.NET Full Stack Interview Guide

Doc version – 1.0

Table of content:

Introduction:

IntegerByte has become a prominent name in the community of software programmers and web developers. We have provided our developers with a platform where they share knowledge and achieve certain goals.

With collaboration with StackBytes, we are helping engineers including freshers and experience to crack interviews providing basic interview questions, some tips and tricks to crack interview, best practices to follow.

Our Services:

- **Website Design and Development and Mobile Applications (Android and IOS).**
- **Domain name, Web Hosting and Security Provider (SSL Certificate).**
- **Logo, Banner and Brochure Designing.**

Our website - www.integerbyte.com

Our Blog - www.integerbyteblog.in

Reach out to us on - tointegerbyte@gmail.com OR stackbytes08@gmail.com

Follow our social media for more updates:

1. What is .NET & .NET Framework?

- .NET Framework**: A software framework by Microsoft for developing and running applications in a controlled environment.**
- **.NET**: A developer platform to build applications for web, mobile, desktop, and IoT, supporting multiple languages like C#, F#, and C++.

---

2. Explain Common Language Runtime (CLR):

- CLR is the runtime environment that manages the execution of .NET programs, providing services like memory management, security, and exception handling.

---

3. What is the Global Assembly Cache (GAC)?

- GAC stores DLLs globally to avoid conflicts and make them accessible across multiple applications.

---

4. What is Garbage Collection?

- Garbage Collection automatically manages memory by allocating and deallocating it. It frees memory once an object is no longer needed.

  Types:

  - Generation 0: Contains the youngest objects.
  - Generation 1: Objects not reclaimed in Gen 0 move here.
  - Generation 2: Objects that persist for a long time.

---

5. What is Value Type and Reference Type?

- **Value Type**: Stored in the stack, directly holds the value.
  Example: `int i = 10;`
- **Reference Type**: Stored in the heap, holds a reference to the memory address.
  Example: `string s = "hello world";`

---

6. Difference Between String, StringBuffer, and StringBuilder?

- **String**: Immutable and fixed length. Modifications create new instances.
- **StringBuilder**: Mutable with variable length, allows modifications without creating new instances.
- **StringBuffer**: Mutable, thread-safe (synchronized), slower than StringBuilder but faster than String.

---

7. What is Boxing and Unboxing?

- **Boxing**: Converts a value type to a reference type.
  Example: `int i = 10; object o = i;`
- **Unboxing**: Converts a reference type back to a value type.
  Example: `object o = 12; int i = (int)o;`

---

8. What is the Difference Between .NET and .NET Core?

- **.NET**: Supports Windows-based applications and is OS-dependent.
- **.NET Core**: A cross-platform, open-source framework for modern application development.

---

9. What is the Difference Between Constant and Read-Only?

- **Constant**: Value is fixed at compile time and cannot change. It is implicitly static.
- **Read-Only**: Value is assigned at runtime and can only be changed in the constructor.

---

10. What is the Difference Between Authentication and Authorization?

- **Authentication**: Verifies the user's identity and ensures they exist in the system.
- **Authorization**: Validates what the authenticated user is allowed to access based on roles or permissions.

SOLID Principles

SOLID principles are a popular set of design guidelines that help developers write readable, reusable, maintainable, and scalable code. These principles ensure the code is modular, easier to understand, and less prone to bugs when changes are introduced.

Single Responsibility Principle (SRP)

- A class should have only one reason to change.
- This means each class should focus on a single responsibility, ensuring clarity and reducing the risk of unintended side effects when modifications are made.

Let's look at the example

```
// Class for creating an order
public class OrderCreator
{
    public void CreateOrder()
    {
        Console.WriteLine("Order created.");
    }
}

// Class for calculating the order total
public class OrderCalculator
{
    public void CalculateOrderTotal()
    {
        Console.WriteLine("Order total calculated.");
    }
}
class Program
{
    public static void Main(string[] args)
    {
        var orderCreator = new OrderCreator();
        var orderCalculator = new OrderCalculator();
        orderCreator.CreateOrder();
        orderCalculator.CalculateOrderTotal();
    }
}
```

Open-Closed Principle (OCP)

- Classes should be open for extension but closed for modification.

- This allows adding new functionality through method overriding in derived classes without altering the base class.
- It promotes stability and scalability in object-oriented design.

Let's look at the example

```
// Base class
public abstract class Order
{
    public abstract void ProcessOrder();
}
// Derived class for online orders
public class OnlineOrder : Order
{
    public override void ProcessOrder()
    {
        Console.WriteLine("Processing online order.");
    }
}
// Derived class for in-store orders
public class InStoreOrder : Order
{
    public override void ProcessOrder()
    {
        Console.WriteLine("Processing in-store order.");
    }
}
// Order processor that uses polymorphism
public class OrderProcessor
{
    public void ProcessOrder(Order order)
    {
        order.ProcessOrder();
    }
}
// Usage
public class Program
{
    public static void Main(string[] args)
    {
        OrderProcessor processor = new OrderProcessor();
        Order onlineOrder = new OnlineOrder();
        Order inStoreOrder = new InStoreOrder();
        processor.ProcessOrder(onlineOrder);  // Output: Processing online order.
        processor.ProcessOrder(inStoreOrder); // Output: Processing in-store order.
    }
}
```

Liskov Substitution Principle (LSP)

- Objects of a derived class should be able to replace objects of the base class without altering the correctness of the program.
- Method overriding is a common implementation mechanism used in both the Open-Closed Principle (OCP) and the Liskov Substitution Principle (LSP), but their purposes and focuses differ.

- OCP emphasizes extensibility and LSP emphasizes compatibility and substitutability between base and derived classes.

Let’s look at the example

```
public class Order
{
    public virtual void ProcessOrder()
    {
        Console.WriteLine("Processing a general order.");
    }
}

// Derived class
public class OnlineOrder : Order
{
    public override void ProcessOrder()
    {
        Console.WriteLine("Processing an online order.");
    }
}

// Another derived class
public class InStoreOrder : Order
{
    public override void ProcessOrder()
    {
        Console.WriteLine("Processing an in-store order.");
    }
}
class Program
{
    public static void Main(string[] args)
    {
        // Base class reference, substituting derived classes
        Order onlineOrder = new OnlineOrder();
        Order inStoreOrder = new InStoreOrder();

        // Both should work correctly without altering the program's behavior
        onlineOrder.ProcessOrder(); // Output: Processing an online order.
        inStoreOrder.ProcessOrder(); // Output: Processing an in-store order.
    }
}
```

Interface Segregation Principle (ISP)

- The Interface Segregation Principle states that a client should not be forced to implement interfaces it does not use.

