ASP.NET Core - True Ultimate Guide

Section 12: Dependency Injection

Services

In ASP.NET Core MVC, services are classes responsible for implementing the core business logic of your application. They are designed to be reusable, self-contained, and independent of specific controllers or views. Services are the backbone of your application, handling tasks like data access, calculations, communication with external systems, and any other operations that involve the "how" of your application's functionality.

Key Purposes of Services

1. **Encapsulation of Business Logic:** Services provide a clean way to encapsulate complex operations and keep them separate from your presentation layer (controllers and views).
2. **Reusability:** A single service can be used by multiple controllers, promoting DRY (Don't Repeat Yourself) principles and making your code more maintainable.
3. **Testability:** Services can be easily unit tested in isolation, allowing you to verify the correctness of your business logic without the overhead of running the entire application.
4. **Dependency Injection (DI):** Services are typically registered in the DI container, making them easily accessible to controllers and other components within your application.

Typical Responsibilities of Services

- **Data Access:** Communicating with databases or other data sources to fetch, insert, update, or delete data.
- **Business Rules:** Implementing the rules that govern how your application behaves (e.g., validation, calculations, transformations).
- **Integration:** Interacting with external systems or APIs.
- **Notifications:** Sending emails, SMS messages, or other notifications.
- **Logging:** Recording events and errors for troubleshooting and analysis.

Code

```
// CitiesService.cs (Service)

namespace Services

{

   public class CitiesService
```

```
    {
        private List<string> _cities;

        // Constructor
        public CitiesService()
        {
            _cities = new List<string>() { "London", "Paris", "New York", "Tokyo", "Rome" };
        }

        public List<string> GetCities()
        {
            return _cities;
        }
    }
}

// HomeController.cs (Controller)
public class HomeController : Controller
{
    private readonly CitiesService _citiesService;

    // Constructor (injecting the service)
    public HomeController()
    {
        _citiesService = new CitiesService();
    }

    [Route("/")]
    public IActionResult Index()
    {
        List<string> cities = _citiesService.GetCities();
```

```
        return View(cities); // Pass the data to the view
    }
}
```

Note that in this code example, there is no dependency injection being used. In real-world projects, it's good practice to register your services with ASP.NET Core's built-in dependency injection container and then have them injected into your controllers (or other components) through the constructor.

Explanation

1. **CitiesService Class:** This class represents a simple service that holds a list of city names and provides a GetCities method to retrieve them.
2. **HomeController Class:**
    - **Dependency:** It has a dependency on the CitiesService class.
    - **Instantiation:** In this simplified example, the CitiesService object is created directly within the controller's constructor.
    - **Action Method:** The Index action method calls the GetCities method of the _citiesService to retrieve the list of cities and then passes this data to the view.

Best Practices

- **Single Responsibility Principle (SRP):** Design your services to have a single responsibility to keep them focused and maintainable.
- **Dependency Injection:** Use dependency injection to manage service lifetimes and dependencies, making your code loosely coupled and easier to test.
- **Interface-Based Programming:** Define interfaces for your services to create abstraction layers and facilitate testing with mocks.
- **Clear Naming:** Use descriptive names for your services and their methods to make your code self-documenting.
- **Testing:** Write unit tests for your services to ensure that your business logic works correctly in isolation.

Key Points to Remember

- **Encapsulation:** Services encapsulate the "how" of your application logic, separating it from the presentation layer.
- **Reusability:** Services can be reused across multiple controllers.

- **Testability:** Services are designed to be easily unit tested.
- **Dependency Injection:** Services are typically managed by the DI container and injected into controllers.

Dependency Inversion Principle (DIP)

DIP is a design principle that promotes loosely coupled software architecture. It states that:

1. **High-level modules should not depend on low-level modules.** Both should depend on abstractions.
2. **Abstractions should not depend on details.** Details should depend on abstractions.

In simpler terms:

- Instead of tightly coupling your classes by having them depend on concrete implementations, they should depend on abstractions (interfaces or abstract classes).
- This allows you to easily swap out implementations without changing the higher-level code.

Inversion of Control (IoC): Shifting Responsibility

IoC is a broad principle that involves transferring the control of object creation and management from your application code to a framework or container. Instead of your classes explicitly creating their dependencies, they receive them from an external source. This external source is often a DI container.

