Mastering .NET Core API Course

By RachelYavetz

Exploring API in .NET Core

Understanding API Development

Explore about api and http protocol and various of api

Implementing API Endpoints

Dive into building various API endpoints and functionalities
Web config, Swagger

Introduction to Git

Learn the basics of Git concepts

Introduction to .NET 8

Learn Layers and Dependency Injection

Diffrenet Approch to Connect to DB

Ado & Entity framework

Logs & Tests

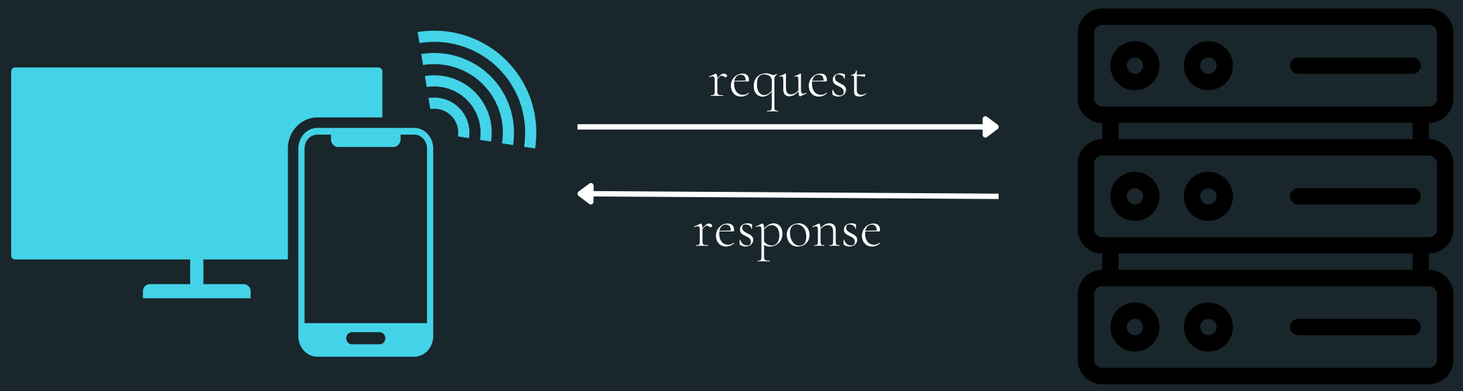
learn how to write good logs and tests

GraphQL

Minimal Api

What is An API

An Application Programming Interface, commonly shortened as API,
is a set of rules which determines how one software program can access the data or functionality provided by another software program.
APIs are an essential part of modern software development.
They allow different systems and applications to communicate with each other and share functionality in a flexible and efficient way.
They're used in a wide variety of contexts, including web development, mobile apps, and Internet of Things (IoT) applications.

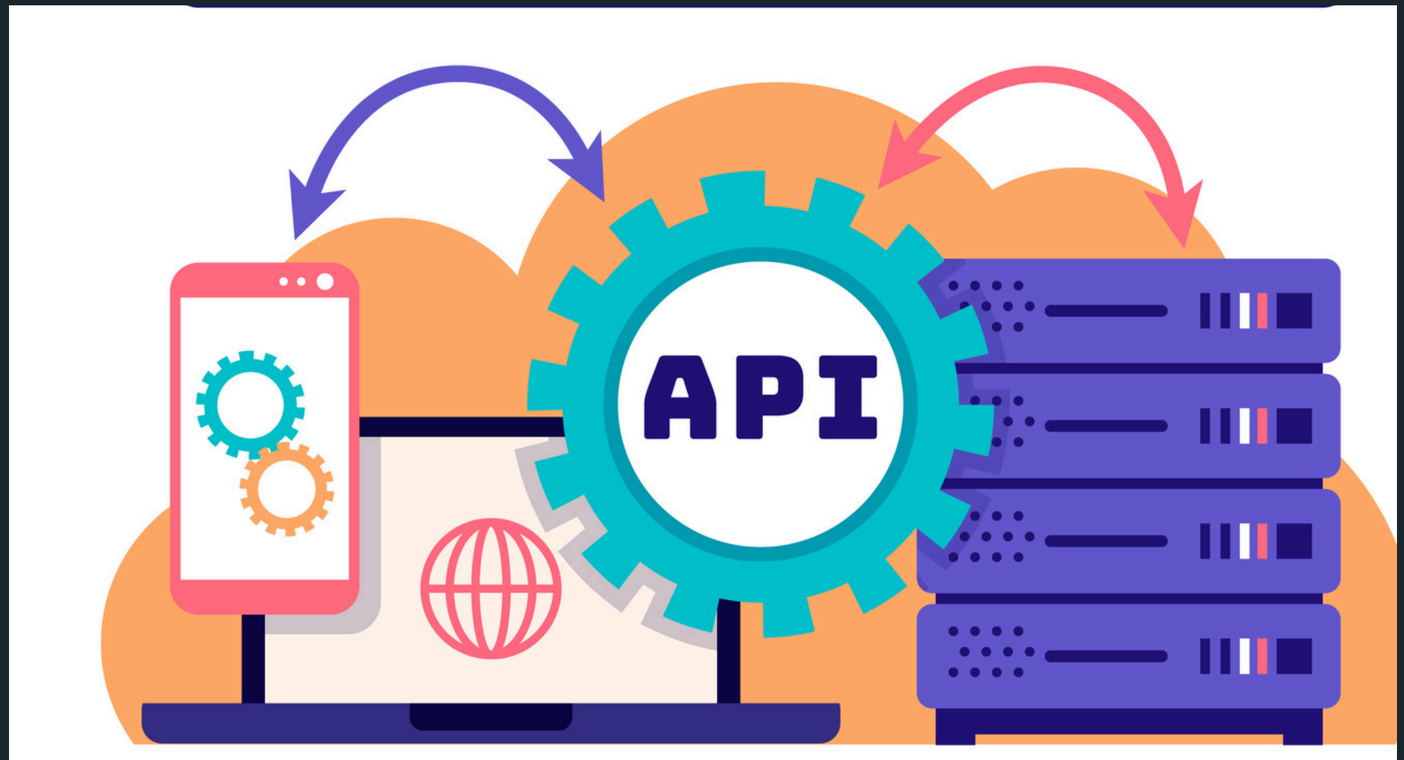


Api Types

Types of API architectures

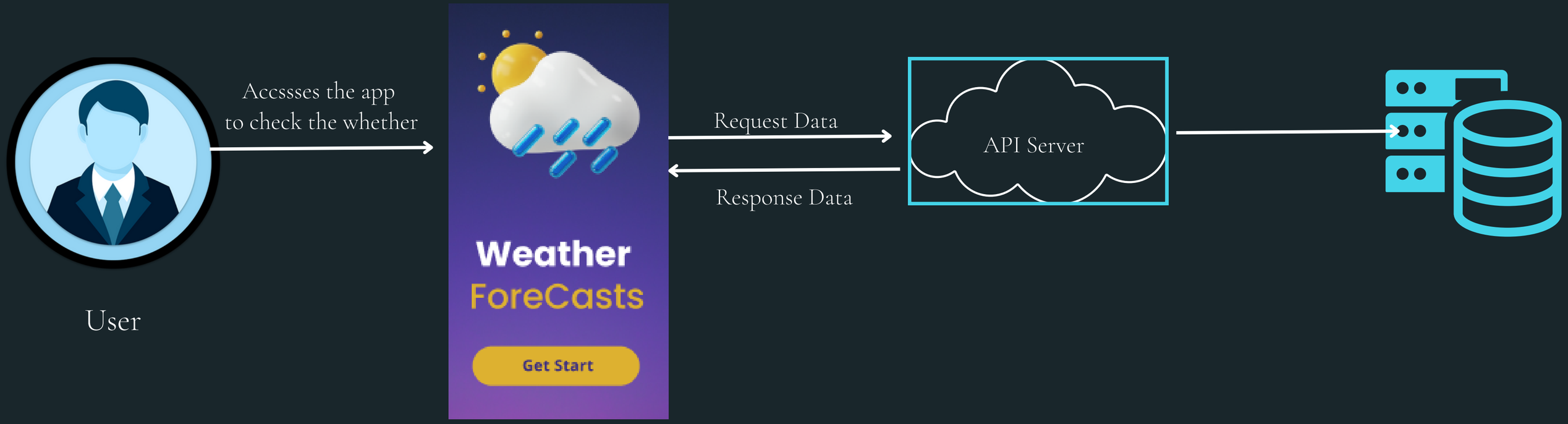
1.Representational State Transfer (REST) APIs
2. Soap Api
3. Rpc Api
4. GraphQL

The Focus is Rest API



Rest Api

Defined Routes With Url - Simple Is it sounds....



Rest Api Benefits

Simplicity- REST APIs use common HTTP methods, including GET, PUT, POST and DELETE requests, making them easy to design, implement and use.

Independence. Developers enjoy platform independence because they can use almost any programming language to create REST APIs. They work with various client devices, such as traditional web browsers, mobile devices and internet-of-things devices.

Flexible. REST APIs support many different data formats, including JSON, XML and plain text. Developers can choose the data format that best suits client needs and available server-side data.

Scalable. The stateless nature of REST APIs supports horizontal scaling, where many API calls run in parallel to handle significant API call loads.

Cacheable. REST APIs support caching, allowing data to be stored in local memory. This approach can speed server-side response time, potentially improving API performance. It might even eliminate the need for an API call if required data is already on the client from a prior call.

Secure. REST APIs can secure calls and data exchanges with Open Authorization (OAuth) authentication and Secure Sockets Layer/Transport Layer Security encryption.

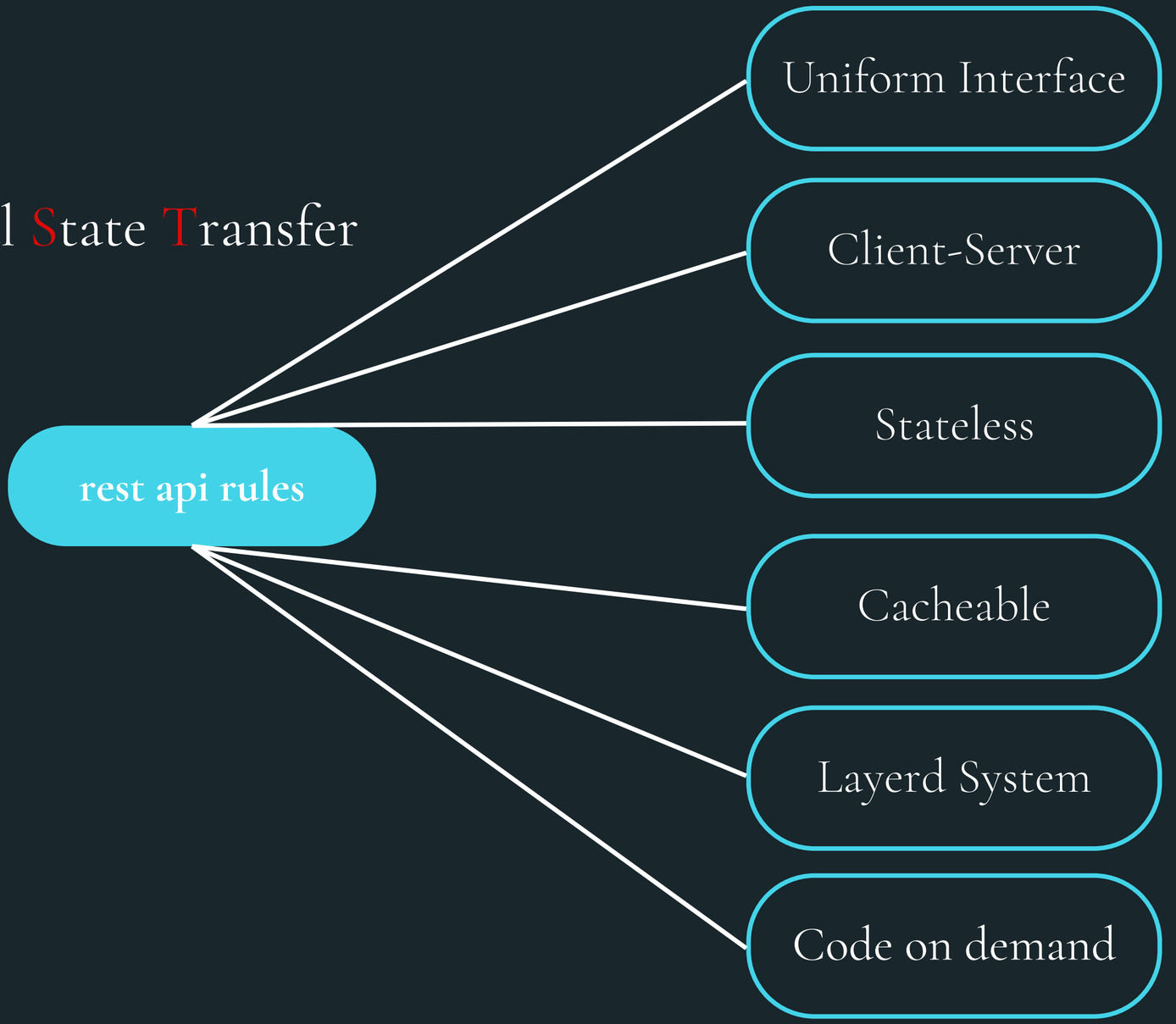
Compatible. Proper use of versioning lets developers treat APIs as any other evolving software, adding new features over time with backward compatibility and support of legacy features for existing clients.

