DataBase server ‘Tier’

Definition:A layer responsible for access and managing databases. Database Management Systems (DBMS) based

Overview: is the layer responsible for storing and managing data within web applications. It acts as an intermediary between Application Tier and Data Tier, and is used to handle operations such as querying, storage, and updating, while ensuring security and efficiency in data management.

Importance:

- Isolate the layer containing the data from the end user
- Manage data
- reduce duplicate data
- An easy way to edit and manage data

Real-world use cases:

1) Store and manage product data, users, orders, and payments such as SheIn.
1) Patient Records and Medical Data Management.
2) Library systems.
3) Manage accounts, transactions, and bank transfers.

Implemented code snippets:

Connection with database :

```
{
  "ConnectionStrings": {
    "DefaultConnection":
"Server=localhost;Database=WebAppDb;Trusted_Connection=True;
MultipleActiveResultSets=true"
  }
}
```

Create a database context (DbContext):

```
public class ApplicationDbContext : DbContext
{
public
ApplicationDbContext(DbContextOptions<ApplicationDbContext>
options) : base(options) { } public DbSet<User> Users { get; set; }
}
```

Dependency Injection : it is a design pattren Used to manage dependencies between objects in an application.

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<ApplicationDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultC
onnection")));

var app = builder.Build();

app.Run();
```

Application server ‘Tier’

Definition: It is the layer that runs the programmatic logic of the application, separating the business from other layers such as the interface and database.

Overview: Application server is tier between Database tier and user tier, The application server processes the programming logic of the web application, including computations, business processes, validation, and data formatting between layers

Importance:

1. The application server separates the logic from the user interface and database, making the application more organized and easier to maintain.
2. Helps speed up processing by applying business logic to the server instead of the client.
3. Security: Application servers can enforce data security and protection policies.

Real-world use cases:

In an e-commerce platform, the Application Server manages complex business logic related to inventory management, order processing, pricing calculations, and user authentication.Example Amazon.

Social media platforms require Application Servers to manage millions of users, interactions (likes, comments, posts), and media content.

---

3 tier system architecture

1) DAL - Data Access Layer:

The data access layer is the layer responsible for connecting to the database or other data source. This layer performs CRUD operations on the data.

Role of the DAL layer:

- Performs query operations (SQL) or manipulates the database using ORMs.
- Keeps data access logic away from the rest of the app, helping improve maintenance.
- Provides data to the BLL layer and helps perform operations based on input.

Implemented code snippets:

```
public class StudentRepo : IStudentRepo
{
    private readonly ApplicationDbContext _context;

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

    public List<Student> GetAllStudents()
    {

        return _context.students.ToList();
    }
```

2) BLL - Business Logic Layer:

This layer is responsible for making decisions based on the input data and performing arithmetic or logical operations.

Role of the BLL layer:

1. Application-related business or operations such as price calculations, verification of input values, or inventory management are performed.
2. The BLL layer communicates with the DAL layer to obtain the data .

Implemented code snippets:

```
public class StudentService : IStudentService
{

    private readonly IStudentRepo _studentRepo;

    public StudentService(IStudentRepo studentRepo)
    {
        _studentRepo = studentRepo;
    }

    public List<Student> GetStudents()
    {
      return _studentRepo.GetAllStudents().OrderBy(s => s.Name).ToList();
}
```

3) Presentation Layer:

A layer that interacts directly with the user, displaying data and receiving input from users. This layer is responsible for rendering the user interface and interacting with users.

Role of the Presentation layer:

- Data Display: Presents the data received from the Business Logic layer to the user in an understandable way.
- User interaction: Deals with input from users such as buttons, forms, and interactions.
- Routing: Receives HTTP requests (in web applications) and routes them to the appropriate layers (such as the BLL).
- MVC Controllers are used to display and interact with the user data.

Implemented code snippets:

```
namespace WebApplication4.Controllers
{
   [ApiController]
   [Route("api / [Controller]")]
   public class StudentController:ControllerBase
   {

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

      [HttpGet("GetALL")]
      public List<Student> GetAll()
      {
        return _studentService.GetStudents().ToList();
```

Infrastructure Layer:

A layer that provides basic services and public functionality that supports other layers. These can include services such as sending email, handling log files, accessing external APIs, or connecting to cloud services.

Role of the Infrastructure layer:
Services such as authentication, validation of inputs, sending email, handling files, or interacting with external APIs include authentication.

Data transfer objects (DTO):

Used to transfer data between layers, such as transferring data between the Presentation Layer and the Business Logic Layer or between the API and the two users. The main purpose of DTO is to separate the internal structures of application data from the transmitted or received data.

Repository pattern in .net and generic repo:

Repository Pattern: It acts as an interface between the Business Logic Layer and the Data Access Layer.

Generic Repository: is a special application of Repository Pattern, where a generic Repository object is created that can handle multiple types of entities. This reduces the need to write redundant code for each entity.

The Unit of Work Pattern: is a design pattern used to manage operations that occur on data in a single context to ensure that they are implemented as a single unit.
The primary goal of this pattern is to ensure:

- Complete operations as a single unit: If all operations are successful, changes are saved, and if any operation fails, all changes are undone.
- Reduce database access: by performing operations in bulk instead of repeated calls.
- Format multiple Repositories: Work with multiple repositories within the same module.

Dependency inversion and dependency injection and object lifecycle:

Dependency Inversion Principle - DIP:

It is one of the design principles known as SOLID. The principle states that you should not rely directly on low-level modules (such as data access or databases). and should rely on abstractions rather than implementations.