Let’s look at the example

```
public interface IOrderCreation
{
    void CreateOrder();
}
public interface IInvoiceGeneration
{
    void GenerateInvoice();
}
public class OnlineOrder : IOrderCreation, IInvoiceGeneration
{
    public void CreateOrder()
    {
        Console.WriteLine("Online order created.");
    }
    public void GenerateInvoice()
    {
        Console.WriteLine("Invoice generated for online order.");
    }
}
public class InStoreOrder : IOrderCreation
{
    public void CreateOrder()
    {
        Console.WriteLine("In-store order created.");
    }
}
// Usage
public class Program
{
    public static void Main(string[] args)
    {
        IOrderCreation onlineOrder = new OnlineOrder();
        onlineOrder.CreateOrder();
        IOrderCreation inStoreOrder = new InStoreOrder();
        inStoreOrder.CreateOrder();
    }
}
```

Dependency Inversion Principle (DIP)

- High-level modules should not depend on low-level modules; both should depend on abstractions.
- This decouples the system and makes it more flexible, maintainable, and testable.

```
// Abstraction
public interface INotifier
{
    void Notify(string message);
}
// Low-level module
public class EmailNotifier : INotifier
{
    public void Notify(string message)
    {
        Console.WriteLine($"Email sent: {message}");
    }
}
// Another low-level module
public class SmsNotifier : INotifier
{
    public void Notify(string message)
    {
        Console.WriteLine($"SMS sent: {message}");
    }
}
// High-level module
public class OrderProcessor
{
    private readonly INotifier _notifier;

    public OrderProcessor(INotifier notifier)
    {
        _notifier = notifier;
    }
    public void ProcessOrder()
    {
        Console.WriteLine("Order processed.");
        _notifier.Notify("Order confirmation.");
    }
}
public class Program
{
    public static void Main(string[] args)
    {
        // Inject EmailNotifier
        INotifier emailNotifier = new EmailNotifier();
        var emailOrderProcessor = new OrderProcessor(emailNotifier);
        emailOrderProcessor.ProcessOrder();
        // Inject SmsNotifier
        INotifier smsNotifier = new SmsNotifier();
        var smsOrderProcessor = new OrderProcessor(smsNotifier);
        smsOrderProcessor.ProcessOrder();
    }
}
```

Design Patterns

Design patterns are reusable and proven solutions to common software design problems. They help developers create code that is **maintainable**, **scalable**, and **easy to understand**.

Types of Design Patterns:

- **Creational Patterns**: Deal with object creation.
  Examples: Singleton, Factory, Abstract Factory
- **Structural Patterns**: Focus on the composition of classes and objects.
  Examples: Adapter, Facade, Repository
- **Behavioral Patterns**: Concerned with communication between objects.
  Examples: Strategy, Iterator, Unit of Work, CQRS
  (*CQRS is an architectural pattern combining structural and behavioral aspects.*)

Singleton Pattern

- The Singleton Design Pattern ensures that a class has only **one instance** and provides a **global point of access** to that instance.
- It is particularly useful for scenarios where a **single shared resource** is required, such as:
  - Logging
  - Caching
  - Configuration settings
  - Database connections

Let's look at the example

```
public class Logger
{
    // Private static instance of the class
    private static Logger _instance;
    // Object for thread-safety
    private static readonly object _lock = new object();
    // Private constructor to prevent instantiation
    private Logger() { }
    // Public static method to provide a single global instance
    public static Logger Instance
    {
        get
        {
            lock (_lock) // Ensures thread safety
            {
                if (_instance == null)
                {
                    _instance = new Logger();
                }
                return _instance;
            }
        }
    }
    // Method to log messages
    public void Log(string message)
    {
        // Example: Write log to a file or console
        Console.WriteLine($"[{DateTime.Now}] {message}");
    }
}
// Example usage in Order Management System
public class OrderManagement
{
    public void CreateOrder(int orderId)
    {
        // Log order creation
        Logger.Instance.Log($"Order {orderId} has been created.");
    }

    public void UpdateOrder(int orderId)
    {
        // Log order update
        Logger.Instance.Log($"Order {orderId} has been updated.");
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        // Create OrderManagement instance
        OrderManagement orderManagement = new OrderManagement();
        // Perform operations
        orderManagement.CreateOrder(101);  // Logs: Order 101 has been created.
        orderManagement.UpdateOrder(101); // Logs: Order 101 has been updated.
    }
}
```

Factory Design Pattern

- The Factory Design Pattern provides a way to **create objects without specifying the exact class** of the object that will be created.
- It is particularly useful when we have **multiple classes sharing a common interface** but requiring **different implementations**.

Let's look at the example

Scenario

In an **Order Management System**, consider different types of orders:

- Online Orders
- Store Orders
- Bulk Orders

Each type of order:

- Shares a **common interface**
- Has a **different implementation**, such as:
    - Processing fees
    - Shipping methods
    - Notifications

The **Factory Pattern** can be used to **create instances of these order types dynamically** based on the input or context.

```
// Step 1: Define a common interface
public interface IOrder
{
    void ProcessOrder(); // Common method for all orders
}
// Step 2: Implement concrete classes for each order type
public class OnlineOrder : IOrder
{
    public void ProcessOrder()
    {
        Console.WriteLine("Processing Online Order with shipping.");
    }
}
public class StoreOrder : IOrder
{
    public void ProcessOrder()
    {
        Console.WriteLine("Processing Store Order with in-store pickup.");
    }
}
public class BulkOrder : IOrder
{
    public void ProcessOrder()
    {
        Console.WriteLine("Processing Bulk Order with discounted rates.");
    }
}
// Step 3: Create the Factory class
public class OrderFactory
{
    public static IOrder CreateOrder(string orderType)
    {
        return orderType switch
        {
            "Online" => new OnlineOrder(),
            "Store" => new StoreOrder(),
            "Bulk" => new BulkOrder(),
            _ => throw new ArgumentException("Invalid order type")
        };
    }
}
```

```
// Step 4: Use the factory in the client code
class Program
{
    static void Main(string[] args)
    {
        // Example: Dynamically create order types
        IOrder onlineOrder = OrderFactory.CreateOrder("Online");
        onlineOrder.ProcessOrder(); // Output: Processing Online Order with shipping.

        IOrder storeOrder = OrderFactory.CreateOrder("Store");
        storeOrder.ProcessOrder(); // Output: Processing Store Order with in-store pickup.

        IOrder bulkOrder = OrderFactory.CreateOrder("Bulk");
        bulkOrder.ProcessOrder(); // Output: Processing Bulk Order with discounted rates.
    }
}
```

Repository Pattern

- The Repository Pattern **abstracts the data access logic**, acting as a **mediator** between the domain/business logic and the data source.
- It provides a **consistent interface** to perform CRUD operations (Create, Read, Update, Delete).
- This allows you to **switch between SQL, NoSQL, or any data store** without affecting your application's business logic.
- It is particularly useful in:

- **Domain-Driven Design (DDD)**
- **RESTful APIs**

Let's look at the example

Scenario

In an **Order Management System (OMS)**, we manage orders stored in a database.