Rest Api

Rest stands for Representational State Transfer

rest is not a spescifcation

rest is a set of rules

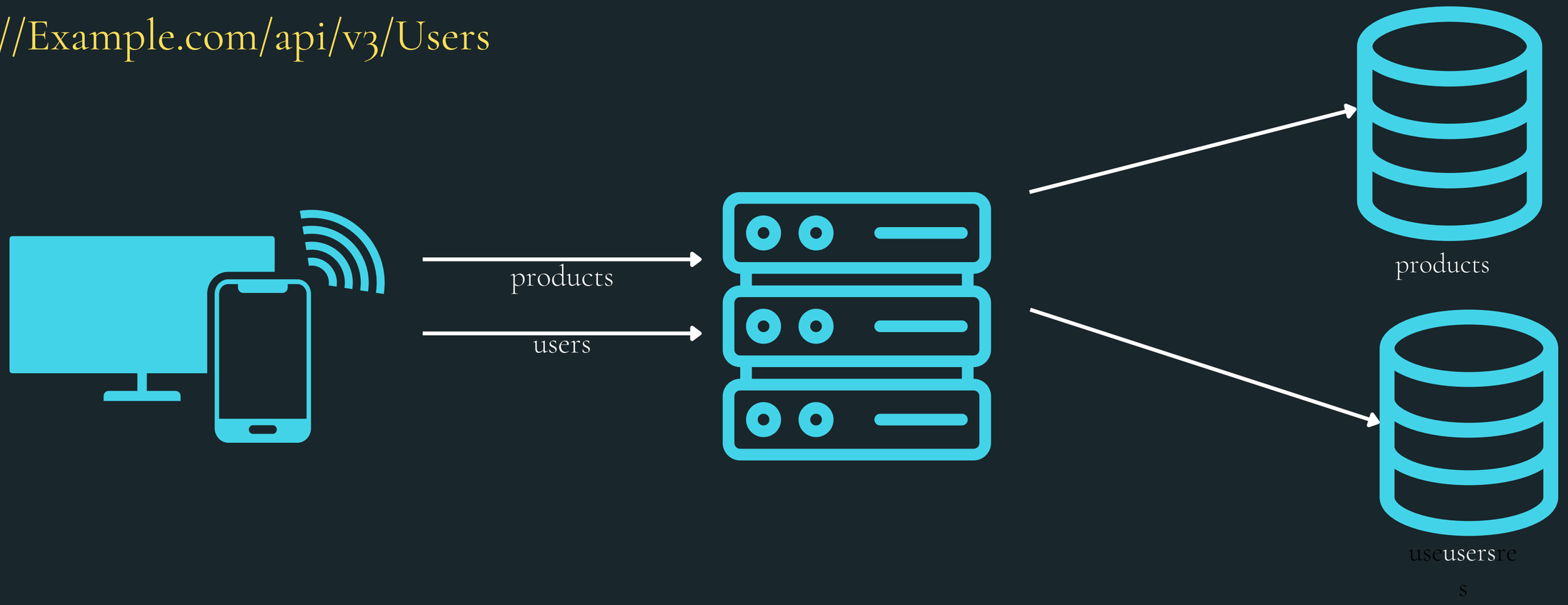


Basics Of Rest

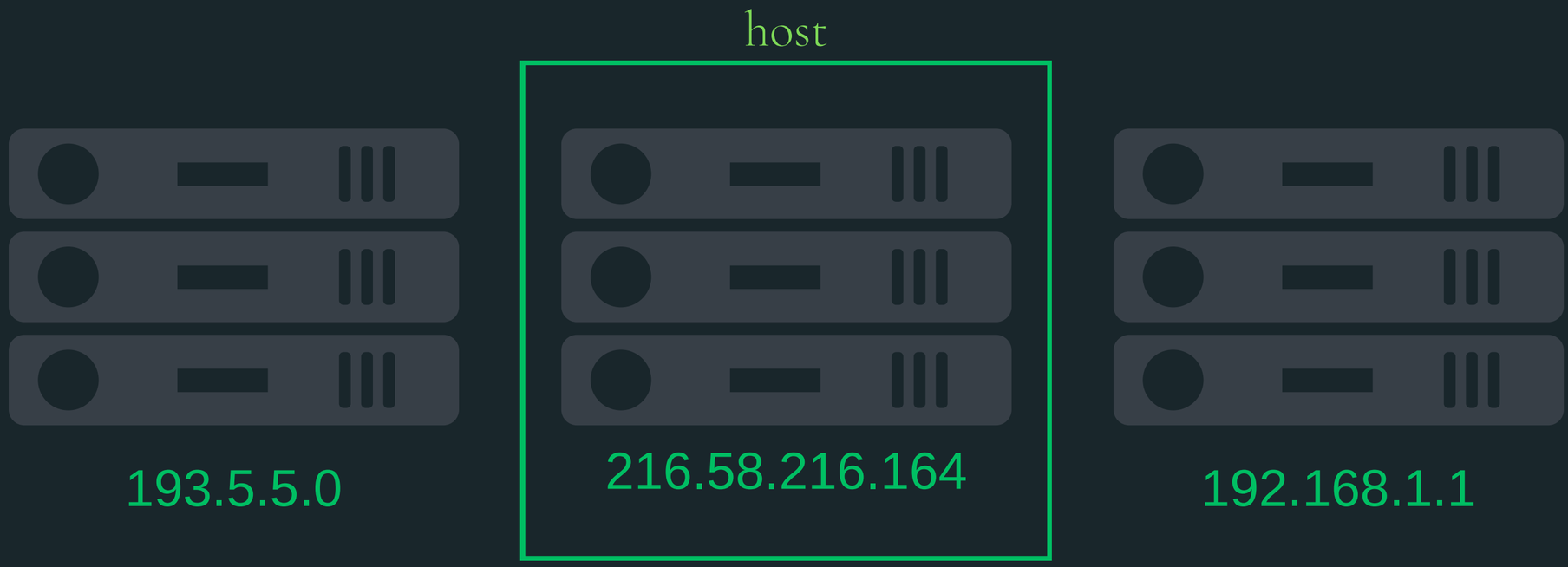
set of unique Uris -Uniform Resource Identifier

Https://Example.com/api/v3/products

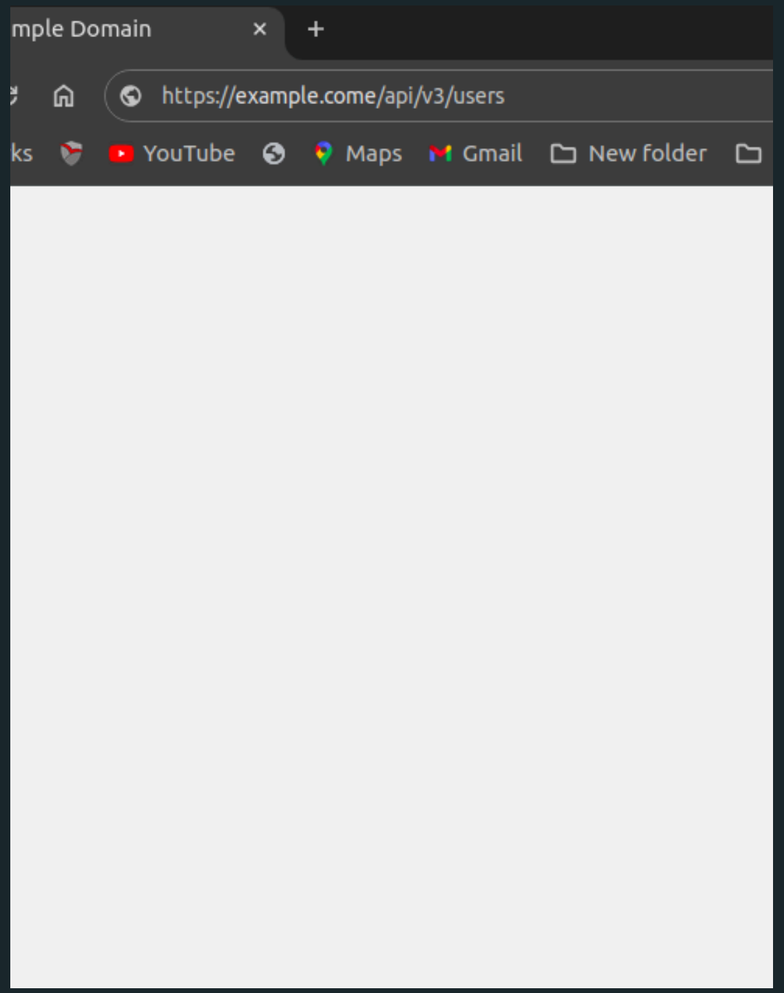
Https://Example.com/api/v3/Users



Little Reminder About Ip and Domains



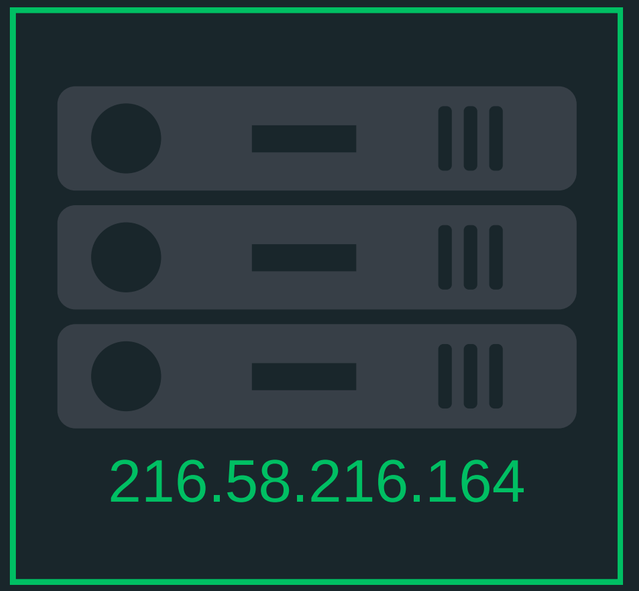
Ip Addresses



look for ip address
associated with that domain

Json/Data/bytes/html page

host



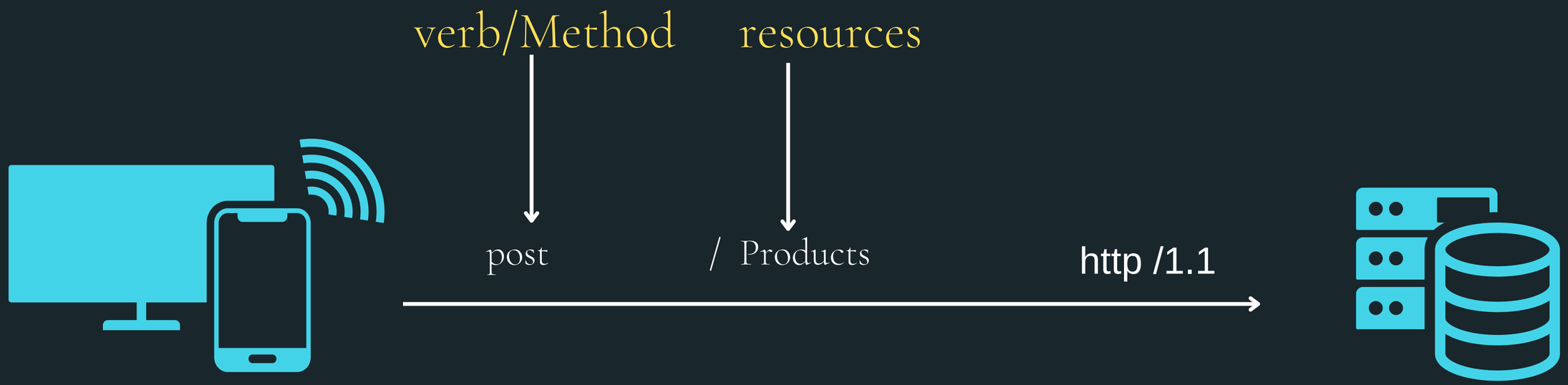
Communcation via Http

Http

HyperText Transfer Protocol

http is used to load webpages using hypertext links.
http is an application layer protocol designed to transfer information between networked devices and runs on top of other layers of the network protocol stack.
A typical flow over HTTP involves a client machine making a request to a server, which then sends a response message.

Http Request



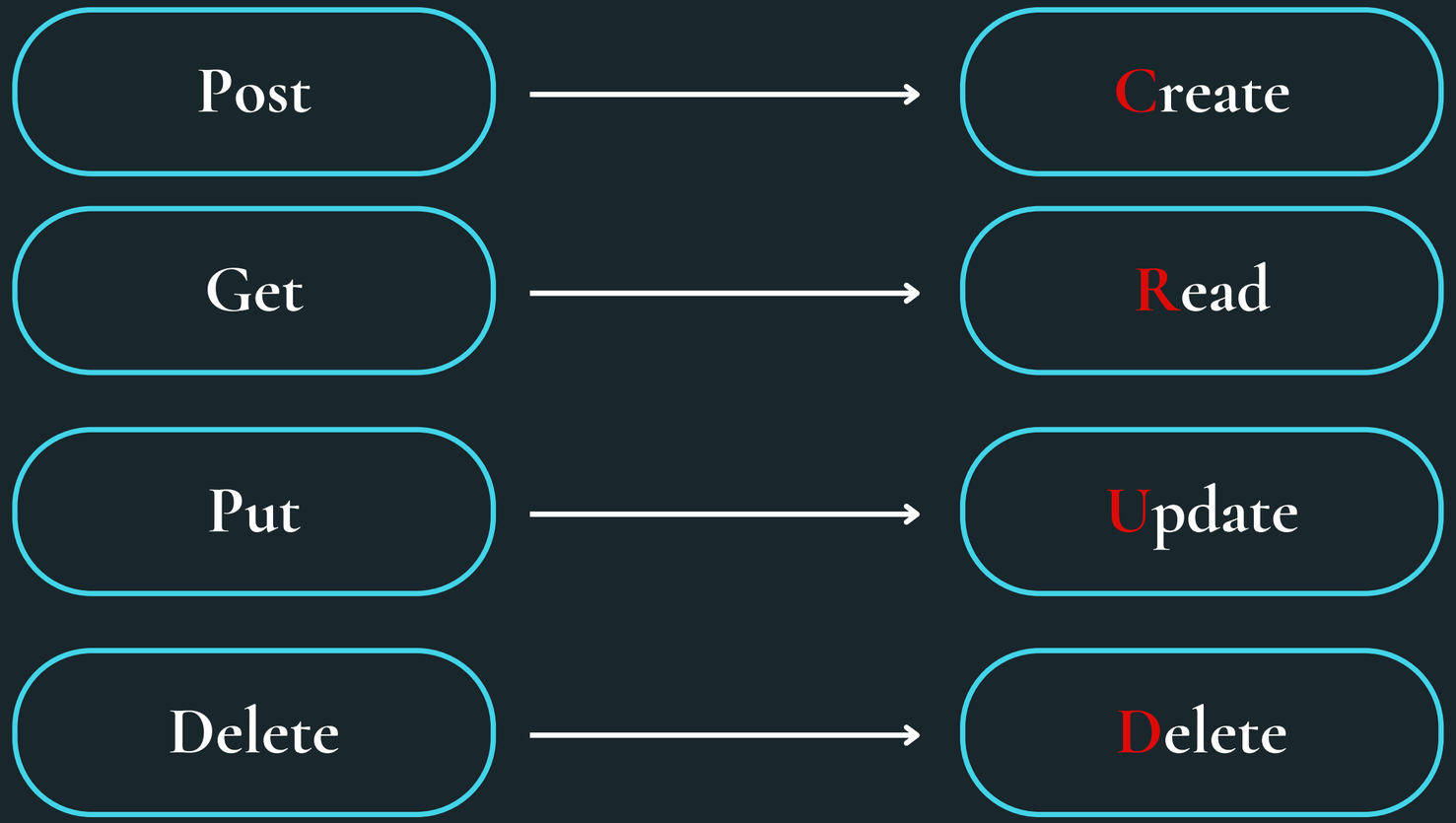
Http Request

An HTTP request is the way Internet communications platforms such as web browsers ask for the information they need to load a website.

