Get()

```
1  [ApiController]
2  [Route("api/users")] // Base route for the controller
3  public class UsersController : ControllerBase {
4
5      [HttpGet] // Explicitly maps to GET
6      public IActionResult FetchAll() => Ok("GET all users");
7
8      [HttpGet("{id}")] // Custom route for getting a user by ID
9      public IActionResult FetchById(int id) => Ok($"GET user {id}");
10
11     [HttpPost("register")] // Custom POST route
12     public IActionResult RegisterUser() => Created("", "User registered");
13
14     [HttpPut("{id}")] // Overrides naming convention to allow PUT with an ID
15     public IActionResult UpdateUser(int id) => Ok($"User {id} updated");
16
17     [HttpDelete("remove/{id}")] // Custom DELETE route
18     public IActionResult RemoveUser(int id) => Ok($"User {id} deleted");
19 }
20
```

Custom routes & explicit HTTP Methods

ASP.NET Controllers: Action Results

- ASP.NET Core provides the following options for web API controller action return types:
  - any Specific type: Most basic; can be used if only one return type is possible (200 Ok)
  - IActionResult: appropriate when multiple ActionResult return types are possible (200 Ok, 400 Bad Request, 404 Not Found)
  - ActionResult<T>: enables returning a type deriving from ActionResult or return a specific type.
  - HttpResults: in contrast to IActionResult and ActionResult<T> this implementation does not leverage the configured formatters and similar. It just does less out of the box (which is favorable in some cases).

```
1  // specific return type
2  [HttpGet("user-count")
3  public int GetUserCount() { ... }
4
5  // IActionResult
6  [ProducesResponseType(200, Type = typeof(User))
7  [ProducesResponseType(404)]
8  public IActionResult GetUserById() { ... }
9
10 // ActionResult<T>
11 [ProducesResponseType(200)] // type is auto derived
12 [ProducesResponseType(404)]
13 public ActionResult<User> GetUserById() { ... }
```

ASP.NET Controllers and Tasks

`ActionResult<T>`

- **Represents HTTP responses** (e.g., Ok(), NotFound(), BadRequest()).
- Can return **both data and status codes** (ActionResult<User>).

```
1  public ActionResult<User> GetUser() => Ok(new User());
```

`Task<ActionResult<T>>`

- Used for **asynchronous** actions (async/await).
- Improves scalability for I/O-bound operations (e.g., DB calls).
- Returns a Task wrapper for async execution.

```
1  public async Task<ActionResult<User>> GetUserAsync() {
2      var user = await _service.GetUserAsync();
3      return user is not null ? Ok(user) : NotFound();
4  }
5
```

Application Architecture

Repository Pattern

- **Encapsulates data access logic:** abstracts database operations from business logic.
- **Provides a consistent interface** (IRepository<T>) for querying and modifying data.
- **Enhances testability:** allows mocking of data sources for unit testing.
- **Decouples business logic from ORM (e.g., Entity Framework)** for flexibility (see later)
- **Supports multiple data sources** (SQL, NoSQL, APIs) without changing service logic.

```
1  public interface IUserRepository {
2      Task<User?> GetByIdAsync(int id);
3      Task<User?> FindByEmailAsync(string email);
4  }
```

```
1  [ApiController]
2  [Route("api/users")]
3  public class UsersController : ControllerBase {
4      private readonly IUserRepository _userRepository;
5
6      public UsersController(IUserRepository userRepository) {
7          _userRepository = userRepository;
8      }
9
10     [HttpGet("{id}")]
11     public async Task<ActionResult<User>> GetUser(int id) {
12         var user = await _userRepository.GetByIdAsync(id);
13         return user is not null ? Ok(user) : NotFound();
14     }
15 }
```

That‘s it ☺
…for now