Using the **Repository Pattern**:

- We can **abstract the logic** for accessing the database.
- This ensures that the **business logic** (such as creating, fetching, or updating orders) **doesn't depend on raw database queries or specific technologies.**

```
public class Order
{
    public int OrderId { get; set; }
    public string CustomerName { get; set; }
    public DateTime OrderDate { get; set; }
    public decimal TotalAmount { get; set; }
}
public interface IOrderRepository
{
    // CRUD operations
    IEnumerable<Order> GetAllOrders();
    Order GetOrderById(int orderId);
    void AddOrder(Order order);
    void UpdateOrder(Order order);
    void DeleteOrder(int orderId);
}
```

```
public class OrderRepository : IOrderRepository
{
    // Simulated in-memory data store (can be replaced with actual database access)
    private readonly List<Order> _orders = new List<Order>();

    public IEnumerable<Order> GetAllOrders()
    {
        return _orders;
    }

    public Order GetOrderById(int orderId)
    {
        return _orders.FirstOrDefault(o => o.OrderId == orderId);
    }

    public void AddOrder(Order order)
    {
        _orders.Add(order);
    }

    public void UpdateOrder(Order order)
    {
        var existingOrder = _orders.FirstOrDefault(o => o.OrderId == order.OrderId);
        if (existingOrder != null)
        {
            existingOrder.CustomerName = order.CustomerName;
            existingOrder.OrderDate = order.OrderDate;
            existingOrder.TotalAmount = order.TotalAmount;
        }
    }

    public void DeleteOrder(int orderId)
    {
        var order = _orders.FirstOrDefault(o => o.OrderId == orderId);
        if (order != null)
        {
            _orders.Remove(order);
        }
    }
}
```

```
public class OrderService
{
    private readonly IOrderRepository _orderRepository;

    public OrderService(IOrderRepository orderRepository)
    {
        _orderRepository = orderRepository;
    }

    public void CreateNewOrder(Order order)
    {
        _orderRepository.AddOrder(order);
        Console.WriteLine($"Order {order.OrderId} created successfully.");
    }

    public void DisplayAllOrders()
    {
        var orders = _orderRepository.GetAllOrders();
        foreach (var order in orders)
        {
            Console.WriteLine($"Order ID: {order.OrderId}, Customer: {order.CustomerName}, Total: {order.TotalAmount}");
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        // Initialize repository and service
        IOrderRepository orderRepository = new OrderRepository();
        OrderService orderService = new OrderService(orderRepository);

        // Create and display orders
        orderService.CreateNewOrder(new Order
        {
            OrderId = 1,
            CustomerName = "Stack Bytes",
            OrderDate = DateTime.Now,
            TotalAmount = 100.50m
        });
        orderService.CreateNewOrder(new Order
        {
            OrderId = 2,
            CustomerName = "IntegerByte",
            OrderDate = DateTime.Now,
            TotalAmount = 250.75m
        });

        Console.WriteLine("All Orders:");
        orderService.DisplayAllOrders();
    }
}
```

Unit of Work Pattern

- The **Unit of Work** pattern helps manage **database transactions** in a consistent way.
- It groups multiple operations — such as **insert**, **update**, and **delete** — into a **single unit of work**.
- If any operation fails, the **entire transaction is rolled back**, ensuring:
    - **Data integrity**
    - **Easier testing**
    - **Simpler maintenance**

Let’s look at the example

Scenario

In an **Order Management System (OMS)**, when a user places an order, the following steps are required:

1. Create a new order record
2. Deduct the stock quantity of the ordered items
3. Log the transaction in the audit table

All of these actions:

- Must either **succeed together**
- Or **fail as a group**

By using the **Unit of Work pattern**, we ensure that all related changes are committed in **one transaction**, maintaining consistency across the system.

```
public class Order
{
    public int Id { get; set; }
    public string CustomerName { get; set; }
    public decimal TotalAmount { get; set; }
    public DateTime OrderDate { get; set; } = DateTime.Now;
}
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int StockQuantity { get; set; }
    public decimal Price { get; set; }
}
public class TransactionLog
{
    public int Id { get; set; }
    public int OrderId { get; set; }
    public string Action { get; set; }
    public DateTime Timestamp { get; set; }
}
public class OMSDbContext : DbContext
{
    public OMSDbContext(DbContextOptions<OMSDbContext> options) : base(options) { }

    // DbSet for each entity
    public DbSet<Order> Orders { get; set; }
    public DbSet<Product> Products { get; set; }
    public DbSet<TransactionLog> TransactionLogs { get; set; }
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        // Fluent API configurations if needed (optional)
        base.OnModelCreating(modelBuilder);
    }
}
public interface IRepository<T> where T : class
{
    void Add(T entity);
    void Update(T entity);
    void Remove(T entity);
    T GetById(int id);
    IEnumerable<T> GetAll();
}
```

```
public class Repository<T> : IRepository<T> where T : class
{
    private readonly DbSet<T> _dbSet;

    public Repository(DbContext context)
    {
        _dbSet = context.Set<T>();
    }

    public void Add(T entity) => _dbSet.Add(entity);
    public void Update(T entity) => _dbSet.Update(entity);
    public void Remove(T entity) => _dbSet.Remove(entity);
    public T GetById(int id) => _dbSet.Find(id);
    public IEnumerable<T> GetAll() => _dbSet.ToList();
}
public interface IUnitOfWork : IDisposable
{
    IRepository<Order> Orders { get; }
    IRepository<Product> Products { get; }
    IRepository<TransactionLog> Logs { get; }
    void Save();
}

public class UnitOfWork : IUnitOfWork
{
    private readonly OMSDbContext _context;

    public IRepository<Order> Orders { get; private set; }
    public IRepository<Product> Products { get; private set; }
    public IRepository<TransactionLog> Logs { get; private set; }

    public UnitOfWork(OMSDbContext context)
    {
        _context = context;
        Orders = new Repository<Order>(_context);
        Products = new Repository<Product>(_context);
        Logs = new Repository<TransactionLog>(_context);
    }

    public void Save() => _context.SaveChanges();
    public void Dispose() => _context.Dispose();
}
```

```
class Program
{
    static void Main(string[] args)
    {
        var options = new DbContextOptionsBuilder<OMSDbContext>()
            .UseInMemoryDatabase("OMSDatabase")
            .Options;

        using (var context = new OMSDbContext(options))
        using (var unitOfWork = new UnitOfWork(context))
        {
            var orderService = new OrderService(unitOfWork);

            // Seed products
            var products = new List<Product>
    {
        new Product { Name = "Pen", StockQuantity = 10, Price = 1000 },
        new Product { Name = "Books", StockQuantity = 50, Price = 20 }
    };
            foreach (var product in products)
                unitOfWork.Products.Add(product);
            unitOfWork.Save();

            // Create order
            var order = new Order
            {
                CustomerName = "Stack Bytes",
                TotalAmount = 1001
            };

            try
            {
                orderService.PlaceOrder(order, products.Take(2).ToList());
                Console.WriteLine("Order placed successfully!");
            }
            catch (Exception ex)
            {
                Console.WriteLine($"Error: {ex.Message}");
            }
        }
    }
}
```