Each HTTP request made across the Internet carries with it a series of encoded data
that carries different types of information. A typical HTTP request contains:

1.HTTP version type
2.a URL
3.an HTTP method
4.HTTP request headers
5.Optional HTTP body.

Http Methodes (Common)



CRUD

Http Methodes - 1

GET

GET is called by the client to retrieve the current representation of a specified resource. It is the primary operation for most servers.

HEAD

The HTTP HEAD method is the same as HTTP GET but only transfers the status line and the HTTP response Headers. No message body is included.

OPTIONS

The HTTP OPTIONS method is called by the client to inquire about what operations are available for the specified resource or server in general.

TRACE

The HTTP TRACE method is called by the client to perform a loopback test on the path to the specified resource or one of the proxies in the request chain.

DELETE

The HTTP DELETE method is called by the client to remove the specified resource and all of its current representations from the server.

Http Methodes - 2

PUT

The HTTP PUT method is called by the client to replace the specified resource and all of its current representations, if any, on the server. It is also used to create resources.

POST

The HTTP POST method is called by the client to interact directly with the resource.

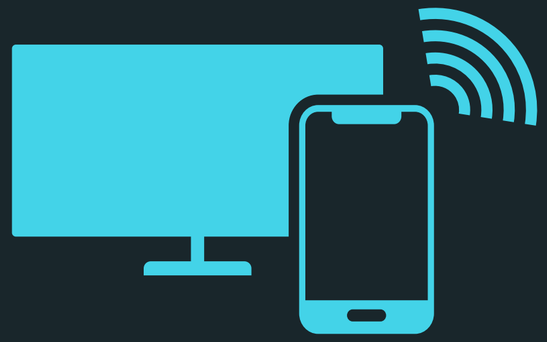
PATCH

The HTTP PATCH method is similar to HTTP PUT in that it is called by the client to change a resource on the server. However, it intended to replace only parts of it. This method can also be used to create resources.

CONNECT

The HTTP CONNECT method is called by the client to establish a tunnel to the origin server, identified by the request-target.

Http Request Headers



POST /products HTTP/ 1.1

Accept : application/json

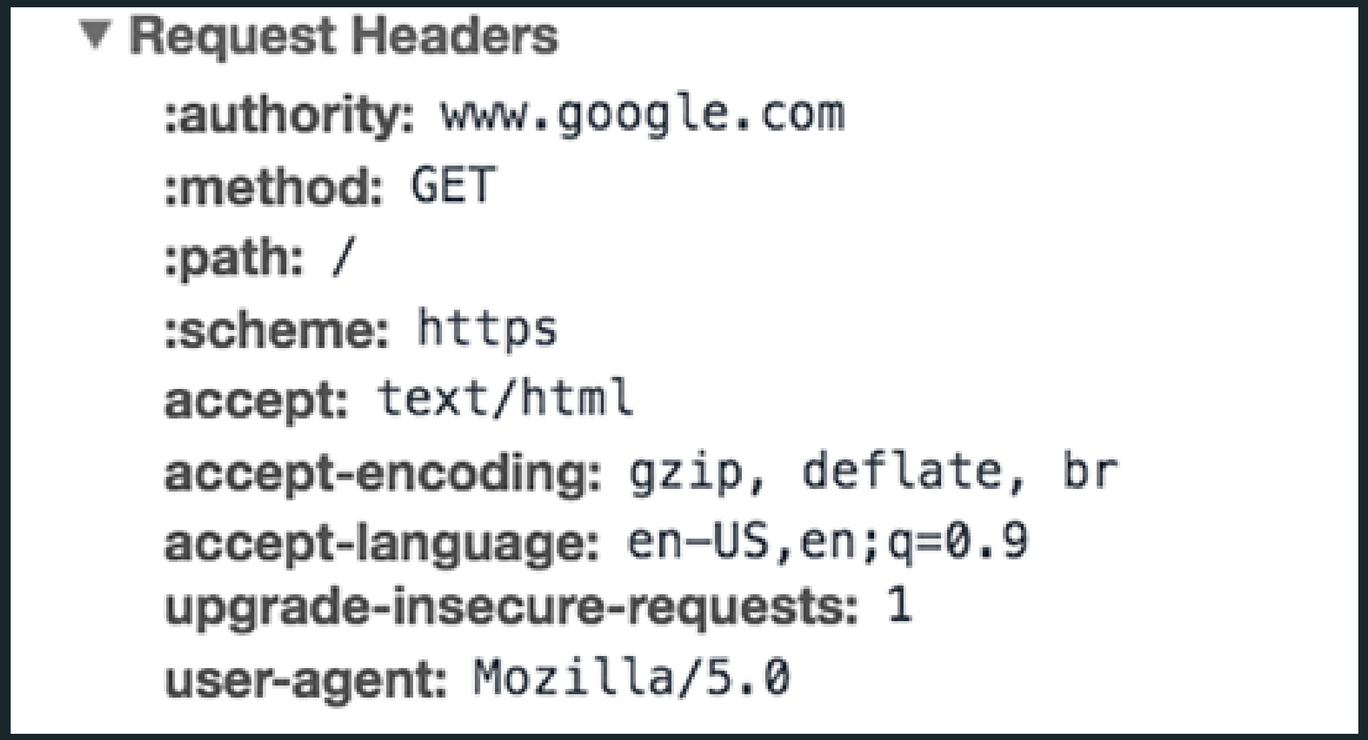
json:

```
{
“customer”:”Isreal Isreali”,
“price”:250,
“amount”:2
}
```

Http Request Headers

HTTP headers are a means for a client, server,
and any intermediaries to exchange information
during the HTTP request-response process

Example of HTTP request headers from Google Chrome's network tab:



Http Request Body

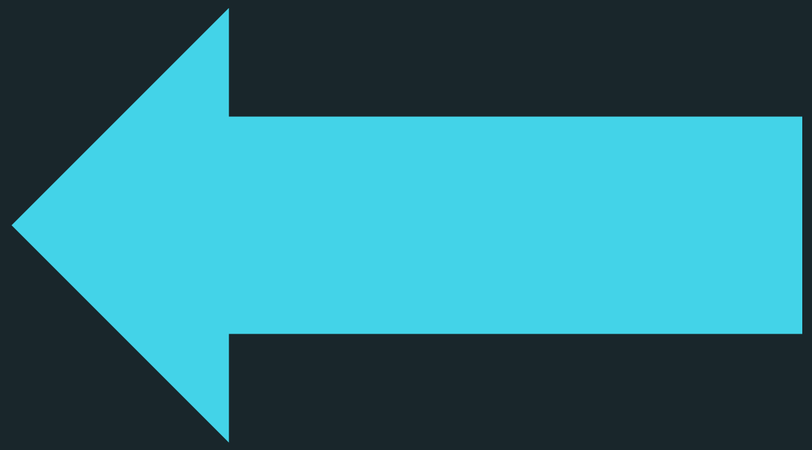
The body of a request is the part that contains the ‘body’ of information the request is transferring. The body of an HTTP request contains any information being submitted to the web server, such as a username and password, or any other data entered into a form.

```
POST /api/users HTTP/1.1
Host: example.com
Content-Type: application/json
Authorization: Bearer <token>

{
    "username": "johndoe",
    "email": "johndoe@example.com",
    "password": "password123",
    "age": 30
}
```

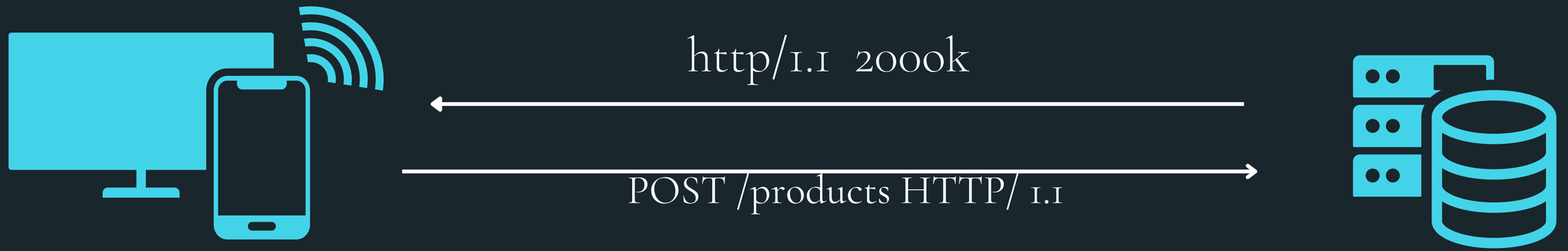
```
POST /api/login HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded

username=johndoe&password=password123
```

Http Response

1. an HTTP status code
2. HTTP response headers
3. optional HTTP body



Http StatusCode



Http StatusCode

HTTP response codes are used to indicate success, failure, and other properties about the result of an HTTP request.
Regardless of the contents of an HTTP response message body,
a client will act according to the response status code

1xx: Informational

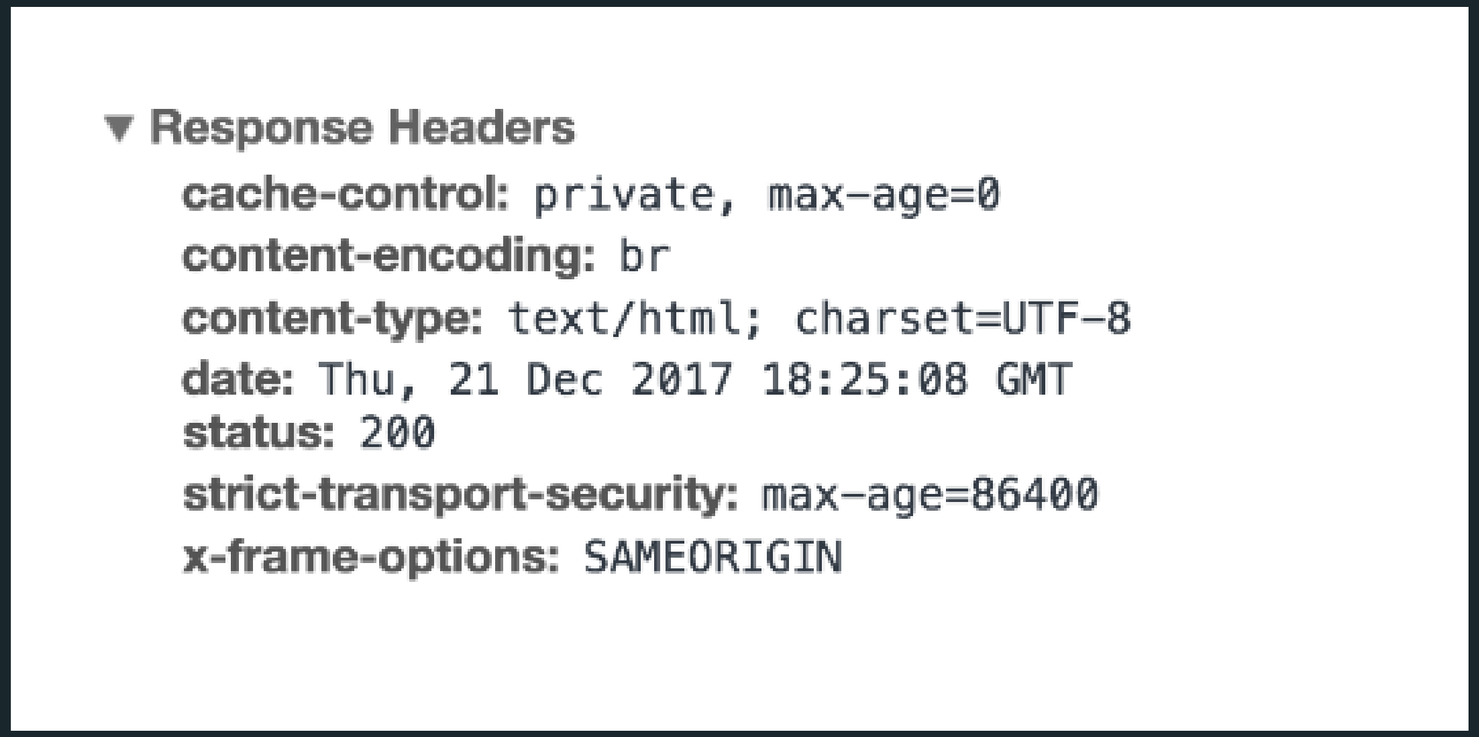
2xx: Success

3xx: Redirection

4xx: Client error

5xx: Server error

Http Response Header



Http Response Body Ex

Json Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "status": "success",
    "data": {
        "id": 1,
        "username": "johndoe",
        "email": "johndoe@example.com",
        "createdAt": "2024-09-08T12:34:56Z"
    }
}
```

Text Response

```
HTTP/1.1 200 OK
Content-Type: text/plain

Operation completed successfully.
```

Http Response Body Ex

Html Response

```
HTTP/1.1 200 OK
Content-Type: text/html

<!DOCTYPE html>
<html>
<head>
    <title>Welcome</title>
</head>
<body>
    <h1>Welcome to the website!</h1>
</body>
</html>
```

Xml Response