```
public ProductService(IProductRepository repository)
{ _repository = repository; }
```

Dependency Injection: is a way to implement DIP.
It consists in passing dependencies to the object from the outside instead of creating them inside the object itself.

Types of Dependency Injection:

- Constructor Injection: Dependencies are passed through the constructor.
- Property Injection: Dependencies are set using properties.
- Method Injection: Dependencies are passed as coefficients for roads.

Object lifecycle: When using a Dependency Injection Container, the lifecycle of objects can be determined when they are registered.

Life cycle types:

Transient:A new object is created each time it is called,
Usage: When a stand-alone object is needed each time.

Scoped:A new object is created for each request or scope,
Usage: For objects that need to share data during a single request only.

Singleton:Only one object is created that is shared across the entire application.
Usage: For objects that need to be static for as long as the application runs.

Middleware in .NET and Middleware Pipeline:is a component that handles HTTP requests and responses in a pipeline. Middleware can inspect, modify, or short-circuit the request/response process.

Key Middleware Components:

- UseRouting(): Defines routing logic that determines how an incoming request is mapped to the appropriate route handler.
- UseEndpoints(): Allows you to configure endpoints for routing, such as controllers and Razor pages.
- UseAuthentication(): Adds authentication middleware to the pipeline.

- UseAuthorization(): Ensures that only authorized users can access specific endpoints.
- UseCors(): Configures cross-origin requests, allowing resources to be requested from different origins.
- UseStaticFiles(): Serves static files such as HTML, CSS, and JavaScript from the wwwroot folder.

App setting .json: It is a configuration file used in .NET applications to store various settings such as:

- Database settings.
- Logging settings.
- Other configuration keys (API Keys, environmental variables)

Implemented code snippets:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },

  "ConnectionStrings": {
    "DefaultConnection": "Data Source=(local);Initial
Catalog=WebAPITest2;Integrated
Security=True;TrustServerCertificate=True"
  },


  "AllowedHosts": "*"
}
```

Rest Api and HTTP protocol

REST API:It is a type of API that follows the RESTful pattern, an architectural design that aims to build communication systems between clients and servers in a simple and efficient way using the HTTP protocol.

HTTP:It is the primary protocol that the REST API uses to transfer data between the client and the server.

Method HTTP:

1) Get: retrieve data
2) Put: Update data
3) Post:Create new
4) Delete:remove
5) Patch:Partial update of an existing resource

Request in HTTP:

1) Methods
2) Url
3) Headers=> Extra info
4) Body => Data sent with request

Response in HTTP:

1) status code
2) Headers=> Extra info
3) body=> Data sent from server

What is Swagger?

Swagger is a framework used to document application programming interfaces (APIs) in a structured and readable way for humans and machines.

Ingredients for Swagger:

OpenAPI Specification:
An open standard for determining the REST API format.
Swagger UI:
Interactive graphical interface for viewing and documenting API.
Swagger Generator:
A tool to automatically generate documentation from application code.

```
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.
```

```
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

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

API Versioning:
It is the process of managing different versions of an application programming interface (API) so that developers can maintain compatibility with older versions while introducing new features in future releases.

Why do we use API Versioning?

- Backward compatibility:
  When an API is updated, some users may need to continue using the old version.

- Manage updates seamlessly:
  Allows new features to be introduced in specific versions without impacting existing users.

- Separation of features or data changes:
  Different versions can be customized to handle different requirements or data changes.

Routing in .net

Routing is the process of matching incoming HTTP requests to an application with appropriate actions within the Controller or with endpoints.

Importance of Routing

- Determine the destination of requests:
  Easily directs incoming requests to appropriate actions based on the address.

- Manage dynamic routes:
  Allows handling of dynamic data within routes, such as resource identifiers (id).

- Organize the application:
  Enhance the organization of the application by separating routes and clarifying the structure of requests.

- Flexibility:
  Supports different methods of specifying routes to suit the nature of the project.

Types of Routing in .NET

1) Conventional Routing:
   - It is based on a pattern defined in the configuration file.
   - It is often used in MVC applications.

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

2) Attribute Routing:
   - Paths are defined using attributes added directly to actions.
   - It is widely used in RESTful APIs.

Feature	Conventional Routing	Attribute Routing
Definition	**In configuration files like**	**Directly within controllers and**
Location	**`Program.cs`.**	**actions.**
Flexibility	**Limited and suitable for general routes.**	**Flexible and ideal for custom routes.**

What is Model Binding?

Model Binding is a feature in .NET that converts data received from an HTTP request (such as form data, query strings, parameters in a path, or JSON data) into strongly typed objects that can be used within an application.

Model Binding:

Supported data sources:

- Query Strings
- Route Parameters
- Body of requests (usually in JSON or XML)
- Form Data
- Headers
- Cookies

How it works:

- The framework searches for a key name in the specified source.
- If the key is found, the value is converted to the requested data type (String, Int, Custom Object).
- If the key cannot be matched or the value cannot be converted, the parameter is assigned a default value.

What is Validation?

Validation is the process of checking the validity of data received by an application. Its purpose is to ensure that the values provided meet specified rules before they are used within the application or stored.

How does Validation work?

- Validation rules are defined within models using attributes.
- The framework automatically runs validation after the Model Binding process.
- If the rules are not met, errors are added to the ModelState.
- Attributes used in Validation

Common attributes:

- [Required]: Ensures that the field is not empty.
- [StringLength]: Specifies the minimum and maximum length of the text.
- [Range]: Specifies an acceptable numeric range.
- [EmailAddress]: Validates the validity of the email address.
- [RegularExpression]: Checks that the text matches a given pattern.