CQRS Pattern (Command Query Responsibility Segregation)

- The **CQRS** pattern is a design approach that **separates read and write operations** into distinct models.
- These are often implemented in **different classes or interfaces**:
  - **Command model** → For writing data (Create, Update, Delete)
  - **Query model** → For reading data (Read-only operations)
- This separation results in:
  - **Better scalability**
  - **Improved maintainability**
  - **Optimized performance**

Let's look at the example

```
//write model
public class PlaceOrderCommand
{
    public int CustomerId { get; set; }
    public List<int> ProductIds { get; set; }
    public decimal TotalAmount { get; set; }
}
//read model
public class OrderDetailsDto
{
    public int OrderId { get; set; }
    public string CustomerName { get; set; }
    public List<string> ProductNames { get; set; }
    public decimal TotalAmount { get; set; }
    public DateTime OrderDate { get; set; }
}
public class PlaceOrderHandler
{
    private readonly OMSDbContext _context;

    public PlaceOrderHandler(OMSDbContext context)
    {
        _context = context;
    }

    public void Handle(PlaceOrderCommand command)
    {
        // Create new order
        var order = new Order
        {
            CustomerName = _context.Customers.Find(command.CustomerId).Name,
            TotalAmount = command.TotalAmount,
            OrderDate = DateTime.Now
        };
        _context.Orders.Add(order);

        // Reduce stock for each product
        foreach (var productId in command.ProductIds)
        {
            var product = _context.Products.Find(productId);
            if (product.StockQuantity <= 0)
                throw new Exception($"Product {product.Name} is out of stock.");

            product.StockQuantity--;
            _context.Products.Update(product);
        }

        // Save changes
        _context.SaveChanges();
    }
}
```

```
public class GetOrderDetailsHandler
{
    private readonly OMSDbContext _context;

    public GetOrderDetailsHandler(OMSDbContext context)
    {
        _context = context;
    }
    public OrderDetailsDto Handle(int orderId)
    {
        var order = _context.Orders.Find(orderId);

        return new OrderDetailsDto
        {
            OrderId = order.Id,
            CustomerName = order.CustomerName,
            ProductNames = _context.OrderProducts
                .Where(op => op.OrderId == orderId)
                .Select(op => op.Product.Name)
                .ToList(),
            TotalAmount = order.TotalAmount,
            OrderDate = order.OrderDate
        };
    }
}
class Program
{
    static void Main(string[] args)
    {
        var options = new DbContextOptionsBuilder<OMSDbContext>()
            .UseInMemoryDatabase("OMSDatabase")
            .Options;
        using (var context = new OMSDbContext(options))
        {
            // Seed database
            context.Products.Add(new Product { Name = "Pen", StockQuantity = 10, Price = 1000 });
            context.Products.Add(new Product { Name = "Books", StockQuantity = 20, Price = 50 });
            context.SaveChanges();
            // Command: Place Order
            var commandHandler = new PlaceOrderHandler(context);
            commandHandler.Handle(new PlaceOrderCommand
            {
                CustomerId = 1,
                ProductIds = new List<int> { 1, 2 },
                TotalAmount = 1001
            });
            // Query: Get Order Details
            var queryHandler = new GetOrderDetailsHandler(context);
            var orderDetails = queryHandler.Handle(1);
            Console.WriteLine($"Order ID: {orderDetails.OrderId}");
            Console.WriteLine($"Customer Name: {orderDetails.CustomerName}");
            Console.WriteLine($"Products: {string.Join(", ", orderDetails.ProductNames)}");
            Console.WriteLine($"Total Amount: {orderDetails.TotalAmount}");
            Console.WriteLine($"Order Date: {orderDetails.OrderDate}");
        }
    }
}
```

.NET Core Interview Questions

---

1. What is DotNetCore and how it differs from DotNet?

- .NET Core is a **cross-platform**, **open-source** framework for building modern applications.
- .NET Framework is **Windows-only** and platform-dependent.
- .NET Core is:
    - Faster
    - Supports microservices
    - Now unified under **.NET 5+** as just ".NET"

---

2. What is Dependency Injection and explain types?

- **Dependency Injection (DI)** is a design pattern to manage object dependencies.

Types in .NET Core:

- **Constructor Injection**: Dependencies are provided through a class's constructor.
- **Property Injection**: Dependencies are injected into public properties after instantiation.
- **Method Injection**: Dependencies are passed as method parameters at runtime.

Let's look at the example

```
// Service to handle photo uploads
public class PhotoService
{
    public void UploadPhoto(string photo)
    {
        Console.WriteLine($"Uploading photo: {photo}");
    }
}

// InstagramUser depends on PhotoService
public class InstagramUser
{
    private readonly PhotoService _photoService;

    // Constructor Injection
    public InstagramUser(PhotoService photoService)
    {
        _photoService = photoService;
    }

    public void PostPhoto(string photo)
    {
        _photoService.UploadPhoto(photo);
    }
}
```

```
// Service to handle video uploads
public class VideoService
{
    public void UploadVideo(string video)
    {
        Console.WriteLine($"Uploading video: {video}");
    }
}

// InstagramUser depends on VideoService
public class InstagramUser
{
    // Property Injection
    public VideoService VideoService { get; set; }

    public void PostVideo(string video)
    {
        VideoService?.UploadVideo(video); // Use the injected VideoService
    }
}

// Usage
class Program
{
    static void Main(string[] args)
    {
        InstagramUser user = new InstagramUser();
        user.VideoService = new VideoService(); // Inject dependency via property
        user.PostVideo("TereBina.mp4");
    }
}
```

```
// Service to handle story uploads
public class StoryService
{
    public void UploadStory(string story)
    {
        Console.WriteLine($"Uploading story: {story}");
    }
}

// InstagramUser posts stories using a method that receives the service
public class InstagramUser
{
    // Method Injection
    public void PostStory(StoryService storyService, string story)
    {
        storyService.UploadStory(story); // Dependency passed as method parameter
    }
}

// Usage
class Program
{
    static void Main(string[] args)
    {
        StoryService storyService = new StoryService();
        InstagramUser user = new InstagramUser();
        user.PostStory(storyService, "MyPic.jpg"); // Inject dependency via method
    }
}
```

3. What are the different service lifetime options available in the Dependency Injection (DI) container?

- **AddSingleton**: Creates one instance for the application's lifetime.
  Example: Logging service
- **AddScoped**: Creates one instance per HTTP request.
  Example: `DbContext` in APIs
- **AddTransient**: Creates a new instance every time it's requested.
  Example: Lightweight utility services

Code Examples:

```
csharp
CopyEdit
services.AddSingleton<ILogger, Logger>();           // Shared logger service
services.AddScoped<IDbContext, AppDbContext>();      // Scoped to each API request
services.AddTransient<IUtilityService, UtilityService>(); // Created on each call
```

---

4. What is middleware in .NET Core and how to create custom middleware?

- Middleware processes **HTTP requests and responses** in the request pipeline.
- It can:
  - Intercept requests
  - Run logic
  - Pass to the next middleware
- You can create custom middleware using `RequestDelegate`.