```
HTTP/1.1 200 OK
Content-Type: application/xml

<response>
    <status>success</status>
    <data>
        <id>1</id>
        <username>johndoe</username>
        <email>johndoe@example.com</email>
    </data>
</response>
```

Idempotency with HTTP Methods

HTTP Method	Idempotent	Safe
GET	✅	✅
HEAD	✅	✅
PUT	✅	❌
DELETE	✅	❌
POST	❌	❌
PATCH	❌	❌

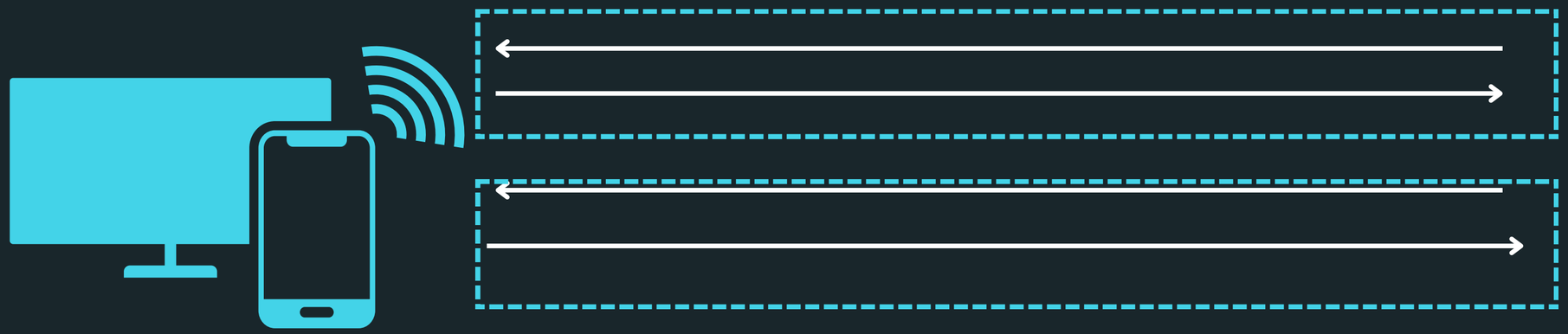
a little brief about 5 architectural constraints that make any web service – a truly RESTful API.

Uniform interface

This is a fundamental principle for the design of any RESTful API.
There should be a uniform and standard way of interacting with a given server for all client types. The uniform interface helps to simplify the overall architecture of the system.
The constraints that help achieve this include:

- Identification of resources: Every system resource should be uniquely identifiable, by using a Uniform Resource Identifier (URI).
- Manipulation of resources through representations: Clients should get a uniform representation of a resource that contains enough information to modify the resource's state in the server, as long as they have the required permissions.
- Self-descriptive messages: Every resource representation should provide enough information for the client to know how to process it further, such as additional actions that can be performed on the resource.
- Hypermedia as the engine of application state (HATEOAS): Clients should have enough information, in the form of hyperlinks, to dynamically discover other resources and drive other interactions.

Rest is Stateless



dont store
state info
in session

independent from all others

dont store
state info
in session

Cacheable

Every response sent by the server should contain information regarding its cacheability.
Simply, the clients should be able to determine whether or not this response can be cached from their side, and if so, for how long.
If a response is cacheable, the client has the right to return the data from its cache for an equivalent request and specified period, without sending another request to the server.
A well-managed caching mechanism greatly improves the availability and performance of an API by completely or partially eliminating some of the client-server interactions.
However, this increases the chances of users receiving stale data.

Client-server

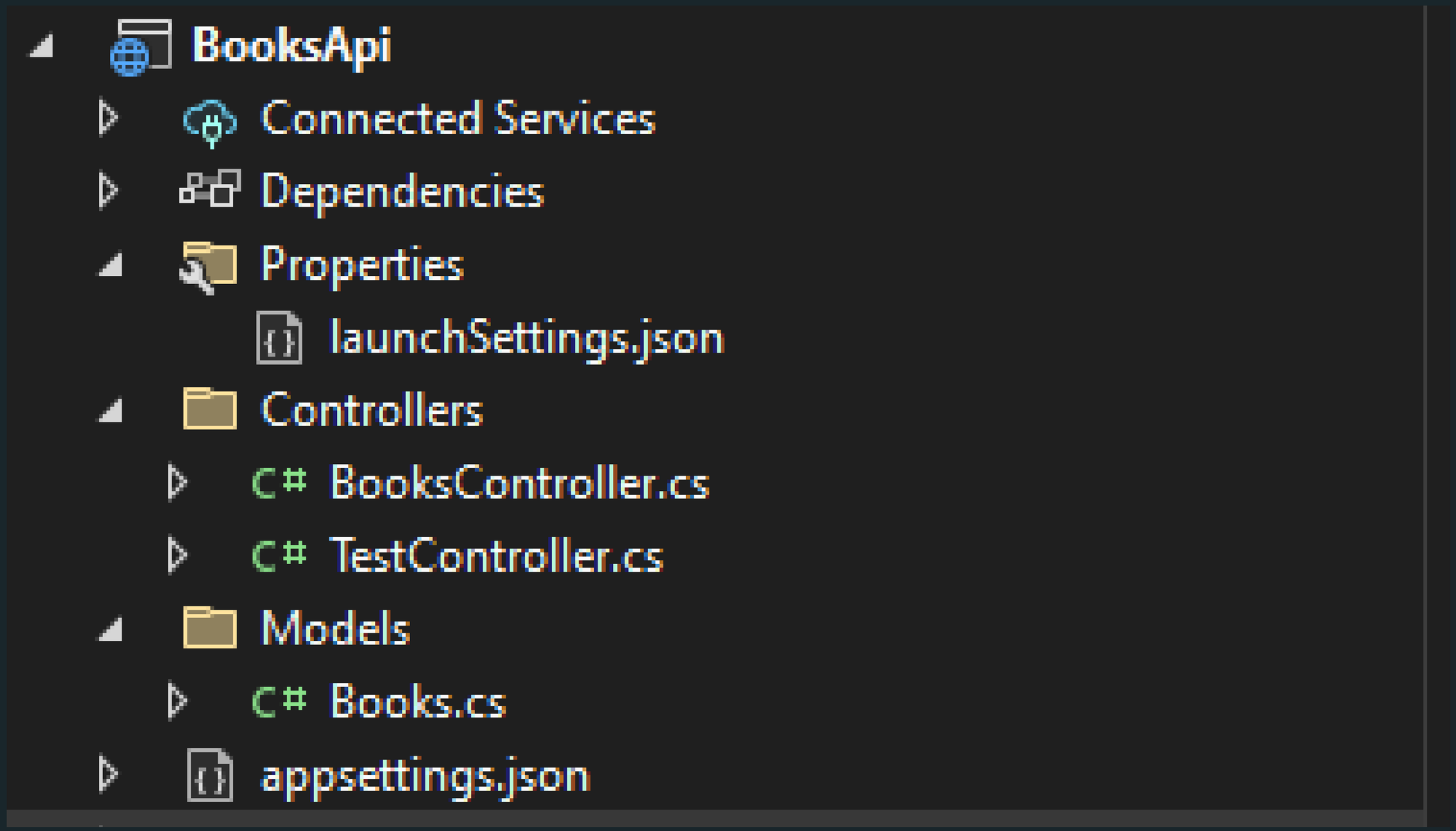
client applications and server applications MUST be able to evolve separately without any dependency on each other. A client should know only resource URIs, and that’s all.

Layerd System

An application with layered architecture is composed of various hierarchical layers. Each layer has information from only itself, and no other layer. There can be multiple intermediate layers between the client and server. You can design your REST APIs with multiple layers, each having tasks such as security, business logic, and application, all working together to fulfill client requests. These layers are invisible to the client and only interact with intermediate servers. This architecture improves the overall system availability and performance by enabling load balancing and shared caches.

Starts Develope Rest API in .Net 8

First Off All Create Controllers and Models

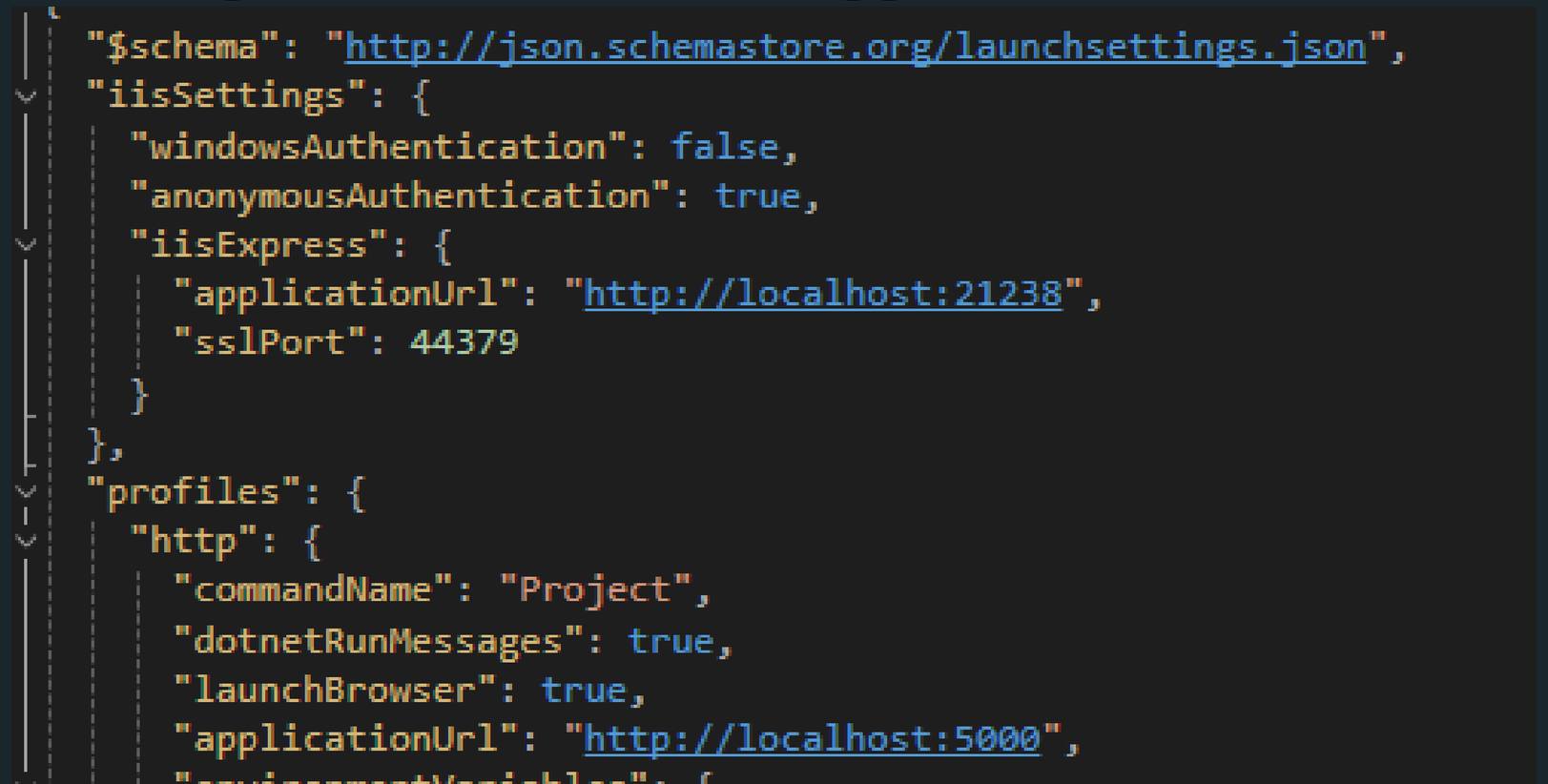


Controller Include the crud method

```
public ActionResult<IEnumerable<Book>> GetBooks()
{
    return booksList;
}

//Get : api/books/{id}
[HttpGet("{id}")]
1 reference
public ActionResult<Book> GetBook(int id)
{
    var Book = booksList.FirstOrDefault(x => x.Id == id);
    if (Book == null)
    {
        return NotFound();
    }
    return Book;
```

We Need To Pay atttention to the LaunchSettings.json

The launchSettings.json file is part of a .NET project's configuration,
typically located under the Properties folder.
It is used during development to configure how the application is launched, defining settings like environment variables, profiles, and how the application should run in development environments



And Of Course To the Program.cs

Program.cs file in a .NET project is where the application's entry point is defined

```
// Add services to the container.
builder.Services.AddControllers(); // This registers controller service

var app = builder.Build();


// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage(); // Detailed error pages in develop
}
else
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts(); // Use HSTS in production
```

Program.cs

First Lines:

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

This method is used to create a builder object that will help configure and build an instance of the WebApplication.

It prepares the application to use essential services like routing, dependency injection, configuration, and logging

Program.cs- Explain WebApplication.CreateBuilder(args)

1. Creating an Application Builder and Configuring the Host

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

// Access environment settings
var environment = builder.Environment.EnvironmentName;

// Load custom configuration file
builder.Configuration.AddJsonFile("customsettings.json", optional: true, reloadOnChange: true);
```

2. Registering Services

```
builder.Services.AddControllers(); // Add MVC controllers
builder.Services.AddLogging(); // Configure logging
builder.Services.AddDbContext<TasksDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("Connection"))
);
builder.Services.AddScoped<IMyService, MyService>(); // Register a custom service
```

Program.cs

3. Configuring the Web Server andMiddleware

The builder allows you to configure the web server settings, such as the URL, ports, and server features

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

app.Use(async (context, next) =>
{
    Console.WriteLine("Handling request: " + context.Request.Path);
    await next();
    Console.WriteLine("Finished handling request.");
});
```

Program.cs

4. Building and Running the Application

```
app.UseRouting();
app.UseAuthorization();

app.MapControllers(); // Maps attribute-routed controllers

// Start the app
app.Run();
```

Program.cs

Important Lines:

app.UseRouting(); // Adds routing to the request pipeline

app.MapControllers(); // This maps attribute-based routing controllers like [Route("api/[controller]")]

```
namespace BooksApi.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    0 references | yvrachel, 2 days ago | 1 author, 1 change
    public class BooksController : ControllerBase
    {
```

Deep Dive Into Routes

Routes Types

1. Route Parameters

```
[HttpGet("{id}")]
public IActionResult GetItemById(int id)
{
    // Logic to get the item by id
}
```

/api/items/123

2. Query Parameters

