Ping authen

Service

Block diagramme

Design principals

Serilog

C#

Number array -odd number square

3 consecutive odd even true

Capitalized first element from string

Can we write try with finally only-yes

If exception happens in catch will it execute finally – yes

If Exception in catch it will always go to Exception ex

Can try has multiple catch-yes

Collections

Generics

Generic and nongeneric collection

Delegates

Async await

Class template to make it immutable

Shallow copy , Deep copy in c#

Sealed static readonly extension methods , linq ,entity framework , sql- indexes nonclusterd index how indexes internally work,task vs thread,semaphore

Oops

Dynamic garbage collection multithreading task parallelism

Middleware dependency injection hosting environment lifetime of services,configurations -app settings ,hosting environment how you read configuration development environment

Dotnetcore execution lifecycle

Web servers inproc outproc kestrel server,custom middleware what is process

Web Api

API gateway,Rest Api SOAP API,CLIENT SERVER Architecture

Logging in WEB API(Generic logger,Custom logger),HttpMethods jsonpath patch different,Custom Model Binding,custom validation,return types web apis,2 series-200,4 series,Security of Web APIS

Idempotency in Web API,Ratelimiting factor in Web API,Routing in WEB API,API versioning

DEVEOPS PIPELINE RELEASE PIPELINE,CODE COVERAGE,CODE COMPLEXITY

How you will make class as immutable

Making a class immutable in C# ensures that its instances cannot be modified after they are created. Here’s how you can achieve this:

**Steps to Create an Immutable Class:**

1. **Use readonly fields** – Declare fields as private readonly so they can only be set during object creation.
2. **Initialize fields via a constructor** – Assign values to fields through a constructor, preventing modification after initialization.
3. **Avoid setters** – Properties should be read-only (get only) and not have set accessors.
4. **Prevent modifications from outside** – If your class has collections, expose them as read-only (ReadOnlyCollection<T> or return copies).
5. **Ensure deep immutability** – Any reference-type fields should also be immutable or return defensive copies.

**Example:**

public class ImmutablePerson

{

public string Name { get; }

public int Age { get; }

public ImmutablePerson(string name, int age)

{

Name = name;

Age = age;

}

}

**Handling Collections:**

If your class contains collections, return a read-only version:

public class ImmutableTeam

{

private readonly List<string> members;

public ImmutableTeam(IEnumerable<string> members)

{

this.members = new List<string>(members); // Defensive copy

}

public IReadOnlyList<string> Members => members.AsReadOnly();

}

**Benefits of Immutability:**

* **Thread Safety** – No unexpected modifications in a multi-threaded environment.
* **Predictability** – Easier debugging and reasoning.
* **Reliability** – Helps avoid unintended side effects.

Would you like more examples or explanations?

**Access Modifiers in C#**

Access modifiers in C# define the **visibility** and **accessibility** of classes and their members. They help enforce **encapsulation** by restricting access to certain parts of the code.

**Types of Access Modifiers**

| **Modifier** | **Accessibility** |
| --- | --- |
| **Public** | Accessible from anywhere in the program. |
| **Private** | Accessible only within the same class. |
| **Protected** | Accessible within the same class and derived classes. |
| **Internal** | Accessible within the same assembly. |
| **protected internal** | Accessible within the same assembly and derived classes in other assemblies. |
| **private protected** | Accessible within the same class and derived classes in the same assembly. |
| **File** | Accessible only within the same file (introduced in C# 11). |

**SOAP VS REST**

**REST vs SOAP in .NET**

**In .NET, both REST (Representational State Transfer) and SOAP (Simple Object Access Protocol) are used for web services, but they differ in architecture, communication style, and implementation.**

**1. REST (Representational State Transfer)**

**REST is an architectural style that uses standard HTTP methods (GET, POST, PUT, DELETE) to interact with resources. It is lightweight, stateless, and widely used for web APIs.**

**Key Features**

**✅ Uses HTTP methods (GET, POST, PUT, DELETE)  
✅ Supports multiple formats (JSON, XML, etc.)  
✅ Stateless – No session is maintained between requests  
✅ Scalable – Works well for distributed systems  
✅ Faster – Less overhead compared to SOAP**