Dependency Injection (DI): The Practical Tool

DI is a specific implementation of the IoC principle. It involves supplying dependencies to a class from outside the class itself. The most common way to do this in ASP.NET Core is through constructor injection, but there are also other techniques like property injection and method injection.

Benefits of DIP, IoC, and DI

- **Loose Coupling:** Reduces the direct dependencies between classes, making them easier to change and test independently.
- **Flexibility:** You can easily swap out different implementations of dependencies without affecting the consuming class.

- **Testability:** Unit testing becomes much easier, as you can provide mock dependencies to isolate the code under test.
- **Maintainability:** Code becomes more modular, easier to understand, and less prone to ripple effects from changes.

Code

```
// ServiceContracts (Interface)
namespace ServiceContracts
{
   public interface ICitiesService // Abstraction of CitiesService
   {
      List<string> GetCities();
   }
}

// Services (Implementation)
namespace Services
{
   public class CitiesService : ICitiesService // CitiesService depends on the ICitiesService abstraction
   {
      // ... (Implementation of GetCities) ...
   }
}
```

The interface ICitiesService defines the abstraction for a service that can retrieve a list of cities. The class CitiesService provides the concrete implementation, but it depends on the ICitiesService interface, not on a concrete class.

Code

```
// Program.cs (or Startup.cs)
builder.Services.Add(new ServiceDescriptor(
   typeof(ICitiesService),     // Interface to register
```

```
    typeof(CitiesService),     // Concrete implementation
    ServiceLifetime.Transient  // Lifetime of the service (more on this later)
));

// HomeController.cs (Controller)
public class HomeController : Controller
{
    private readonly ICitiesService _citiesService; // Dependency on the interface

    // Constructor injection
    public HomeController(ICitiesService citiesService)
    {
        _citiesService = citiesService;
    }

    // ... (Action methods) ...
}
```

In this code:

1. **Service Registration:** The CitiesService is registered in the DI container using the Add method. The ServiceDescriptor specifies:
    - The interface type (ICitiesService) that other components will request.
    - The concrete implementation type (CitiesService) that the container will create.
    - The lifetime of the service (ServiceLifetime.Transient means a new instance is created for each request).
2. **Constructor Injection:** The HomeController constructor has a parameter of type ICitiesService. This means the DI container will automatically provide an instance of CitiesService when the controller is created.

Notes

- **Abstractions:** Focus on designing interfaces or abstract classes to represent your dependencies.
- **Loose Coupling:** Your classes should depend on abstractions, not concrete implementations.

- **Dependency Injection:** Use DI containers (like the one built into ASP.NET Core) to manage and resolve dependencies.
- **Service Lifetimes:** Understand the different service lifetimes (Transient, Scoped, Singleton) and choose the right one for each service.

Scenario - A Light Switch and Light Bulb

- **Without DIP/IoC/DI:**
    - Imagine a traditional light switch directly wired to a specific light bulb. If you want to change the light bulb to a different type, you might need to rewire the switch, as it's tightly coupled to the original bulb. This is analogous to tightly coupled code, where classes depend directly on specific implementations of other classes.
- **With DIP/IoC/DI:**
    - Now, imagine a standard electrical outlet and a plug. The outlet represents an interface (an abstraction), while the plug represents a class that implements this interface. You can plug any compatible device (light bulb, fan, etc.) into the outlet, and it will work. This is the essence of DIP - depending on abstractions, not concrete implementations.
    - The "inversion of control" comes in because the outlet (interface) dictates the shape of the plug (implementation), not the other way around.
    - "Dependency injection" happens when you plug a device into the outlet. The outlet doesn't create the device; it simply receives it and allows it to function.

Visual Representation

Without DIP/IoC/DI:

Light Switch -----> Specific Light Bulb

(Tightly Coupled)

With DIP/IoC/DI:

Outlet (Interface) <--- Plug (Implementation)

|

|

Light Bulb, Fan, etc.