```
[HttpGet]
public IActionResult GetItemsByFilter(string name, int page)
{
    // Logic to filter items by name and page number
}
```

/api/items?name=test&page=2

Routes Types

3. Body Parameters

```
[HttpPost]
public IActionResult CreateItem([FromBody] ItemDto item)
{
    // Logic to create the item using the body data
}
```

```
{
  "name": "NewItem",
  "price": 100
}
```

When to use every type:

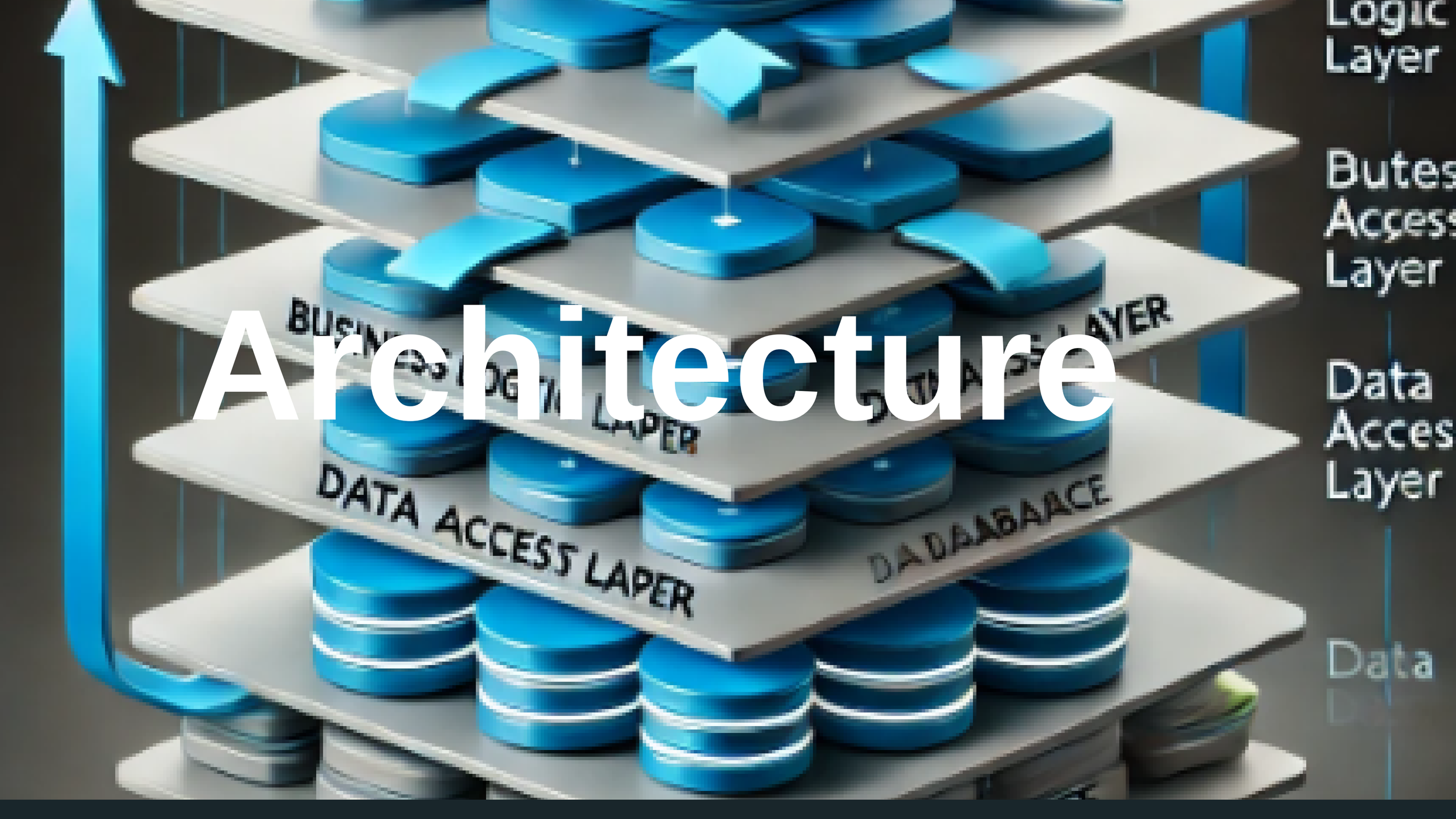
- Route Parameters are generally used for identifying resources (e.g., IDs).
- Query Parameters are ideal for optional filtering, sorting, and pagination.
- Body Parameters are useful for passing complex objects in the request body (usually in POST or PUT).

```
/api/items?name=test&page=2
```

PostMan-

Postman is an API platform for building and using APIs. Postman simplifies each step of the API lifecycle and streamlines collaboration so you can create better APIs—faster.

https://www.postman.com/



Architecture

3-Tier Architecture (The Most Common Model):

PRESENTATION TIER-
(USER INTERFACE (UI) LAYER)

Application Tier-
The core of the application, this layer contains business rules and logic

Data Tier-
(EF,ADO)

4-Tier Architecture

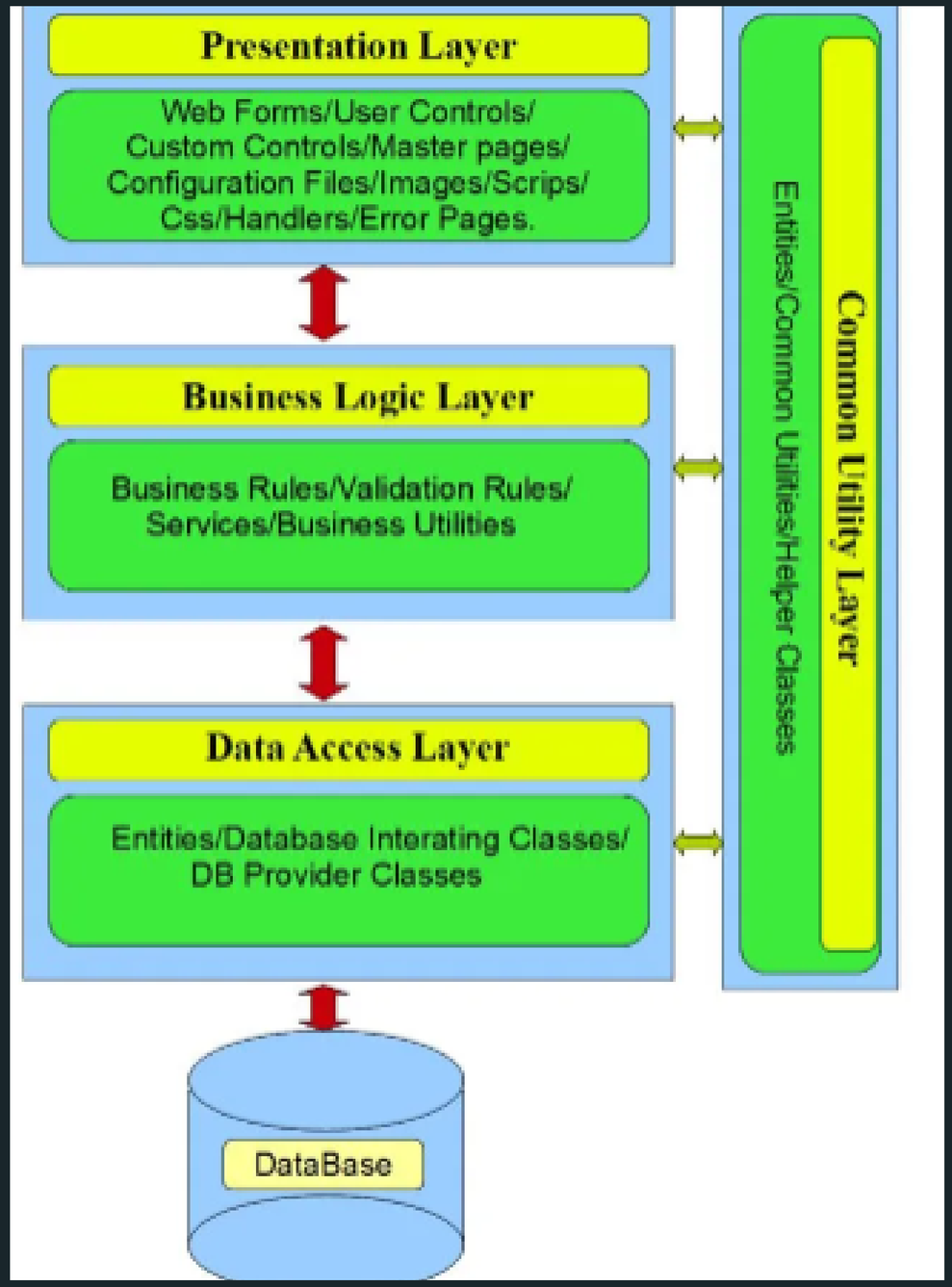
PRESENTATION TIER-
(USER INTERFACE (UI) LAYER)

Application Tier-
The core of the application, this layer contains business rules and logic

Business Tier-
(logic)

Data Tier-
(EF,ADO)

Power of 3-Tier Architecture



Presentation Layer

handles user interface and interactions, including HTTP request handling and response generation.
controllers and routing are key components of the Presentation Layer

Role and Responsibilities

- Receives and handles incoming HTTP requests from clients.
- Manages the user interface and user interactions.
- Maps requests to appropriate controllers and action methods.
- Validates and sanitizes user inputs.
- Orchestrates the communication between clients and the underlying business logic layer.

Presentation Layer

Controllers

Routing

content

Input/Output Data

Error Handling

Business Layer

Sits between the Presentation Layer and the Data Access Layer

Role and Responsibilities:

- Implements the business rules and logic of the application.
- Processes and validates data received from the Presentation Layer.
- Enforces business policies and rules.
- Orchestrates the interaction between the Presentation Layer and the Data Access Layer.

Business Layer

Business Rules Enforcement

Integration with External Systems

validation

calculations

Transformations

Unit Testing

Authentication and Authorization

workflow management

Validation

Data access layer

The Data Access Layer facilitates the interaction
between the application and the underlying data storage systems

Benefits

1.Modularity and Scalability

2.Enhanced Security and Authentication

3.Code Reusability and Maintainability

Example

- Controllers (Presentation Layer)
  - BooksController.cs
- Services (Business Logic Layer)
  - BookService.cs
  - IBookService.cs
- Repositories (Data Access Layer)
  - BookRepository.cs
  - IBookRepository.cs
- Models
  - Book.cs
- Program.cs
- books.json (File for storing book data)

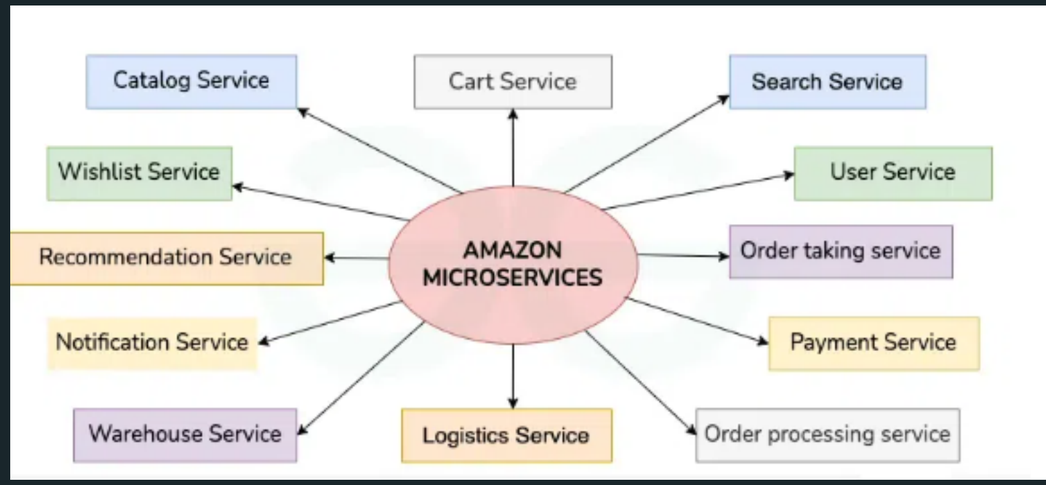
Architecture

Clean Architecture (Onion Architecture)

CQRS (Command Query Responsibility Segregation)

Microservices Architecture

Event-Driven Architecture



Swagger in net 8

Steps:

1. Install Swashbuckle.AspNetCore.Swagger package
2. Modify Program.cs to add Swagger services and middleware.
3. Run the application, and Swagger UI will be available to interact with your API

Swagger definition:

```
builder.Services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new OpenApiInfo
    {
        Version = "v1",
        Title = "Books API",
        Description = "A simple example ASP.NET Core API to manage books"
        Contact = new OpenApiContact
        {
            Name = "Your Name",
            Email = "your.email@example.com",
            Url = new Uri("https://yourwebsite.com"),
        }
    });
});
```

Swagger definition:

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

    // Enable middleware to serve generated Swagger as a JSON endpoint.
    app.UseSwagger();

    // Enable middleware to serve Swagger UI (HTML, JS, CSS, etc.)
    app.UseSwaggerUI(c =>
    {
        c.SwaggerEndpoint("/swagger/v1/swagger.json", "Books API V1");
        c.RoutePrefix = string.Empty; // Serve Swagger UI at the app's root
    });
}
```

Swagger -Result...