**Example: REST API in .NET**

**[ApiController]**

**[Route("api/products")]**

**public class ProductsController : ControllerBase**

**{**

**[HttpGet]**

**public IActionResult GetProducts()**

**{**

**var products = new List<string> { "Laptop", "Phone", "Tablet" };**

**return Ok(products); // Returns JSON response**

**}**

**}**

**📌 REST APIs in .NET are built using ASP.NET Core Web API.**

**2. SOAP (Simple Object Access Protocol)**

**SOAP is a protocol that defines strict rules for communication using XML. It is used in enterprise applications where security and reliability are critical.**

**Key Features**

**✅ Uses XML format for structured communication  
✅ Supports WS-Security for secure transactions  
✅ Maintains state – Can store session information  
✅ Works over multiple protocols (HTTP, SMTP, TCP)  
✅ Standardized – Uses WSDL (Web Services Description Language)**

**Example: SOAP Service in .NET**

**[ServiceContract]**

**public interface IProductService**

**{**

**[OperationContract]**

**List<string> GetProducts();**

**}**

**public class ProductService : IProductService**

**{**

**public List<string> GetProducts()**

**{**

**return new List<string> { "Laptop", "Phone", "Tablet" };**

**}**

**}**

**📌 SOAP services in .NET are built using WCF (Windows Communication Foundation).**

**3. Key Differences**

| **Feature** | **REST** | **SOAP** |
| --- | --- | --- |
| **Architecture** | **Stateless** | **Stateful** |
| **Data Format** | **JSON, XML** | **XML only** |
| **Protocol** | **HTTP** | **HTTP, SMTP, TCP** |
| **Security** | **OAuth, JWT** | **WS-Security** |
| **Performance** | **Faster** | **Slower (XML overhead)** |
| **Best For** | **Web APIs, mobile apps** | **Enterprise applications, banking** |

**4. When to Use REST vs SOAP**

* **Use REST for web APIs, mobile apps, and microservices.**
* **Use SOAP for banking, financial transactions, and enterprise security.**

**Client Server Architecture**

**Client-Server Architecture in Web API**

Client-server architecture is a fundamental design pattern in web applications where a client requests services, and a server processes and responds to those requests. This model ensures scalability, security, and efficient resource management.

1. Key Components

* Client – The front-end application or user interface that sends requests.
* Server – The back-end system that processes requests and returns responses.
* Communication Protocol – Typically HTTP/HTTPS for web APIs.
* Database – Stores and retrieves data based on client requests.

2. How It Works

1. The client sends an HTTP request (e.g., GET, POST, PUT, DELETE).
2. The server processes the request and interacts with the database if needed.
3. The server sends a response (e.g., JSON or XML) back to the client.

3. Types of Client-Server Architectures

| Architecture | Description |
| --- | --- |
| 1-Tier | Client and server are combined (e.g., desktop applications). |
| 2-Tier | Client communicates directly with the database server. |
| 3-Tier | Client, application server, and database server are separate. |
| N-Tier | Multiple layers for better scalability and security. |

4. Example: Web API in .NET Core

[ApiController]

[Route("api/products")]

public class ProductsController : ControllerBase

{

[HttpGet]

public IActionResult GetProducts()

{

var products = new List<string> { "Laptop", "Phone", "Tablet" };

return Ok(products); // Returns JSON response

}

}

📌 ASP.NET Core Web API follows the client-server model, where the client (browser or mobile app) interacts with the API.

5. Advantages of Client-Server Architecture

✅ Scalability – Can handle multiple clients efficiently.  
✅ Security – Centralized control over data access.  
✅ Performance – Optimized request handling and caching.  
✅ Flexibility – Supports different clients (web, mobile, IoT).

SAML VS OAUTH for security

OAuth and SAML are both widely used authentication and authorization protocols, but they serve different purposes in .NET applications.