Code Analogy

```
// Without DIP
class LightSwitch
{
   private SpecificLightBulb _bulb = new SpecificLightBulb();

   public void TurnOn()
   {
      _bulb.Illuminate();
   }
}

// With DIP
interface ILight
{
   void Illuminate();
}

class LightBulb : ILight { /* ... */ }

class LightSwitch
{
   private ILight _light;

   public LightSwitch(ILight light) // Dependency injection
   {
      _light = light;
```

```
    }

    public void TurnOn()
    {
        _light.Illuminate();
    }
}
```

In the DIP version:

- LightSwitch depends on the ILight interface, not a specific bulb.
- The LightBulb class implements ILight.
- The LightSwitch constructor takes an ILight parameter (dependency injection). Now, you can pass in any object that implements ILight (a different type of bulb, a fan, etc.), and the LightSwitch will work with it.

Key Points

- **DIP:** Depend on abstractions (interfaces) to make your code more flexible and maintainable.
- **IoC:** Let a framework or container manage object creation and dependencies.
- **DI:** Implement IoC by having dependencies provided to your classes (often through constructors).
- **Benefits:** Achieve loose coupling, flexibility, testability, and maintainability in your code.

Service Lifetimes

When you register a service in the DI container, you specify its lifetime. This determines how the DI container creates and manages instances of that service throughout your application's execution.

Three Main Lifetime Options

1. **Transient:**
    - **Creation:** A new instance is created each time the service is requested (injected).
    - **Lifetime:** The instance lives only as long as it's needed to fulfill the current request.
    - **Usage:** Ideal for lightweight, stateless services where each request requires a fresh instance.
    - **Example:** Database context, logger, helper classes.

2. **Scoped:**
    - **Creation:** A single instance is created per HTTP request (or scope) within your application.
    - **Lifetime:** The instance is shared throughout the request and disposed of when the request ends.
    - **Usage:** The most common lifetime for web applications. Ensures consistency within a request while avoiding long-lived objects.
    - **Example:** User-specific data, transaction handling, shopping carts.

3. **Singleton:**
    - **Creation:** A single instance is created for the entire lifetime of your application.
    - **Lifetime:** The instance is shared across all requests and components.
    - **Usage:** Suitable for stateless services, caches, background tasks, or configurations that you want to load once and share globally.
    - **Example:** Application-wide configuration settings, shared caches, singleton design pattern implementations.

Choosing the Right Lifetime

The lifetime you choose for a service depends on its purpose and how you intend to use it:

- **State:** If your service holds state that needs to be unique per request, use Scoped. If the state needs to be shared globally, use Singleton. If state is irrelevant, Transient is often sufficient.
- **Resource Usage:** Singleton services consume memory for the entire application lifetime, so use them judiciously.
- **Concurrency:** Be mindful of concurrency issues when using singleton services in multi-threaded environments.

Registration Examples

```
// Startup.cs (or Program.cs)

builder.Services.AddTransient<ITransientService, TransientService>();

builder.Services.AddScoped<IScopedService, ScopedService>();

builder.Services.AddSingleton<ISingletonService, SingletonService>();
```

Lifetime Best Practices

- **Prefer Scoped for Web Apps:** In most cases, Scoped is the recommended lifetime for services in web applications.
- **Avoid Captive Dependencies:** Don't inject a shorter-lived service (e.g., Transient) into a longer-lived one (e.g., Singleton). This can lead to unexpected behavior and memory leaks.
- **Consider Thread Safety:** If you use a singleton service, ensure it's thread-safe if it will be accessed concurrently.

Dependency Injection Techniques in ASP.NET Core

ASP.NET Core provides a robust built-in dependency injection (DI) container that allows you to inject services into your application's components (controllers, middleware, view components, etc.). Here are the primary ways you can inject dependencies:

1. **Constructor Injection (Most Common):**
    - **Mechanism:** Dependencies are passed as parameters to the class's constructor.
    - **Benefits:**
        - Easy to understand and use.
        - Encourages loose coupling and testability.
        - Ensures that required dependencies are available before the class is used.
    - **Example:**

```
public class ProductsController : Controller
{
   private readonly IProductService _productService;

   public ProductsController(IProductService productService)
   {
      _productService = productService;
   }
}
```

2. **Property Injection (Less Common):**
    - **Mechanism:** Dependencies are assigned to public properties with a [FromServices] attribute.
    - **Benefits:**
        - Can be useful when you have optional dependencies or want to avoid constructor clutter.
        - Allows for lazy loading of dependencies.
    - **Example:**

```
   public class MyMiddleware
   {
```

```
    [FromServices]
    public ILogger<MyMiddleware> Logger { get; set; }
}
```

3. **Method Injection (Least Common):**
    - **Mechanism:** Dependencies are passed as parameters to individual methods.
    - **Benefits:**
        - Provides fine-grained control over when dependencies are resolved.
        - Can be useful in cases where you only need a dependency within a specific method.
    - **Example:**

```
public IActionResult Index([FromServices] IUserService userService)
{
  // ... use the userService within this method
}
```

    - Use this injection technique if you require a service in one or few actions.