Books

- GET /api/Books
- POST /api/Books
- GET /api/Books/{id}
- PUT /api/Books/{id}
- DELETE /api/Books/{id}

Test

- GET /api/Test

AppSettings (Web Config)

The main configuration file used to store key-value pairs, such as connection strings, API keys, and other settings.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "ConnectionStrings": {
    "DefaultConnection": "Server=myServer;Database=myDB;User=myUser;Password=myPass;"
  }
}
```

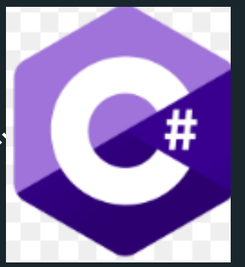
Connection To DB

Entity Framework

Object Relational Mapping

books	
Id	int
Name	varchar
Auther	varchar
Description	varchar(50)

```
public class Book
{
    7 references | - changes | -authors, -changes
    public int Id { get; set; }
    4 references | - changes | -authors, -changes
    public string? Name { get; set; }
    4 references | - changes | -authors, -changes
    public string? Author { get; set; }
    2 references | - changes | -authors, -changes
    public string? Description { get; set; }
}
```



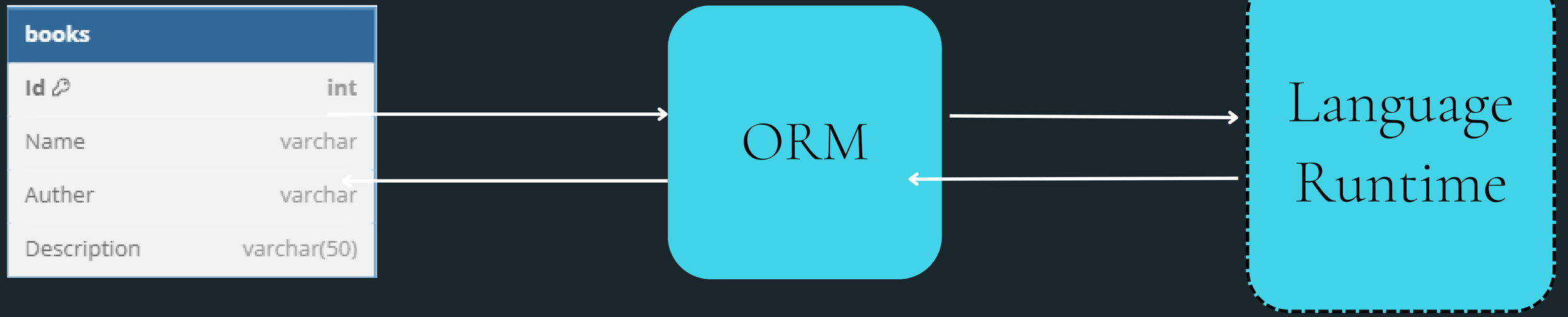
Object Relational Mapping

Object-Relational Mapping, is a programming technique that connects the data in a relational database with the objects in your code

Think of it as a translator between the application and the database

~~select * from books~~

Relational



Non Relational

Object Relational Mapping

For Example -

```
var books = context.Books.Where(b => b.Author==Haim").ToList();
```

context.Books acts like a collection of Book objects,
and use LINQ (a query language in C#) to filter books by author.
The ORM translates this into an SQL query like:

```
SELECT * FROM Books WHERE Author = 'Haim';
```

Object Relational Mapping

Advantages :

1.Less SQL: Instead of writing raw SQL queries, you use code, which is often easier to read and maintain.
2.Data Consistency: ORM libraries help keep data consistent by managing transactions and changes to objects.
3.Productivity: ORMs automate common database tasks, freeing you to focus on business logic.

Entity Framework

Orm library for .net

can work with variety of databases

Entity Framework

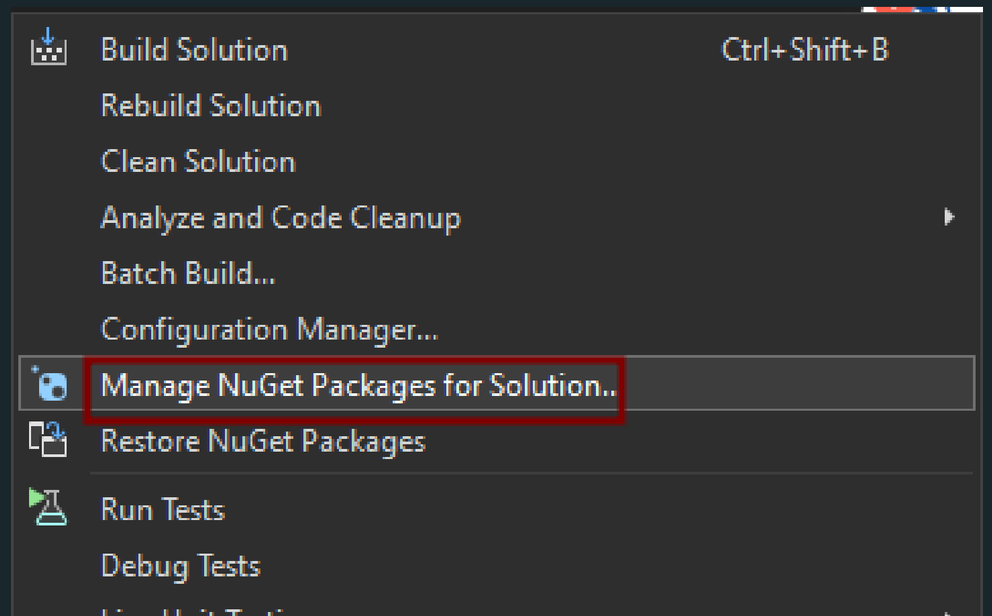
steps to use EF:

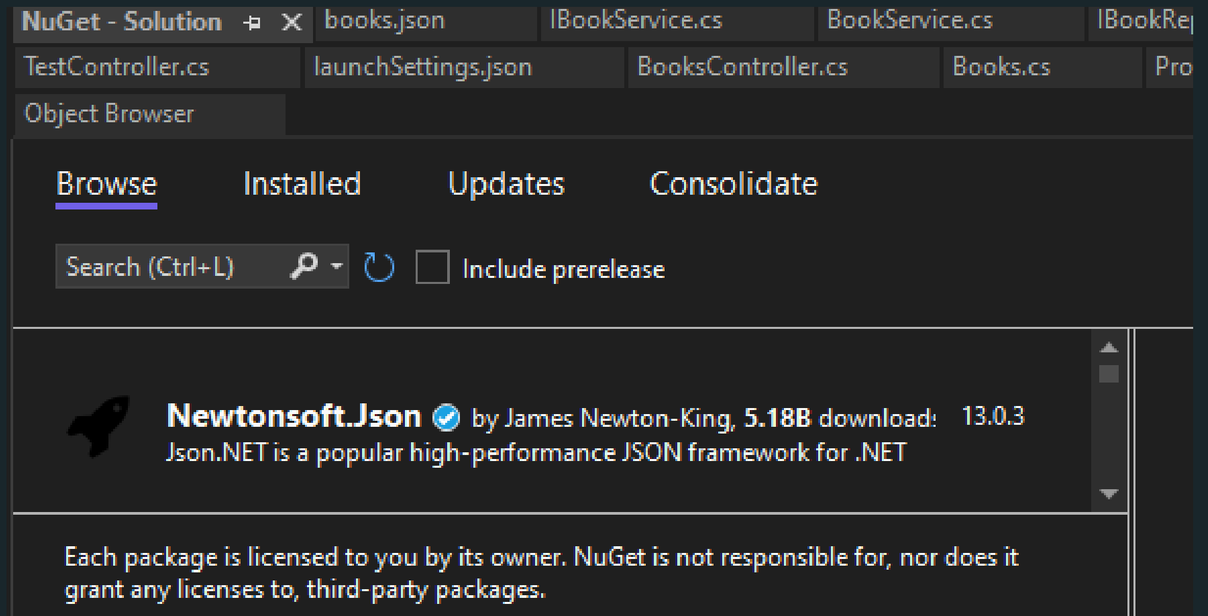
1. install packages
 Microsoft.EntityFrameworkCore.SqlServer
 Microsoft.EntityFrameworkCore.Tools

from nuget - explanation about it in next page....

Nuget-https://www.nuget.org/packages

NuGet is a free and open-source package manager designed specifically for the Microsoft development platform, including .NET. It simplifies the process of incorporating reusable code (known as packages) into your projects, managing dependencies, and sharing code with others





Entity Framework

steps to use EF:

2. Define a DbContext

   The DbContext represents the connection to the database and allows querying and saving data, acts as a bridge between your .NET code and the database.

Entity Framework-DBContext

every table define as a DbSet<Books> the name of the class who repersenate the table

```
public BooksDbContext()
{
}

0 references | 0 changes | 0 authors, 0 changes
public BooksDbContext(DbContextOptions<BooksDbContext> options)
    : base(options)
{
}
```

Entity Framework-DBContext

```
0 references | 0 changes | 0 authors, 0 changes
public BooksDbContext(DbContextOptions<BooksDbContext> options)
    : base(options)
{
}
```

- This is the constructor for the BooksDbContext class.
- It takes a single parameter, options, of type DbContextOptions<BooksDbContext>.
- DbContextOptions is a class that holds configuration options for the context, such as the database provider (e.g., SQL Server, SQLite) and connection string.
- Passing these options allows the configuration (such as which database to connect to) to be set outside the BooksDbContext class itself, usually in the Startup or Program file when setting up dependency injection.

Entity Framework-DBContext

```
0 references | 0 changes | 0 authors, 0 changes
public BooksDbContext(DbContextOptions<BooksDbContext> options)
    : base(options)
{
}
```

- This part is calling the base class constructor of DbContext, passing in the options parameter.
- By passing options to the base class, you are configuring the base DbContext class with the settings provided, such as the connection string and other configuration details.
- This lets EF Core know how to set up the database context, including the type of database and other context-specific settings.

Entity Framework-DBContext

```
0 references | 0 changes | 0 authors, 0 changes
public BooksDbContext(DbContextOptions<BooksDbContext> options)
    : base(options)
{
}
```

- Since no additional setup is needed in the constructor, the body is empty.
- This keeps the constructor simple and allows all configuration to be managed through the options provided.

Entity Framework-DBContext

This constructor is a standard way to set up an EF Core DbContext using dependency injection.

When using EF Core in a .NET application (especially in ASP.NET Core), this allows the BooksDbContext to be configured and injected automatically by the .NET runtime.

It enables you to set the database connection details in a centralized configuration file (e.g., appsettings.json) rather than hardcoding them within the context class

Entity Framework

steps to use EF:

3.Configure Connection String in appsettings.json

```
"ConnectionStrings": {
    "DefaultConnection": "Server=your_server;Database=your_db;User Id=your_use
}
```

Connection String

A connection string is a series of parameters that specify how an application connects to a database. Each part of the connection string has a specific purpose, allowing you to customize the connection based on your requirements, such as database location, authentication method, and connection propertie

```
Server=myServerAddress;Database=myDataBase;User Id=myUsername;Password=myPassword;
```

Connection String

Server=myServerAddress;Database=myDataBase;User Id=myUsername;Password=myPassword;

1.Server -Specifies the address of the SQL Server instance.
It can be a network address, a local server name, or an IP address.
2.Database/initial catalog - Specifies the name of the database on the SQL Server instance
3.User Id and Password -Provide the credentials for SQL Server authentication
4.Integrated Security (or Trusted_Connection) - pecifies whether to use Windows Authentication (integrated security) or SQL Server Authentication.

Entity Framework

steps to use EF:

4.Configure DbContext in Program.cs

```
builder.Services.AddDbContext<BooksDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("Defaul
```

Entity Framework

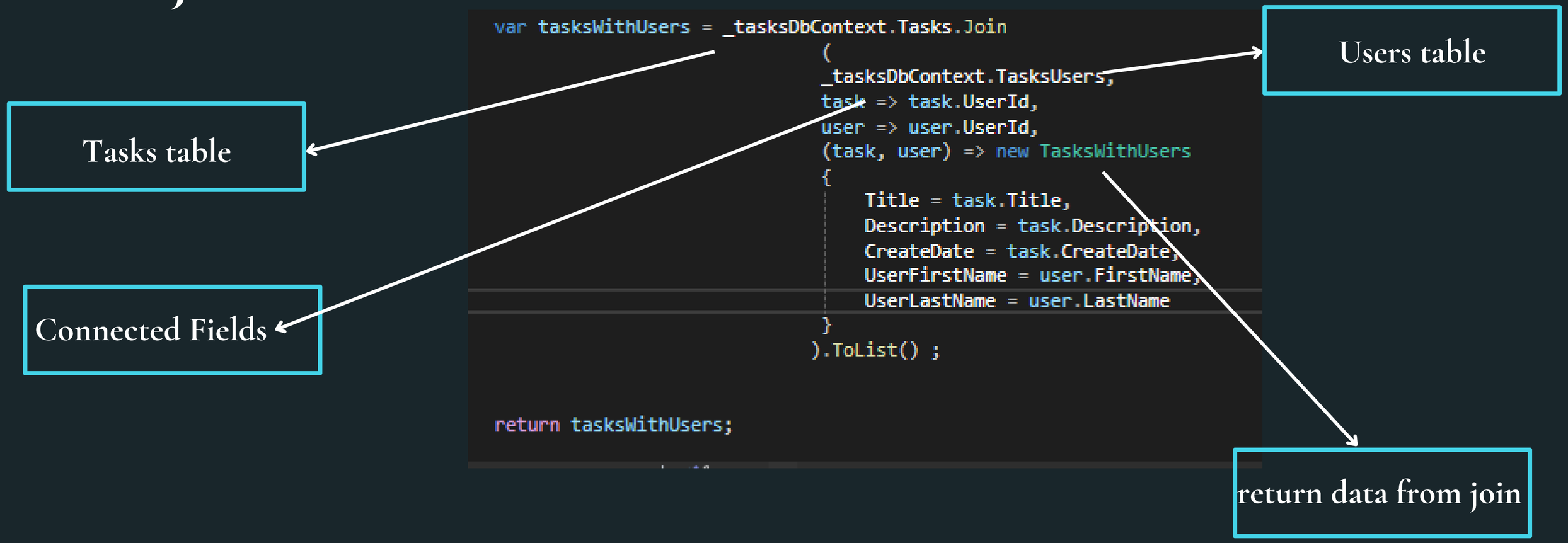
steps to use EF:

5. use it...