**OAuth**

* **Purpose**: Primarily used for authorization.
* **How it works**: OAuth allows applications to access resources on behalf of a user without exposing credentials.
* **Common Use Cases**: API security, third-party integrations (e.g., logging in with Google or Microsoft).
* **Implementation in .NET**: ASP.NET Core supports OAuth 2.0 via IdentityServer or Microsoft Entra ID.

**SAML**

* **Purpose**: Primarily used for authentication.
* **How it works**: SAML enables Single Sign-On (SSO) by exchanging authentication tokens between an identity provider and a service provider.
* **Common Use Cases**: Enterprise authentication, federated identity management.
* **Implementation in .NET**: Can be integrated using libraries like Sustainsys.Saml2 for ASP.NET applications.

**Key Differences**

| **Feature** | **OAuth** | **SAML** |
| --- | --- | --- |
| **Primary Function** | Authorization | Authentication |
| **Token Type** | Access Token (JWT) | XML-based Assertions |
| **Use Case** | API access, mobile apps | SSO for enterprise apps |
| **Common Providers** | Microsoft Entra ID, Google, Facebook | Okta, ADFS, Ping Identity |

If you're working on securing APIs in .NET, OAuth is generally the better choice. If you're implementing SSO for enterprise applications, SAML is more suitable. You can find more details [here](https://www.c-sharpcorner.com/article/understanding-concepts-openid-oauth-and-saml/) and [here](https://www.okta.com/identity-101/saml-vs-oauth/). Let me know if you want a deeper dive into implementation!

Secure your Web APIS

Securing web APIs is crucial to protect sensitive data and prevent unauthorized access. Here are some best practices:

1. **Use Authentication & Authorization** – Implement strong authentication mechanisms like OAuth, JWT, or API keys to ensure only authorized users can access your API.
2. **Encrypt Data Transmission** – Always use HTTPS to encrypt data in transit and prevent interception.
3. **Validate Input & Output** – Prevent injection attacks by validating and sanitizing user inputs.
4. **Rate Limiting & Throttling** – Limit the number of requests to prevent abuse and denial-of-service (DoS) attacks.
5. **Implement CORS Policies** – Configure Cross-Origin Resource Sharing (CORS) to control which domains can access your API.
6. **Monitor & Log API Activity** – Keep track of API usage and detect anomalies or suspicious activities.
7. **Use Secure API Design Principles** – Follow security principles like least privilege, fail-safe defaults, and complete mediation.

Would you like guidance on implementing security in a specific framework or language?

How you read environment specific configuration

In .NET Core, you can read configuration based on the **development environment** using environment variables and configuration files like appsettings.Development.json. Here’s how:

**1. Using Environment Variables**

.NET Core uses the ASPNETCORE\_ENVIRONMENT variable to determine the environment (Development, Staging, Production).

**Example:**

var environment = Environment.GetEnvironmentVariable("ASPNETCORE\_ENVIRONMENT");

Console.WriteLine($"Current Environment: {environment}");

**2. Using IWebHostEnvironment**

You can inject IWebHostEnvironment to check the environment dynamically.

**Example:**

public class HomeController : Controller

{

private readonly IWebHostEnvironment \_env;

public HomeController(IWebHostEnvironment env)

{

\_env = env;

}

public IActionResult Index()

{

string envName = \_env.EnvironmentName;

return Content($"Current Environment: {envName}");

}

}

**3. Using appsettings.{Environment}.json**

You can create environment-specific configuration files like:

* appsettings.Development.json
* appsettings.Staging.json
* appsettings.Production.json

**Example appsettings.Development.json:**

{

"Logging": {

"LogLevel": {

"Default": "Debug"

}

}

}

**4. Reading Configuration in Program.cs**

var builder = WebApplication.CreateBuilder(args);

var config = builder.Configuration;

string logLevel = config["Logging:LogLevel:Default"];

Console.WriteLine($"Log Level: {logLevel}");

For more details, check out [Microsoft Learn](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/configuration/?view=aspnetcore-9.0) and [Stack Overflow discussions](https://stackoverflow.com/questions/32548948/how-to-get-the-development-staging-production-hosting-environment-in-configurese). Would you like an example with a different configuration source?