RequestDelegate: A function that handles HTTP requests and optionally forwards to the next middleware using `await`.

---

5. How to use filters in .NET Core?

- Filters in .NET Core allow you to execute logic **before or after** controller actions.

Types of Filters:

- Authorization Filter
- Action Filter
- Exception Filter
- Result Filter
- Resource Filter

6. What is Kestrel Server and how it differs from IIS?

- **Kestrel**:
  - A lightweight, cross-platform web server built into .NET Core.
  - Handles **actual request processing**.
- **IIS/Nginx**:
  - Public-facing web servers (often reverse proxies).
  - IIS is **Windows-specific**, while Kestrel is **cross-platform**.
  - Typically, Kestrel is used behind IIS or Nginx for production deployments.

7. Explain the role of Docker and Kubernetes in ASP.NET Core deployment

- **Docker**:
  - Containerizes your app so it runs consistently across environments.
  - Includes app + dependencies + runtime.
- **Kubernetes (K8s)**:
  - Manages, deploys, and scales containerized applications.
  - Features:
    - **Auto-scaling**
    - **Load balancing**
    - **Self-healing** (restarts failed containers)
    - **Service discovery**

8. What is Threading? How can we use Async, Await, and Task in ASP.NET Core?

- **Threading**: Runs multiple tasks simultaneously for better responsiveness.
- **Async & Await**: Enables non-blocking, asynchronous execution.
- **Task**: Represents a future operation.

Using `Task` keeps the app responsive while it waits for I/O or long-running operations to complete.

9. What is ORM? Give 2 examples

- **ORM (Object-Relational Mapping)**:
  - A technique to interact with the database using **objects**, rather than raw SQL.
  - Maps tables to classes and rows to objects.

Examples:

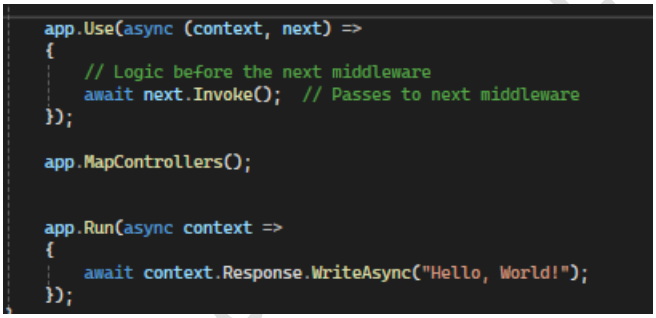
- Entity Framework Core
- Dapper

---

10. What is the difference between app.Use and app.Run?

- **app.Use**:
  - Adds middleware to the request pipeline.
  - Can **pass the request to the next middleware** using next().
- **app.Run**:
  - Defines a **terminal middleware**.
  - **Ends** the pipeline — does **not forward** the request.

Let's look at the example



```
app.Use(async (context, next) =>
{
    // Logic before the next middleware
    await next.Invoke();  // Passes to next middleware
});

app.MapControllers();


app.Run(async context =>
{
    await context.Response.WriteAsync("Hello, World!");
});
```

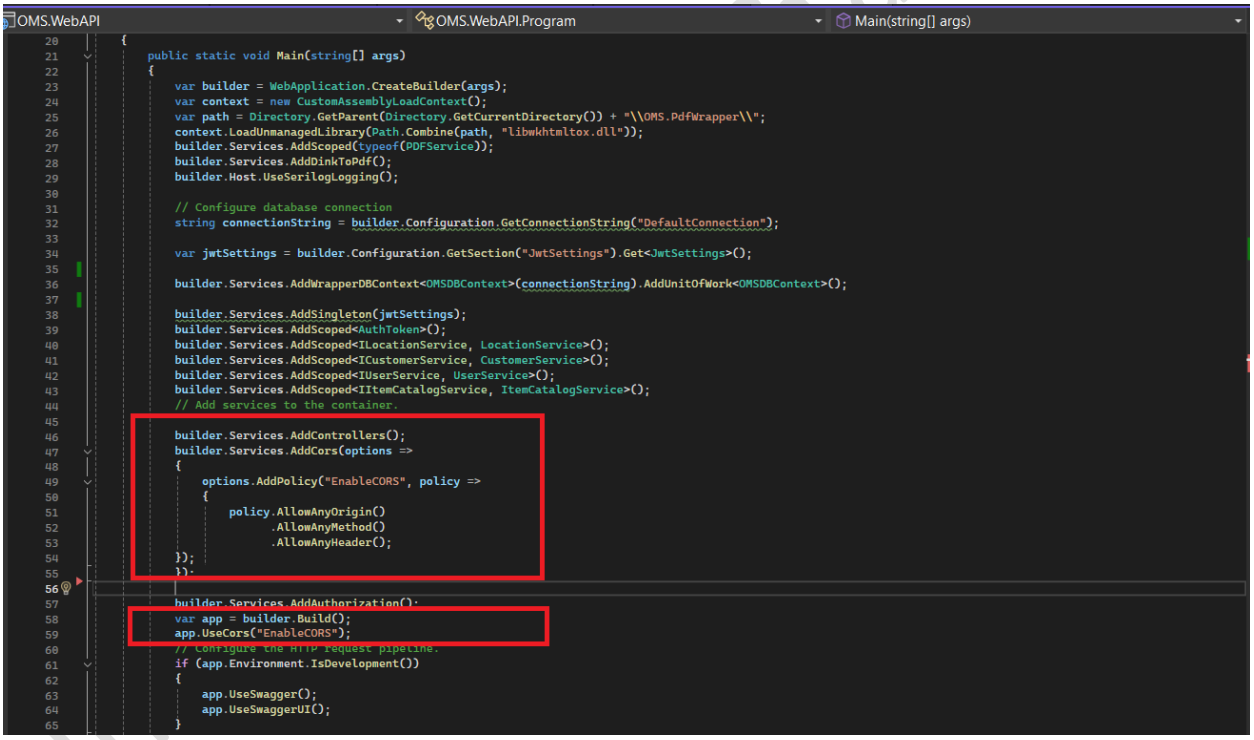
Web API Interview Questions

1. What is CORS (Cross-Origin Resource Sharing), and how do we configure it in ASP.NET Core Web API?

- **CORS** is a **security feature** that allows or restricts web applications from making HTTP requests to a domain **different from the one that served the web page**.
- By default, browsers block such cross-origin requests for security.