```
public class CustomerService
{
    private readonly AppDbContext _context;

    public CustomerService(AppDbContext context)
    {
        _context = context;
    }

    public List<Customer> GetCustomers()
    {
        return _context.Customers.ToList();
    }
}
```

LINQ In C#- Join Functions

Inner join



LINQ In C#- Join Functions

left join

```
var tasksWithUsers = _tasksDbContext.Tasks  //tasks table
                .GroupJoin(
                _tasksDbContext.TasksUsers,  //users table
                task => task.UserId,//fk
                user => user.UserId,//pk
                (task, users) => new
                {
                    Task = task,
                    User = users.FirstOrDefault() // Take the first user if it exists, otherwise null
                })
                .Select(joinResult => new TasksWithUsers
                {

                    Title = joinResult.Task.Title,
                    Description = joinResult.Task.Description,
                    UserFirstName = joinResult.User.FirstName,
                    UserLastName = joinResult.User.LastName
                })
                 .ToList();


return tasksWithUsers;
```

LINQ In C#- Join Functions
Include (like join)

```
var tasksWithUsers = _tasksDbContext.Tasks
                                    .Include(t => t.User) //like join
                                    .Select(t => new TasksWithUsers
                                    {
                                        TaskId = t.TaskId,
                                        Title = t.Title,
                                        Description = t.Description,
                                        CreateDate = t.CreateDate,
                                        UpdateDate = t.UpdateDate,
                                        UserFirstName = t.User != null ? t.User.FirstName : null,
                                        UserLastName = t.User != null ? t.User.LastName : null
                                    })
                                    .ToList();

return tasksWithUsers;
```

LINQ In C#- Common Functions

1. Any vs. All

Any: Checks if any element in a collection matches a condition (returns true if at least one match is found).

All: Checks if all elements in a collection satisfy a condition (returns true only if every element matches).

```
var tasks = new List<Tasks>

    new Tasks { Title = "Task 1", Description = "Description 1", UserId = 1 },
    new Tasks { Title = "Task 2", Description = "Description 2", UserId = null }
};

// Any: Checks if there are any tasks with a UserId
bool hasAssignedTasks = tasks.Any(t => t.UserId != null); // true

// All: Checks if all tasks have a UserId
bool allAssignedTasks = tasks.All(t => t.UserId != null); // false
```

LINQ In C#- Common Functions

2.Contains vs. Does Not Contain

Contains: Checks if a collection contains a specific element.

!Contains (inverse): Checks if a collection does not contain a specific element

```
var taskTitles = new List<string> { "Task 1", "Task 2", "Task 3" };
// Contains
bool containsTask2 = taskTitles.Contains("Task 2"); // true

// Does Not Contain
bool doesNotContainTask4 = !taskTitles.Contains("Task 4"); // true
```

LINQ In C#- Common Functions

3. Where vs. Except

Where: Filters elements based on a condition.

Except: Returns elements that are not in a specified collection.

```
tasksWithUserIds = _tasksDbContext.Tasks.Where(t => t.UserId != null); // Returns tasks with

allTaskIds = new List<int> { 1, 2, 3 };
assignedTaskIds = _tasksDbContext.Tasks.Select(t => t.UserId.GetValueOrDefault()).ToList();

xcept: Returns task IDs that don't have UserId assigned
unassignedTaskIds = allTaskIds.Except(assignedTaskIds);
```

LINQ In C#- Common Functions

4. First vs. Last

First: Retrieves the first element that matches a condition.

Last: Retrieves the last element that matches a condition.

```
var firstTaskWithUser =_tasksDbContext.Tasks.FirstOrDefault(t => t.UserId != null);
var lastTaskWithUser = _tasksDbContext.Tasks.LastOrDefault(t => t.UserId != null);
```

LINQ In C#- Common Functions

6. Select vs. SelectMany

Select: Projects each element of a sequence into a new form.

SelectMany: Projects each element of a sequence to a collection and flattens the resulting collections into one sequence

```
var taskDescriptions =_tasksDbContext.Tasks.Select(t => t.Description); // List of descriptions

var usersWithTasks = _tasksDbContext.TasksUsers
    .SelectMany(user => user.Tasks, (user, task) => new { user.UserId, task.Title });
```

LINQ In C#- Common Functions

7.Count vs. Sum

Count: Counts the number of elements in a collection.

Sum: Sums the values of a numeric property in a collection.

```
int totalTasks = _tasksDbContext.Tasks.Count(); // Counts the total number of tasks
int sumUserIds = _tasksDbContext.Tasks.Sum(t => t.UserId ?? 0); // Sum of UserIds (ignoring nulls)
```

LINQ In C#- Common Functions

8. Distinct vs. Union

Distinct: Removes duplicates from a collection.

Union: Combines two collections and removes duplicates.

```
var distinctTitles = _tasksDbContext.Tasks.Distinct(); //return non dulicate valuews

var taskTitlesList2 = new List<Tasks> { new Tasks() { Title = "test", TaskId = 3 } };
var unionTitles = _tasksDbContext.Tasks.Union(taskTitlesList2);
```

Entity Framework -Run Procedure/query

1. return data from proedure/query

```
erences | yvrachel, 41 days ago | 1 author, 1 change
lic async Task<List<Book>> GetBooksByAuthorAsync(int authorId)

return await Books.FromSqlRaw("EXEC GetBooksByAuthor @AuthorId = {0}", authorId).ToL
```

2.Executing a Stored Procedure/query That Does Not Return Data

```
public async Task<int> UpdateBookData(int name, string id)
{
    var sql = "UPDATE Books SET name = @name WHERE id = @id";
    var parameters = new[]
    {
    new SqlParameter("@name", name),
    new SqlParameter("@id", id)
};

    return await _context.Database.ExecuteSqlRawAsync(sql, parameters);
}
```

Entity Framework -Run Procedure/query

3.Handling Complex Results

If your stored procedure returns a result set that doesn't directly map to an entity, you can define a class to hold the result and use FromSqlRaw with that class

```
public class DataSourceResult
{
    public int Id { get; set; }
    public string Name { get; set; }
    public bool IsActive { get; set; }
}

// Call the stored procedure and map results to DataSourceResult
var status = true;
var results = await context.Set<DataSourceResult>()
    .FromSqlRaw("EXEC GetDataSourcesByStatus @Status = {0}", status)
    .ToListAsync();
```

Entity Framework -Run Procedure/query

4.Using Output Parameters

For stored procedures with output parameters,
use DbParameter objects with Direction =parameterDirection.Output

```
var sql = "EXEC UpdateBookData @AuthorId,@OutputParam OUTPUT";
var outputParam = new SqlParameter
{
    ParameterName = "@OutputParam",
    SqlDbType = SqlDbType.Int,
    Direction = ParameterDirection.Output
};

var parameters = new[]
{
new SqlParameter("@AuthorId", AuthorId),
outputParam
};

await _context.Database.ExecuteSqlRawAsync(sql, parameters);
int result = (int)outputParam.Value;
```

Entity Framework -Run Procedure/query

Summary

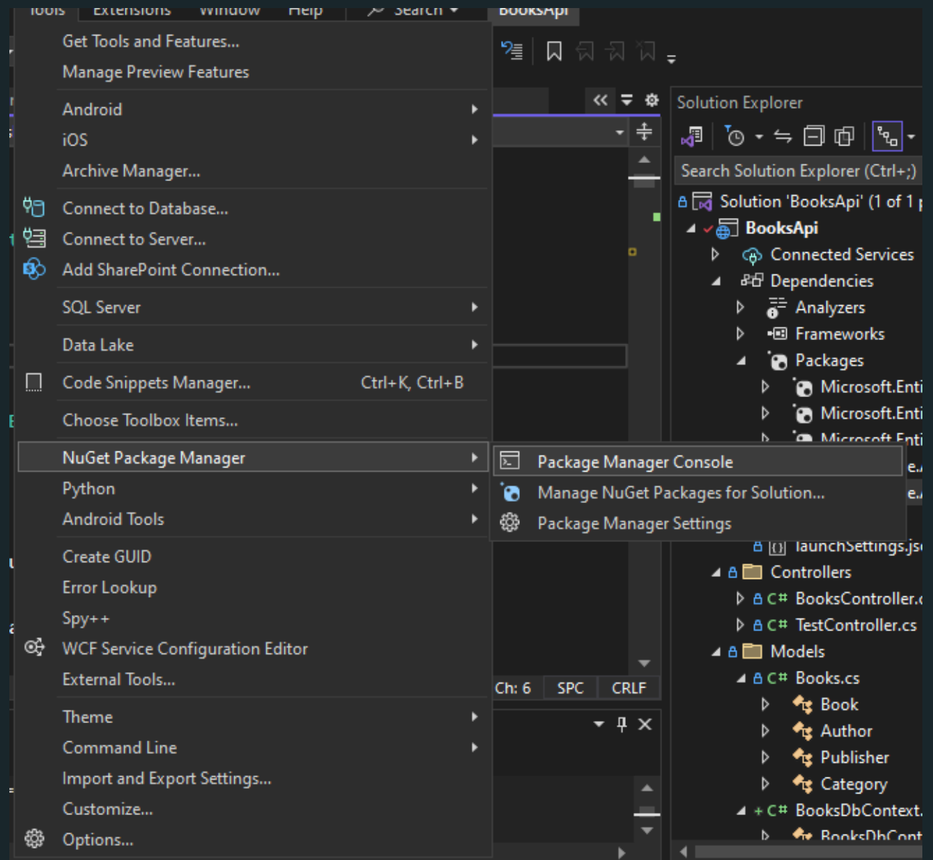
- Use FromSqlRaw or FromSqlInterpolated to execute stored procedures that return data.
- Use ExecuteSqlRaw or ExecuteSqlInterpolated for stored procedures that perform actions but don’t return data.
- For complex results, define a custom class and map the procedure result to it.
- For output parameters, use DbParameter with Direction = Output

Entity Framework-Database First

creates model automatic:

1.create tables,procedures etc in the database

2.open dotnet cli or the Package Manager Console



Entity Framework-Database First

3.navigate to the project folder and then run:
dotnet ef dbcontext scaffold "YourConnectionString"
Microsoft.EntityFrameworkCore.SqlServer -o Models

if it is not work , try this :
dotnet tool install --global dotnet-ef --version 8.*

Pay Attension:
Replace "YourConnectionString" with your actual
database connection string, and Microsoft.EntityFrameworkCore.SqlServer with
the appropriate provider for your database (such as
Microsoft.EntityFrameworkCore.Sqlite for SQLite).

Entity Framework-Scaffold (more options)

Options you might include:

-o Models: Specifies the output folder for the generated models.

--context MyDbContext: Specifies the name for the generated DbContext.

-t Table1 -t Table2 (--table): Restricts the generated models to specific tables.

--schema schemaName: Specifies the schema if you want to limit scaffolding to a specific schema.

-f: Forces overwriting existing files.

Entity Framework-Model Create

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<DataSource>(entity =>
    {
        entity.HasKey(e => e.DsId);
```

- Basic Configuration for Properties

```
modelBuilder.Entity<DataSource>(entity =>
{
    // Configure property as required
    entity.Property(e => e.Name)
        .IsRequired()
        .HasMaxLength(100); // Max length constraint

    // Configure property with default value
    entity.Property(e => e.IsActive)
        .HasDefaultValue(true);

    // Configure property as a computed column
    entity.Property(e => e.CalculatedField)
        .HasComputedColumnSql("[SomeField] + [OtherField]");

    // Configure property for specific column type
    entity.Property(e => e.Description)
        .HasColumnType("nvarchar(500)");
});
```

Entity Framework-Model Create

- Configuring Primary Keys

```
modelBuilder.Entity<DataSource>(entity =>
{
    // Set primary key
    entity.HasKey(e => e.Id);

    // Composite primary key
    entity.HasKey(e => new { e.Part1, e.Part2 });
});
```

Entity Framework-Model Create

- Configuring Relationships

```
modelBuilder.Entity<DataSource>(entity =>
{
    // One-to-Many relationship
    entity.HasOne(e => e.ParentEntity)
        .WithMany(p => p.DataSources)
        .HasForeignKey(e => e.ParentEntityId)
        .OnDelete(DeleteBehavior.Cascade);

    // Many-to-Many relationship (EF Core 5.0+)
    entity.HasMany(e => e.Tags)
        .WithMany(t => t.DataSources)
        .UsingEntity<DataSourceTag>(
            j => j
                .HasOne(pt => pt.Tag)
                .WithMany(t => t.DataSourceTags)
                .HasForeignKey(pt => pt.TagId),
            j => j
                .HasOne(pt => pt.DataSource)
                .WithMany(d => d.DataSourceTags)
                .HasForeignKey(pt => pt.DataSourceId),
            j => j.HasKey(pt => new { pt.DataSourceId, pt.TagId })
        );
});
```

Entity Framework-Model Create

- Configuring Indexes

```
modelBuilder.Entity<DataSource>(entity =>
{
    // Create a unique index
    entity.HasIndex(e => e.Name)
        .IsUnique();

    // Create a non-unique index
    entity.HasIndex(e => e.CreatedDate);
});
```

Entity Framework-Model Create

Configuring Table and Column Names

```csharp
modelBuilder.Entity<DataSource>(entity =>
{
    // Configure table name
    entity.ToTable("DataSourcesTable");

    // Configure column name
    entity.Property(e => e.Name)
        .HasColumnName("DataSourceName");
});
```

Entity Framework -Migration -Code First

Each migration represents a change in your model, so you should add a migration whenever your model changes (such as when adding, modifying, or removing an entity or property).

1. create a migration:
   “dotnet ef migrations add InitialCreate”
2. Applying Migrations to the Database:
   “dotnet ef database update”

Entity Framework -Migration

Here are some additional useful commands for working with migrations

1. Remove the Last Migration (if it hasn't been applied)
   “dotnet ef migrations remove”
2. Update the Database to a Specific Migration
   “ dotnet ef database update MigrationName”
3. Checking Migration Status
   “dotnet ef migrations list”
4. Creating SQL Scripts for Migrations
   “dotnet ef migrations script”

Entity Framework -Migration

Automatic Migration on Application Startup (Optional)

For development, you might want to automatically apply migrations when the application starts. This is not recommended for production environments but can speed up development.