4. **Action Method Injection:**

- **Mechanism:** Injects services directly into action methods as parameters.
- **Benefits:**
    - Simplifies dependency management within specific actions.
    - Useful for scenarios where a dependency is needed only in a particular action method.
- **Example:**

```
public IActionResult MyAction([FromServices] IMyService service)
{
  // ... use the service within this action
}
```

Choosing the Right Injection Technique

- **Constructor Injection:** The recommended and most common approach for mandatory dependencies.
- **Property Injection:** Use for optional dependencies or when constructor injection is cumbersome.
- **Method Injection:** Consider this for dependencies that are only needed within specific methods or for finer control over dependency resolution.
- **Action Method Injection:** Ideal for scenarios where a dependency is required only within a specific action method.

Key Points to Remember

- **Loose Coupling:** Regardless of the injection type, the core principle of DI is to achieve loose coupling between components.
- **Dependency Inversion Principle (DIP):** Ensure that your classes depend on abstractions (interfaces) rather than concrete implementations.
- **Dependency Injection Container:** ASP.NET Core's built-in DI container handles the registration and resolution of services.
- **Service Lifetimes:** Understand the different service lifetimes (Transient, Scoped, Singleton) and choose the appropriate one for each dependency.

Best Practices of DI

1. Constructor Injection as the Default

- **Why:** Constructor injection is the most straightforward and reliable way to inject dependencies. It ensures that a class has all its required dependencies before it can be used, promoting object validity.
- **How:** Declare all the necessary dependencies as constructor parameters.

```
public class ProductService : IProductService
{
    private readonly IProductRepository _productRepository;
    private readonly ILogger<ProductService> _logger;
```

```
    public ProductService(IProductRepository productRepository, ILogger<ProductService> logger)
    {
        _productRepository = productRepository;
        _logger = logger;
    }
}
```

2. Use Interfaces for Dependencies

- **Why:** Interfaces promote loose coupling, enabling you to easily swap implementations during testing or when using different environments.
- **How:** Define interfaces for your services and have your classes depend on the interfaces, not concrete implementations.

C#

```
public interface IProductRepository { /* ... */ }

public class ProductRepository : IProductRepository { /* ... */ }
```

3. Avoid Service Locator Anti-Pattern

- **Why:** The Service Locator pattern involves directly accessing the DI container from within your classes (e.g., using IServiceProvider.GetService()). This tightly couples your code to the DI container and makes testing harder.
- **How:** Instead, have the DI container inject dependencies directly into your classes.

4. Register Dependencies at the Composition Root

- **Why:** The composition root (typically the Program.cs or Startup.cs file) is where you should configure your DI container. This centralizes dependency registration and makes it easier to manage and understand your application's structure.
- **How:** Use the IServiceCollection.Add* methods (e.g., AddTransient, AddScoped, AddSingleton) to register your services and their lifetimes.

5. Choose the Appropriate Service Lifetime

- **Transient:** A new instance is created each time the service is requested.
- **Scoped:** A single instance is created per request.
- **Singleton:** A single instance is created for the entire application lifetime.

- **How:** Carefully consider the nature of your service (stateful vs. stateless) and its usage patterns to choose the right lifetime.

6. Avoid Captive Dependencies

- **Why:** A captive dependency occurs when you inject a shorter-lived service (e.g., Transient) into a longer-lived service (e.g., Singleton). This can lead to unexpected behavior and memory leaks.
- **How:** Ensure that your service lifetimes are compatible and that you don't inadvertently capture a transient instance within a singleton.

7. Use Decorators to Add Cross-Cutting Concerns

- **Why:** Decorators wrap existing services and allow you to add additional behavior (e.g., logging, caching) without modifying the original service.
- **How:** Implement the same interface as the service you want to decorate and inject the original service into the decorator.

8. Leverage Options Pattern for Configuration

- **Why:** The Options pattern provides a strongly typed way to access configuration settings in your services.
- **How:** Create classes that represent your configuration sections and use the IOptions interface to inject them.