In **ASP.NET Core**, you can configure CORS in the Startup.cs file to allow requests from specific origins.

Let's look at the example



2. What is JWT authentication and how is it securing .NET Core API?

- **JWT (JSON Web Token)** is a compact, URL-safe token format used to represent **claims** between two parties.
- A JWT consists of **three parts**:
  - **Header**: Contains algorithm and token type.
  - **Payload**: Contains claims (user data, roles, etc.).
  - **Signature**: Used to verify the token's authenticity and integrity.
- JWT is **stateless**:
  - The server does **not store session information**.
  - The token itself carries all the required data to **authenticate** and **authorize** the user.
- In **ASP.NET Core**, JWT is used for authentication by:
  - Validating the token present in the **Authorization header** of incoming HTTP requests.

Let's look at the example


```
string connectionString = builder.Configuration.GetConnectionString("DefaultConnection");

var jwtSettings = builder.Configuration.GetSection("JwtSettings").Get<JwtSettings>();

builder.Services.AddWrapperDBContext<OMSDBContext>(connectionString).AddUnitOfWork<OMSDBContext>();

builder.Services.AddSingleton(jwtSettings);
builder.Services.AddScoped<AuthToken>();
builder.Services.AddScoped<ILocationService, LocationService>();
builder.Services.AddScoped<ICustomerService, CustomerService>();
builder.Services.AddScoped<IUserService, UserService>();
builder.Services.AddScoped<IItemCatalogService, ItemCatalogService>();
// Add services to the container.

builder.Services.AddControllers();
builder.Services.AddCors(options =>
{
    options.AddPolicy("EnableCORS", policy =>
    {
        policy.AllowAnyOrigin()
              .AllowAnyMethod()
              .AllowAnyHeader();
});
});
// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();

builder.Services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
    options.DefaultScheme = JwtBearerDefaults.AuthenticationScheme;
}).AddJwtBearer(o =>
{
    o.RequireHttpsMetadata = true;
    o.SaveToken = true;
    o.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuerSigningKey = true,
        IssuerSigningKey = new SymmetricSecurityKey(Encoding.ASCII.GetBytes(builder.Configuration["JwtSettings:Key"])),
        ValidateIssuer = false,
        ValidateAudience = false
    };
});
```

3. What is RESTful API and explain the difference between HttpPost, HttpPatch, and HttpPut?

- A **RESTful API** (Representational State Transfer) is an architectural style used for building:
  - **Stateless**
  - **Scalable**
  - **Resource-based** web applications
- It uses standard **HTTP methods** to perform CRUD operations.

HTTP Methods:

- **HttpPost**
  - Used to **create** a new resource.
  - **Behavior**: Each POST creates a new resource. It is **not idempotent** (multiple requests = multiple new entries).
- **HttpPut**
  - Used to **replace** an existing resource with new data.
  - **Behavior**: If you send the same PUT request multiple times, the result is the same. It is **idempotent**. It **replaces the entire resource**.
- **HttpPatch**
  - Used to **partially update** an existing resource.
  - **Behavior**: Only the **specified fields** are updated. The rest of the resource remains unchanged.

---

4. What is a Refresh Token and how to implement it in .NET Core?

- A **Refresh Token** is a **long-lived token** used to obtain a **new Access Token** after the original one has expired.
- It allows users to **stay logged in** without needing to re-authenticate every time.
- Commonly used with **JWT authentication** for secure, persistent login sessions.

Let's look at the example

```
public class AuthToken
{
    private readonly JwtSettings _jwtSettings;


    public AuthToken(JwtSettings jwtSettings)
    {
        _jwtSettings = jwtSettings;
    }
    public string GenerateJwtToken(string username)
    {
        var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(_jwtSettings.Key));
        var credentials = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);

        var claims = new[]
        {
        new Claim(JwtRegisteredClaimNames.Sub, username),
        new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString())
    };

        var token = new JwtSecurityToken(
            issuer: _jwtSettings.Issuer,
            audience: _jwtSettings.Audience,
            claims: claims,
            expires: DateTime.UtcNow.AddMinutes(_jwtSettings.ExpiryMinutes),
            signingCredentials: credentials
        );

        return new JwtSecurityTokenHandler().WriteToken(token);
    }
    public string GenerateRefreshToken()
    {
        return Guid.NewGuid().ToString();  // Generate a unique refresh token
    }
}
```

```
[HttpPost("login")]
public async Task<IActionResult> Login([FromBody] LoginModel login)
{
    try
    {
        byte[] hashPassword = [];

        if (login.UserId == null)
        {
            return Unauthorized(ApiResponse<string>.Unauthorized("Invalid username or password."));
        }
        //Verify User from database
        UserDTO loginUser = await VerifyUser(login.UserId);
        if (loginUser!= null)
        {
            if (loginUser.UserId == login.UserId && PasswordHelper.VerifyPassword(login.Password, loginUser.Password))
            {
                string token = _authToken.GenerateJwtToken(login.UserId);
                var refreshToken = _authToken.GenerateRefreshToken();

                // Store Refresh Token in Cookie
                Response.Cookies.Append("refreshToken", refreshToken, new CookieOptions
                {
                    HttpOnly = true,
                    Secure = true, // Use true in production for HTTPS
                    SameSite = SameSiteMode.Strict, // CSRF protection
                    Expires = DateTime.Now.AddDays(7) // Set expiration for the refresh token
                });
                var loginResponse = new { Token = token, User = loginUser, RefreshToken = refreshToken };
                return Ok(ApiResponse<object>.Success(loginResponse));
            }
            else
            {
                return Unauthorized(ApiResponse<string>.Unauthorized("Invalid credentials"));
            }
        }
        else {
            return Unauthorized(ApiResponse<string>.Unauthorized("Invalid credentials"));
        }
    }
    catch (Exception ex) {
        return BadRequest(ApiResponse<string>.Fail(new List<string> { ex.Message }));
    }
}
```

```
[HttpPost("refresh")]
public IActionResult RefreshToken(LoginModel login)
{
    // Check if the Refresh Token exists in the request cookies
    var refreshToken = Request.Cookies["refreshToken"];
    if (string.IsNullOrEmpty(refreshToken))
    {
        return Unauthorized("Refresh token is missing or invalid.");
    }

    // Validate the Refresh Token (you can check the expiration here, or in a real-world scenario, you would validate it in a database)
    // Here we simply check if the Refresh Token is valid. For real-world scenarios, you should have a mechanism to validate it.
    if (!IsValidRefreshToken(refreshToken)) // Replace with your custom validation logic
    {
        return Unauthorized("Invalid or expired refresh token.");
    }

    // Generate a new Access Token
    var newAccessToken = _authToken.GenerateJwtToken(login.UserId); // Replace with actual user ID

    return Ok(new { AccessToken = newAccessToken });
}
private bool IsValidRefreshToken(string refreshToken)
{
    // Retrieve the stored refresh token from the cookie
    var storedRefreshToken = Request.Cookies["refreshToken"];
    var storedExpirationTime = Request.Cookies["refreshTokenExpiration"];

    // Check if the refresh token exists in the cookie and matches the one passed in
    if (string.IsNullOrEmpty(storedRefreshToken) || storedRefreshToken != refreshToken)
    {
        return false; // Invalid if refresh token is missing or doesn't match
    }

    // Check if the expiration time is present and valid
    if (string.IsNullOrEmpty(storedExpirationTime) || !DateTime.TryParse(storedExpirationTime, out var expirationTime))
    {
        return false; // Invalid if expiration time is missing or can't be parsed
    }

    // Check if the refresh token has expired
    if (expirationTime < DateTime.Now)
    {
        return false; // Token is expired
    }

    // Optionally, you can check if the token has been revoked or blacklisted (if you're tracking it somewhere)
    return true; // Valid refresh token
}
```