Add this code to Program.cs in ASP.NET Core:

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

using (var scope = app.Services.CreateScope())
{
    var dbContext = scope.ServiceProvider.GetRequiredService<DbContext>();
    dbContext.Database.Migrate();
}
```

ADO.NET

ADO.NET Core is a part of the .NET Core framework designed for data access. It provides a rich set of components for connecting to databases, executing commands, and managing data

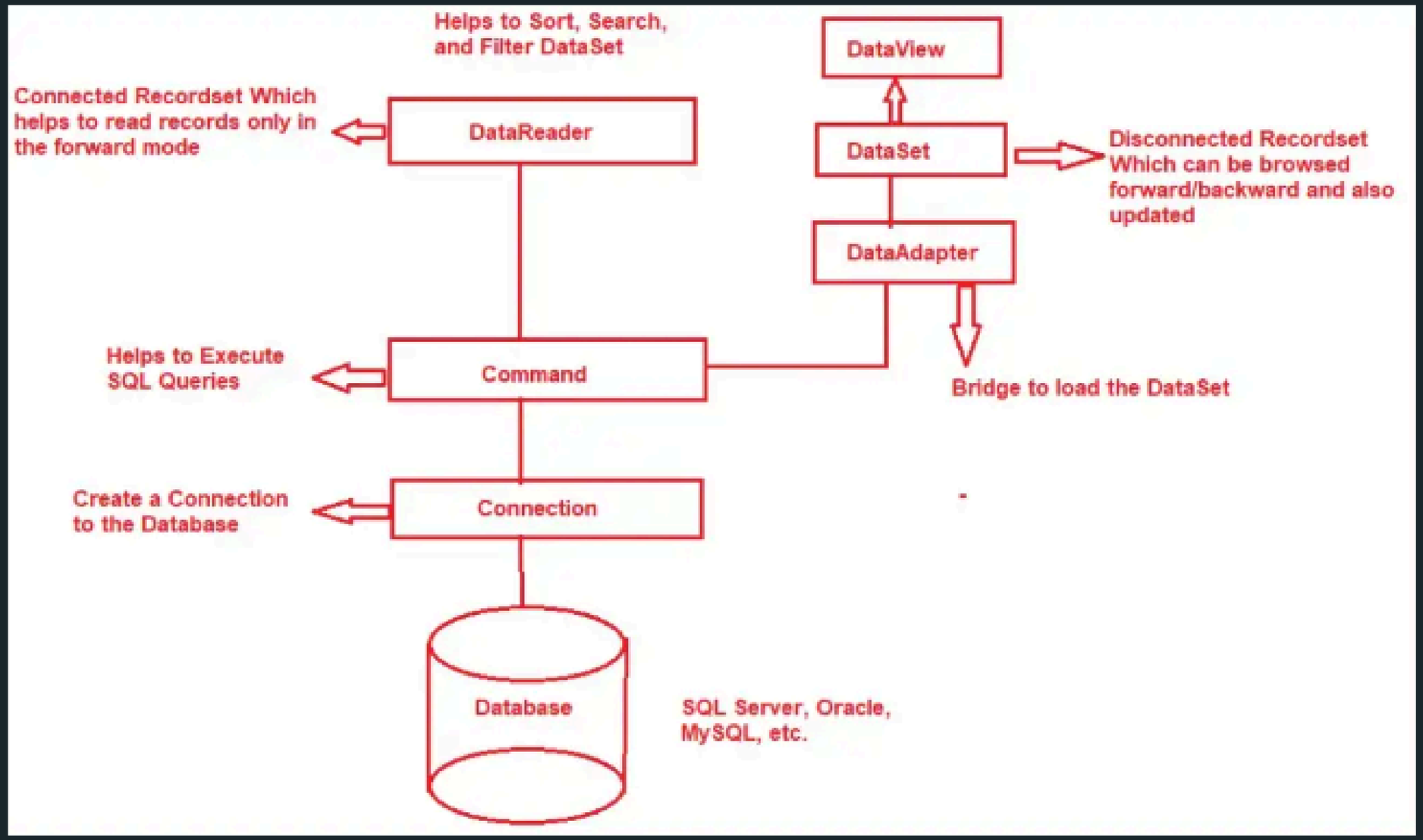
this is the package: System.Data.SqlClient;

ADO.NET

The ADO.NET Core Architecture is comprised of 6 important components. They are as follows:

- Connection
- Command
- DataReader
- DataAdapter
- DataSet
- DataView

ADO.NET


Helps to Sort, Search, and Filter DataSet
DataView
Connected Recordset Which helps to read records only in the forward mode
DataReader
DataSet
Disconnected Recordset Which can be browsed forward/backward and also updated
DataAdapter
Helps to Execute SQL Queries
Command
Bridge to load the DataSet
Create a Connection to the Database
Connection
Database
SQL Server, Oracle, MySQL, etc.

ADO.NET -Connection

The first important component of ADO.NET Core Architecture is the Connection Object.

The Connection component in ADO.NET Core is responsible for establishing a connection to a specific data source.

It’s the first step in interacting with a database. Each database provider (e.g., SQL Server, MySQL, PostgreSQL) has its own specific Connection class derived from the abstract DbConnection class.

This component handles all aspects of database connection, including opening, managing, and closing the connection.

ADO.NET -Connection

```
using (var conn = new SqlConnection(settings.Cnn))
{
    using (SqlCommand cmd = new SqlCommand(settings.ProcName, conn))
    {

        cmd.CommandType = CommandType.StoredProcedure;
        if (settings.Parameters != null)...
        conn.Open();
        var value = cmd.ExecuteScalar();
        if (value != null)
            return value.ToString();
```

ADO.NET -Command

The Second important component of ADO.NET Core Architecture is the Command Object.
The Command component is used to execute SQL queries, stored procedures, or commands on the database.
Derived from the DbCommand class, it allows you to interact with the database by sending commands and receiving results.
You can use it to perform CRUD (Create, Read, Update, Delete) operations.

ADO.NET -Command

Key properties and methods include:

- CommandText: The SQL query or stored procedure to be executed.
- CommandType: Indicates whether the CommandText is a text command, stored procedure, or table name.
- ExecuteReader(): Executes commands that return rows.
- ExecuteNonQuery(): Executes commands such as INSERT, DELETE, UPDATE, and SET statements.
- ExecuteScalar(): Executes commands that return a single value.

ADO.NET -Command

CommandText And Type

1.query

```
using (var conn = new SqlConnection(settings.Cnn))
{
    using (SqlCommand cmd = new SqlCommand("select * from books", conn))
    {

        cmd.CommandType = CommandType.Text;
```

2.procedure

```
using (var conn = new SqlConnection(settings.Cnn))
{
    using (SqlCommand cmd = new SqlCommand("procName", conn))
    {
        cmd.CommandType = CommandType.StoredProcedure;
```

ADO.NET -Command

ExecuteReader- Read Data From DataBase

```
using (SqlConnection connection = new SqlConnection(connectionString))
{
    try
    {
        connection.Open();
        Console.WriteLine("Connection to the database is successful!");

        string query = "SELECT Id, Title, Author FROM Books";
        using (SqlCommand command = new SqlCommand(query, connection))
        {
            using (SqlDataReader reader = command.ExecuteReader())
            {
                while (reader.Read())
                {
                    Console.WriteLine($"ID: {reader["Id"]}, Title: {reader
                }
            }
        }
    }
    catch (SqlException ex)
    {
        Console.WriteLine("An error occurred while connecting to the datab
        Console.WriteLin x.Message);
    }
```

ADO.NET -Command

ExecuteNonQuery-

ExecuteNonQuery is a method in ADO.NET used for executing SQL commands that don't return any data rows. Instead, it’s commonly used for operations like INSERT, UPDATE, DELETE, and CREATE TABLE, where the focus is on modifying the database rather than retrieving data. When ExecuteNonQuery is called, it returns an integer representing the number of rows affected by the command.

ADO.NET -Command

ExecuteNonQuery-Example

```
using (SqlConnection connection = new SqlConnection(connectionString))
{
    try
    {
        connection.Open();
        Console.WriteLine("Connection to the database is successful!");

        string insertQuery = "INSERT INTO Books (Title, Author, PublishedYear)

        using (SqlCommand command = new SqlCommand(insertQuery, connection))
        {
            // Adding parameters to avoid SQL injection
            command.Parameters.AddWithValue("@Title", "New Book Title");
            command.Parameters.AddWithValue("@Author", "Author Name");
            command.Parameters.AddWithValue("@PublishedYear", 2023);

            int rowsAffected = command.ExecuteNonQuery();
```

ADO.NET -Command

ExecuteScalar

ExecuteScalar is a method used when you want to execute a query that returns a single value

```
try                                                                    Copy code
{
    connection.Open();
    Console.WriteLine("Connection to the database is successful!");

    string countQuery = "SELECT COUNT(*) FROM Books";

    using (SqlCommand command = new SqlCommand(countQuery, connection))
    {
        // ExecuteScalar returns the result of the query as an object
        int bookCount = (int)command.ExecuteScalar();
        Console.WriteLine($"Total number of books: {bookCount}");
    }
}
catch (SqlException ex)
{
    Console.WriteLine("An error occurred while executing the command.");
    Console.WriteLine(ex.Message);
}
```

ADO.NET -DataReader

The third important component of ADO.NET Core Architecture is the DataReader Object.
The DataReader component is used to read data from a database in a forward-only, read-only manner.
It provides an efficient way to retrieve data using the ExecuteReader method of the Command object.
DbDataReader is the abstract class from which specific DataReader implementations derive.

ADO.NET -DataReader

```
var jsonResult = new StringBuilder();
var reader = cmd.ExecuteReader();
if (!reader.HasRows)
{
    jsonResult.Append("[]");
}
else
{
    while (reader.Read())
    {
        jsonResult.Append(reader.GetValue(0).ToString());
    }
}
return JArray.Parse(jsonResult.ToString());
```

```
using (SqlDataReader reader = command.ExecuteReader())
{
    // Check if the reader has rows
    if (reader.HasRows)
    {
        // Read each row
        while (reader.Read())
        {
            // Access each column by name or index
            int id = reader.GetInt32(0);  // GetInt32 expects the
            string title = reader.GetString(1);  // GetString for
            string author = reader.GetString(2);
            int publishedYear = reader.GetInt32(3);

            Console.WriteLine($"ID: {id}, Title: {title}, Author:
        }
    }
    else
    {
        Console.WriteLine("No rows found.");
    }
}
```

ADO.NET -DataAdapter

The fourth important component of ADO.NET Core Architecture is the DataAdapter Object.
A DataAdapter acts as a bridge between a DataSet (an in-memory representation of data) and a database
It uses Command objects to execute SQL commands at the data source to both load data into the DataSet and update the data source with changes made in the DataSet.

ADO.NET -DataAdapter

The DataAdapter uses four main commands:
SelectCommand – Retrieves data from the database.
InsertCommand – Adds new records to the database.
UpdateCommand – Updates existing records in the database.
DeleteCommand – Deletes records from the database.

ADO.NET -DataAdapter

```
DataTable dt = new DataTable();
try
{
    using (var conn = new SqlConnection(settings.Cnn))
    {
        using (SqlCommand cmd = new SqlCommand(settings.ProcName, conn))
        {
            cmd.CommandType = CommandType.StoredProcedure;
            if (settings.Parameters != null)
            {
                var parameters = JsonConvert.DeserializeObject<Dictionary<string, s
                foreach (var p in parameters)
                    cmd.Parameters.AddWithValue("@" + p.Key, p.Value);
            }

            using (SqlDataAdapter da = new SqlDataAdapter(cmd))
            {
                conn.Open();
                da.Fill (dt);
```

ADO.NET -DataSet

A DataSet is an in-memory representation of data,
often mirroring one or more tables in a relational database.
It’s a collection of DataTable objects that can hold multiple tables,
along with their relationships.
DataSet allows you to work with data in a disconnected mode
(without a continuous connection to the database).

ADO.NET -DataSet

```
using (SqlConnection connection = new SqlConnection(connectionString))
{
    connection.Open();

    // Fill DataSet with multiple tables
    SqlDataAdapter booksAdapter = new SqlDataAdapter("SELECT * FROM Books", c
    SqlDataAdapter authorsAdapter = new SqlDataAdapter("SELECT * FROM Authors"

    booksAdapter.Fill(dataSet, "Books");
    authorsAdapter.Fill(dataSet, "Authors");

    // Define a relationship between Books and Authors tables
    dataSet.Relations.Add("BookAuthorRelation",
        dataSet.Tables["Authors"].Columns["AuthorId"],
        dataSet.Tables["Books"].Columns["AuthorId"]);

    Console.WriteLine("DataSet loaded with Books and Authors tables.");
}
```

Convert DataTable to Entity Class

```
public static List<T> ConvertDataTableToListUsingJson<T>(DataTable dataTable)
{
    // Serialize DataTable to JSON
    string jsonString = JsonSerializer.Serialize(dataTable);

    // Deserialize JSON to a List of the specified type
    var modelList = JsonSerializer.Deserialize<List<T>>(jsonString);

    return modelList;
}
```

ADO.NET -SqlParameter

SqlParameter is used with SqlCommand objects to safely pass parameters into SQL queries. By using SqlParameter, you can prevent SQL injection attacks by parameterizing queries rather than directly embedding variables into the SQL command text

```
using (SqlConnection connection = new SqlConnection(connectionString))
{
    string query = "SELECT * FROM TasksUsers WHERE UserId = @UserId";
    using (SqlCommand command = new SqlCommand(query, connection))
    {
        // Define and add the parameter
        SqlParameter sqlParameter = new SqlParameter("@UserId", userId);
        command.Parameters.AddWithValue("@userID", userId);
        command.Parameters.Add(sqlParameter);

        connection.Open();
        using (SqlDataReader reader = command.ExecuteReader())
        {
            while (reader.Read())
            {
                // Access your data here
```

ADO.NET -Transaction

transactions are managed using the SqlTransaction class to ensure that a series of database operations either complete successfully as a unit or fail together.

This helps maintain data integrity in cases where multiple operations depend on each other.

```
// Start a local transaction
SqlTransaction transaction = connection.BeginTransaction();

try
{
    using (SqlCommand command1 = new SqlCommand("INSERT INTO Users (FirstName) " +
        "VALUES (@FirstName)", connection, transaction))
    {
        command1.Parameters.AddWithValue("@FirstName",FirstName);
        command1.ExecuteNonQuery();
    }

    using (SqlCommand command2 = new SqlCommand("INSERT INTO Tasks (Title) VALUES (@Title)", connection, transaction))
    {
        command2.Parameters.AddWithValue("@Title",Title);
        command2.ExecuteNonQuery();
    }

    transaction.Commit();
    Console.WriteLine("Transaction committed.");
}
catch (Exception ex)
{
    Console.WriteLine("Transaction failed: " + ex.Message);
    try
    {
        transaction.Rollback();
    }
```