Self-Study

1. What is a controller factory and how is it used under the hood for every request ?

What is a Controller Factory?

A Controller Factory in ASP.NET MVC is a component responsible for creating controller instances for every incoming HTTP request. It's an implementation of the IControllerFactory interface, which defines how controllers are instantiated and released.

How it is used under the hood for every request:

1. When the MVC framework receives a request, it parses the URL to determine which controller and action method to invoke.
2. Before executing the action, MVC calls the controller factory to create an instance of the requested controller.
3. The controller factory uses the controller name (extracted from the route) and optionally the RequestContext to locate the controller class and instantiate it. By default, ASP.NET MVC uses the DefaultControllerFactory.
4. The DefaultControllerFactory creates the controller instance using reflection and also interacts with the Dependency Injection container if one is configured (via the IControllerActivator or dependency resolver internally).
5. Once the controller is created, the MVC framework invokes the specified action method.
6. After the request is processed, the controller factory is also responsible for releasing (disposing) the controller instance.

Key Methods in IControllerFactory:

- CreateController(RequestContext requestContext, string controllerName): Creates and returns the controller instance.

- `ReleaseController(IController controller)`: Handles releasing or disposing the controller.

Why Controller Factory?

- It decouples the MVC framework from the actual creation logic of controllers.
- Allows injecting dependencies into controllers via custom factories.
- Makes it possible to extend or override controller creation behavior, such as for unit testing or using custom IoC containers.

Summary:

- The Controller Factory abstracts controller creation per request.
- It is called on each request to instantiate the controller based on route data.
- It enables dependency injection and customization of controller lifecycle management.
- The MVC pipeline relies on it to provide controller instances ready to handle incoming HTTP requests.

2. What does this line mean builder.Services.AddControllersWithViews(); and what does this service do ?

The line:

`builder.Services.AddControllersWithViews();`

means that in the ASP.NET Core application's service container, you are registering the services needed to support MVC controllers with views.

What this service does:

- It configures the essential MVC services for controllers and views.
- Adds support for:

    - Controller activation and routing.
    - Model binding and validation.
    - View rendering with Razor view engine.
    - Support for features like data annotations, cache tag helpers, authorization, and formatting.
- It does not add services for Razor Pages (which are a different page-focused model).
- Essentially, it prepares the app to handle incoming HTTP requests via MVC controller actions that return views (HTML pages).

This method extends the service collection with everything you need to build a traditional MVC web application using controllers and Razor views, enabling you to separate concerns and build rich server-rendered UI combined with server-side logic.

In short: AddControllersWithViews() sets up the MVC pattern services necessary for controller/view based web apps in ASP.NET Core.

3. What is dependency injection and how is it used in the cycle of handling the request ?

What is Dependency Injection?

Dependency Injection (DI) is a design pattern where an object's dependencies are provided (injected) from an external source rather than the object creating them itself. This promotes loose coupling, easier testing, and better modularity.

How Dependency Injection is Used in the Request Handling Cycle in ASP.NET Core?

1. Service Registration: At app startup (typically in Program.cs), services and their implementations are registered with the dependency injection container using methods like:

```
builder.Services.AddScoped<IMyService, MyService>();
```

2. This tells the container how to create instances of services.
3. Request Start: When a new HTTP request arrives, ASP.NET Core begins processing request middleware and controllers.
4. Controller Activation:
    - When the framework needs to create a controller to handle the request, it asks the DI container to resolve the controller instance.
    - The container analyzes the controller's constructor parameters to determine which dependencies are required.
    - It automatically instantiates and injects those dependencies as per the registered services.
5. Dependency Injection into Services and Middleware:
    - Similarly, if the controller depends on other services (e.g., logging, database context), these are injected into the controller's constructor.
    - Middleware components can also have dependencies injected via the constructor.
6. **Request Processing:** The controller action executes with all required dependencies injected and ready to use.
7. **Disposal:** After the request completes, scoped dependencies tied to the request lifetime are disposed of by the DI container.

Why Use Dependency Injection?

- **Loose Coupling:** Classes do not create or manage their dependencies directly.
- **Testability:** Easier to mock dependencies for unit testing.
- **Single Responsibility:** Classes focus on their logic, not on managing dependencies.

- **Maintainability and Flexibility:** Services can be swapped or modified without changing dependent code.

Example

```
public interface IMessageService
{
    string GetMessage();
}

public class MessageService : IMessageService
{
    public string GetMessage() => "Hello, DI!";
}

public class HomeController : Controller
{
    private readonly IMessageService _messageService;

    public HomeController(IMessageService messageService)
    {
        _messageService = messageService;
    }

    public IActionResult Index()
    {
        var message = _messageService.GetMessage();
        return Content(message);
    }
}

// In Program.cs - Register service
builder.Services.AddScoped<IMessageService, MessageService>();
```

- The HomeController declares it needs IMessageService.
- DI container creates MessageService and injects it when instantiating the controller.

- Controller uses the injected service to handle the request.

Summary

Dependency Injection in ASP.NET Core is integral to the request handling pipeline. It ensures that controllers and middleware get their required dependencies automatically, which improves application design, testability, and maintainability.

4. What is a builder design pattern with an example ?

What is the Builder Design Pattern?

The Builder Design Pattern is a creational design pattern that provides a step-by-step approach to constructing complex objects. It separates the construction process of an object from its representation, allowing the same construction steps to create different representations. This pattern improves code readability, maintainability, and flexibility by avoiding large constructors with many parameters.

Key Characteristics:

- Encapsulates complex object construction in a separate builder class.
- Allows incremental construction of objects.
- Supports creating different types or variations of an object using the same building process.
- Helps manage complex creation logic and multiple combinations of object parts.

Example of Builder Design Pattern

Suppose you want to build a customizable computer where users can specify various components (CPU, RAM, storage, GPU):

```
// Product class representing the complex object
public class Computer
{
    public string CPU { get; set; }
```

```
    public int RAM { get; set; }
    public int Storage { get; set; }
    public string GPU { get; set; }
}

// Builder interface declaring building steps
public interface IComputerBuilder
{
    void SetCPU(string cpu);
    void SetRAM(int ram);
    void SetStorage(int storage);
    void SetGPU(string gpu);
    Computer Build();
}

// Concrete Builder implementing the building steps
public class ComputerBuilder : IComputerBuilder
{
    private Computer _computer = new Computer();

    public void SetCPU(string cpu) => _computer.CPU = cpu;
    public void SetRAM(int ram) => _computer.RAM = ram;
    public void SetStorage(int storage) => _computer.Storage =
storage;
    public void SetGPU(string gpu) => _computer.GPU = gpu;

    public Computer Build() => _computer;
}

// Client code using the builder step-by-step
class Program
{
    static void Main()
    {
        IComputerBuilder builder = new ComputerBuilder();

        builder.SetCPU("Intel i7");
        builder.SetRAM(16);
        builder.SetStorage(512);
```

```
        builder.SetGPU("NVIDIA RTX 3070");

        Computer myComputer = builder.Build();

        Console.WriteLine($"CPU: {myComputer.CPU}, RAM:
{myComputer.RAM}GB, Storage: {myComputer.Storage}GB, GPU:
{myComputer.GPU}");
    }
}
```

Summary

- The Builder Pattern separates object construction from its representation.
- Enables creation of complex objects step-by-step.
- Provides flexibility to easily change components without complex constructors.
- Useful for objects with many optional or configurable parts.

This pattern is widely used in scenarios like building complex UI components, SQL query construction, and assembling various combinations of objects