5. What is Clean Architecture in .NET Core Web API?

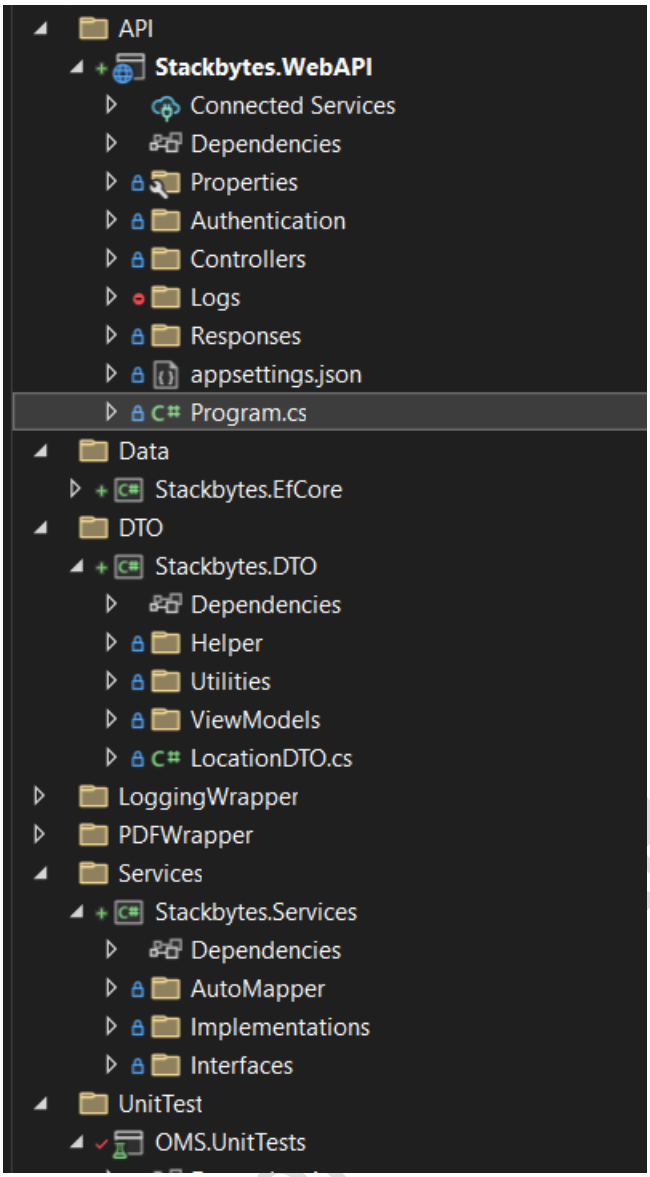
- **Clean Architecture** ensures **separation of concerns** by organizing the code into layers, each with its own responsibilities.
- This makes the application **more maintainable**, **testable**, and **scalable**.

Main Layers:

- **Presentation Layer (API)**
  - Handles incoming HTTP requests
  - Performs input validation
  - Maps requests to DTOs
- **DTO Layer**
  - Contains **Data Transfer Objects**
  - Decouples API contracts from database models
- **Service Layer**
  - Implements **business logic** and **application use cases**
  - Interacts with the repository and maps entities to DTOs
- **Data Layer**
  - Manages database access using **Entity Framework Core**
  - Contains database entities

Design Patterns Used:

- **Repository Pattern**:
  Encapsulates database operations for entities.
- **Unit of Work Pattern**:
  Manages transactional consistency across multiple repositories.



6. How to handle exceptions and errors in a .NET Core Web API?

- In .NET Core Web API, exceptions and errors can be handled using built-in **middleware** such as:
  - UseExceptionHandler() – For global exception handling in production.
  - UseDeveloperExceptionPage() – For detailed error information during development.
- You can also implement **custom error handling** by:
  - Creating a **global exception filter**
  - Using **try-catch blocks** inside controllers for local exception handling

7. What is the difference between WCF and Web API?

- **WCF (Windows Communication Foundation)**:
  - Provides **service-oriented architecture** (SOA)
  - Supports **multiple protocols**: HTTP, TCP, MSMQ, etc.
  - Offers advanced features: **transactions**, **message queuing**, **security**
  - Ideal for **enterprise applications** needing **multi-protocol support**
- **Web API**:
  - Lightweight framework for building **RESTful services** over **HTTP**
  - Communicates using **JSON or XML**
  - Focuses on **simplicity**, **scalability**, and modern **web/mobile applications**
  - Supports only **HTTP/HTTPS**

---

8. How do we implement response caching in a Web API?

- Use the [ResponseCache] attribute in your controller actions to define caching behavior.