9. Consider Pure DI for Testability

- **Why:** Pure DI (avoiding the IServiceProvider altogether) makes your classes more testable, as you can easily provide mock dependencies during unit testing.
- **How:** Design your classes so that all their dependencies are passed through the constructor or other injection points.

10. Don't Overuse DI

- **Why:** Dependency injection is a powerful tool, but it should be used judiciously. Overusing it can lead to complex object graphs and make code harder to reason about.
- **How:** Don't inject every single class in your application. Use DI for services and components with clear dependencies and where you need flexibility and testability.

Autofac

While ASP.NET Core has a built-in dependency injection (DI) container, Autofac is a popular third-party IoC container known for its flexibility, advanced features, and customization options. It seamlessly integrates with ASP.NET Core, providing you with more powerful tools to manage your dependencies.

Key Advantages of Autofac

- **Flexibility:** Offers a wider range of component lifetime scopes and registration options compared to the built-in container.
- **Customization:** Provides more fine-grained control over how dependencies are resolved and managed.
- **Advanced Features:** Supports features like module-based registration, property injection, assembly scanning, and interception (for cross-cutting concerns).
- **Performance:** Generally considered to have a good performance profile.

Integrating Autofac with ASP.NET Core

1. **Install Package:** Add the Autofac.Extensions.DependencyInjection NuGet package to your project.
2. **Configure Container:** In your Program.cs (or Startup.cs in older versions), replace the default service provider factory with Autofac's:

```
builder.Host.UseServiceProviderFactory(new AutofacServiceProviderFactory());
```

3. **Register Services:** Use the builder.Host.ConfigureContainer<ContainerBuilder> method to access Autofac's ContainerBuilder and register your services: C#
4. builder.Host.ConfigureContainer<ContainerBuilder>(containerBuilder =>
5. {
6. // Your Autofac registration logic here

```
});
```

Code

```
// Program.cs
// ... other imports ...

builder.Host.UseServiceProviderFactory(new AutofacServiceProviderFactory()); // Use Autofac
builder.Services.AddControllersWithViews(); // Add MVC services

builder.Host.ConfigureContainer<ContainerBuilder>(containerBuilder =>
{
   containerBuilder.RegisterType<CitiesService>().As<ICitiesService>().InstancePerLifetimeScope(); // Register CitiesService as Scoped
});

var app = builder.Build();
// ... the rest of the code ...
```

In this code:

1. **UseServiceProviderFactory:** This line tells ASP.NET Core to use Autofac as the service provider.
2. **AddControllersWithViews:** This registers the necessary services for MVC (models, views, controllers).
3. **ConfigureContainer:** This lambda expression gives you access to Autofac's ContainerBuilder.
4. **RegisterType<CitiesService>().As<ICitiesService>().InstancePerLifetimeScope();:** This registers the CitiesService class as an implementation of the ICitiesService interface with a scoped lifetime.

Autofac Registration Methods

- **RegisterType<T>():** Registers a specific type.
- **As<T>():** Specifies the interface or base type that the registered type should be resolved as.
- **Lifetime Scopes:**
    - InstancePerDependency() (equivalent to Transient)
    - InstancePerLifetimeScope() (equivalent to Scoped)
    - SingleInstance() (equivalent to Singleton)

Notes

- **Why Autofac?**
    - More flexibility and control over dependency resolution.
    - Additional features (modules, property injection, etc.).
- **Integration:** Replace the default service provider factory with Autofac's.
- **Registration:** Use Autofac's syntax (RegisterType, As, lifetime scopes) within the ConfigureContainer lambda.
- **Familiar Concepts:** The underlying concepts of DI (abstractions, lifetimes) remain the same, just with different syntax.

Service Scope

In ASP.NET Core DI, a service scope is a logical boundary that defines the lifetime of services registered as *Scoped*. When a scope is created, the DI container instantiates any scoped services that are required within that scope. These scoped service instances are then shared across all components within that scope, ensuring consistency and avoiding unnecessary object creation.

How Service Scopes Work in ASP.NET Core

1. **Request Scope (Default):** In ASP.NET Core web applications, the most common scope is the *request scope*. A new scope is automatically created at the beginning of each HTTP request. All scoped services are resolved from this request scope and remain alive throughout the entire request-response cycle. Once the request is processed, the scope is disposed, and all scoped services within it are also disposed.