Settings you can define:

- Duration: Time in seconds to cache the response
- Location: Client or server-side
- NoStore: Whether to store the response

Steps to enable:

1. Configure caching in Startup.cs using AddResponseCaching()
2. Apply the [ResponseCache] attribute to relevant controller actions

```
[HttpGet("GetCustomers")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[ProducesResponseType(StatusCodes.Status401Unauthorized)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
[ResponseCache(Duration = 60, Location = ResponseCacheLocation.Client)]
public async Task<IActionResult> GetCustomers( string userID, string? customerID, int?status )
{
    try
    {
        if (userID == string.Empty)
            return BadRequest(ApiResponse<string>.Fail(new List<string> {"Invalid UserID"}));

        var customers = await _customerService.GetCustomerByUserID(userID, customerID, status);
        if (customers == null || customers.Count == 0)
            return NotFound(ApiResponse<string>.NotFound("Customer not found."));

        return Ok(ApiResponse<object>.Success(customers));
    }
    catch (Exception ex)
    {
        Logger.Log().Information(ex.Message.ToString());
        return BadRequest(ApiResponse<string>.Fail(new List<string> { ex.Message }));
    }
}
```

9. What are ways to optimize a .NET Core Web API for high performance?

- Enable **Response Caching** to store and serve frequently accessed data, reducing redundant processing.
- Use **Asynchronous Programming** to handle multiple requests concurrently, improving scalability and responsiveness.
- Minimize **Middleware** to streamline request processing by eliminating unnecessary components.
- **Optimize Database Queries** using techniques like lazy loading, eager loading, or stored procedures to enhance data retrieval efficiency.
- Implement **Connection Pooling** to reuse database connections, lowering connection overhead and boosting performance.

---

10. Explain the difference between IActionResult and Task<IActionResult>?

- IActionResult is a **synchronous** return type that indicates the outcome of an action, such as Ok(), BadRequest(), or NotFound().
- It provides a result immediately after the action finishes.
- Task<IActionResult> is an **asynchronous** return type used when the action includes non-blocking operations such as database queries or API calls.
- It enables the method to **run asynchronously** and return once the operation is complete.

---

11. Explain the difference between IEnumerable and IQueryable.

IEnumerable:

- Works with **in-memory collections** like Lists and Arrays.
- Loads all data into memory first, then applies filters.
- Best suited for **small collections** already loaded in memory.

IQueryable:

- Works with **databases** via ORMs like Entity Framework or LINQ to SQL.
- Applies filters at the **database level** before fetching.
- Best suited for **large datasets** and **efficient querying**.

---

12. Explain the difference between Filters and Middleware.

Middleware:

- Runs for **every request** in the pipeline.
- Used for global concerns like **authentication**, **logging**, and **exception handling**.
- Registered in Program.cs using app.UseMiddleware<>().

Filters:

- Execute inside the **API pipeline**, at the controller or action level.
- Used for cross-cutting concerns like **validation**, **authorization**, and **caching**.
- Applied using attributes like [Authorize], [ValidateModel] or globally in AddControllers().

Summary:

- Middleware runs for all HTTP requests.
- Filters apply specifically to **controller actions** in the Web API.

---

13. Explain the difference between Lazy Loading and Eager Loading.

Lazy Loading:

- Loads **related data only when accessed**.
- Improves initial performance but may cause **multiple database calls**.
- **Use Case**: Load customer info first, then fetch orders only when needed (e.g., detailed view).

Eager Loading:

- Loads **related data upfront** with the main entity.
- Reduces number of database queries, but might load **unnecessary data**.
- **Use Case**: Load customer and order details together (e.g., admin dashboard showing order summary).

Summary:

- **Lazy Loading** = fetch on-demand.
- **Eager Loading** = fetch everything upfront.

Angular Interview Questions

---

1. What is the difference between Angular and AngularJS?

a. AngularJS:

- Supports **JavaScript**
- Follows **MVC (Model View Controller)** architecture
- Does **not have CLI**
- Does **not use Dependency Injection**
- **Slower** compared to modern frameworks

b. Angular:

- Supports **TypeScript and JavaScript**
- Uses **Component-based architecture**
- Comes with a **powerful CLI**
- Uses **Dependency Injection**
- **Faster** and optimized for performance

---

2. What is Angular Expression?

- Used in templates to **bind data** between the component and the view.
- Enables **communication between TypeScript and HTML**.

Data Binding Types:

- **String Interpolation** → {{ data }}
- **Property Binding** → [property]="data"
- **Event Binding** → (event)="expression"
- **Two-way Binding** → [(ngModel)]="data"

---

3. What is CanActivate and AuthGuard in Angular?

- Used to **secure routes** based on authentication and authorization.

Details:

- **CanActivate**: An **interface** that determines whether a route can be activated.
- **AuthGuard**: A **service** that implements CanActivate, contains logic to **allow or deny access**.

Let's look at the example

---

4. What is Eager Loading and Lazy Loading in Angular?

How to implement Lazy Loading?

Eager Loading:

- Loads **all modules at application startup**.
- Increases initial load time.
- Ideal for **small apps** or **core modules**.
- Modules are imported directly in app.module.ts.

Lazy Loading:

- Loads modules **only when needed**.
- Improves performance by **loading on demand**.
- Ideal for **large applications**.
- Uses loadChildren in app-routing.module.ts.

Let's look at the example of how to implement Lazy Loading

---

5. What is a Service in Angular?

- Services are used to **share data, logic, or functions** across multiple components.
- They help centralize **business logic and API calls**.

Key points:

- A **service** is a class decorated with `@Injectable`.
- Injected into components using **dependency injection**.
- Example use: A service fetches user data and shares it across components.

6. What are Directives in Angular?

- Directives **extend HTML behavior** and allow dynamic DOM manipulation.

Types of Directives:

- **Component Directives**: Special directives that have an HTML template.
- **Structural Directives**: Change structure by adding/removing elements
  Examples: `*ngIf`, `*ngFor`, `*ngSwitch`
- **Attribute Directives**: Change appearance or behavior of elements
  Examples: `ngClass`, `ngStyle`

---

7. What is a Pipe in Angular?

- A **Pipe** transforms data in templates before displaying it.

Common Usage:

- Formatting **dates**, **numbers**, **strings**, and **currency**

Types:

a. Pure Pipes:

- Executed **only when input changes**
- Examples: `DatePipe`, `UpperCasePipe`, `CurrencyPipe`

b. Impure Pipes:

- Executed **on every change detection cycle**
- Useful for **mutable data** like dynamic filtering/sorting

8. What are the ways to pass data in Angular?

Between components:

- @Input: From **parent → child**
- @Output: From **child → parent**
- **Shared Service**: For **sibling components**
- #Var: Access child component properties/methods in the template
- @ViewChild: Access child properties/methods in **TypeScript**

---

9. What is Microservices?

- Microservices is an **architectural style** where apps are split into **independent, modular services**.
- Each service handles a **specific business function** and communicates via **APIs**.
- Improves:
  - **Scalability**
  - **Flexibility**
  - **Maintainability**

Tools Used:

- Docker
- Kubernetes
- API Gateway

Thank you!

Hope you liked our above interview questions and we wish you all the best for your next interview.

In case of any doubts or clarification, please reach out to us on our email id – stackbytes08@gmail.com or our social media pages.