2. **Explicitly Creating Scopes:** You can also create custom scopes manually. This is useful in scenarios where you need a scoped lifetime for operations that don't directly correspond to an HTTP request (e.g., background tasks, unit testing). You can create a scope using the IServiceProvider.CreateScope() method.

Code

```
using (var scope = provider.CreateScope())
{
   var scopedService = scope.ServiceProvider.GetRequiredService<IScopedService>();
   // Use the scopedService within this scope
}
```

Lifetime of Scoped Services

- **Creation:** A new instance of a scoped service is created the first time it's requested within a scope.
- **Sharing:** Subsequent requests for the same scoped service within the same scope will receive the same instance.
- **Disposal:** When the scope is disposed (e.g., at the end of an HTTP request), all scoped services within that scope are also disposed.

Benefits of Service Scopes

- **State Management:** Scoped services are perfect for managing state that needs to persist throughout a request but should not leak across different requests.
- **Efficient Resource Usage:** Scopes ensure that you don't create unnecessary instances of services, leading to better memory management.
- **Consistency:** Scoped services provide a consistent view of data and state within a single request.

Common Scenarios for Scoped Services

- **Database Contexts (EF Core):** A new database context instance is usually created per request to ensure data isolation and avoid concurrency issues.
- **User-Specific Data:** Services holding data specific to the current user (e.g., shopping cart) are often scoped to the request.
- **Logging with Context:** If you need to log information with request-specific context, a scoped logger is beneficial.
- **Transactions:** If you need to maintain transactional integrity within a request, you can use a scoped service to manage the transaction.

Important Considerations

- **Avoid Captive Dependencies:** Be cautious of injecting a scoped service into a singleton service. This can lead to unexpected behavior and memory leaks because the scoped service will be held alive for the entire application lifetime.
- **Explicit Disposal:** When you create custom scopes, remember to dispose them properly using a using statement or by manually calling Dispose() on the IServiceScope object.

Key Points to Remember

Dependency Inversion Principle (DIP)

- **Core Idea:** High-level modules shouldn't depend on low-level modules; both should depend on abstractions (interfaces/abstract classes).
- **Goal:** Loose coupling, flexibility, testability.

Inversion of Control (IoC)

- **Core Idea:** Transfer control of object creation and management from your code to a framework or container (e.g., the DI container).
- **Goal:** Decoupling, simplified configuration, improved testability.

Dependency Injection (DI)

- **Core Idea:** Dependencies are provided (injected) into a class from an external source (usually a DI container).
- **Types in ASP.NET Core:**
    - **Constructor Injection:** Dependencies are passed as constructor parameters (most common).
    - **Property Injection:** Dependencies are assigned to properties with the [FromServices] attribute.
    - **Method Injection:** Dependencies are passed as method parameters.
- **Benefits:**
    - Loose coupling
    - Flexibility
    - Testability
    - Maintainability

Service Lifetimes in ASP.NET Core DI

- **Transient:** A new instance created each time a service is requested.
- **Scoped:** A single instance created per request/scope.
- **Singleton:** A single instance created once and shared throughout the application's lifetime.

Autofac: A Powerful IoC Container

- **Purpose:** Alternative to the built-in ASP.NET Core DI container.
- **Benefits:**
    - More flexible component registration and lifetime options
    - Advanced features (modules, property injection, interception)
    - Good performance
- **Integration:**
        - Install Autofac.Extensions.DependencyInjection.
        - Use builder.Host.UseServiceProviderFactory(new AutofacServiceProviderFactory()) in Program.cs.
        - Register services in builder.Host.ConfigureContainer<ContainerBuilder>.

Autofac Registration

- containerBuilder.RegisterType<T>().As<TInterface>().InstancePerLifetimeScope();
    - Equivalent to services.AddScoped<TInterface, T>().
- InstancePerDependency() (Transient), SingleInstance() (Singleton)

Interview Tips

- **Conceptual Understanding:** Be able to explain the principles of DIP, IoC, and DI and how they relate to each other.
- **Practical Application:** Demonstrate your ability to choose the right lifetime for a given service and explain the implications of each choice.
- **Best Practices:** Discuss the advantages of using interfaces and constructor injection.
- **Autofac:** Highlight the benefits of using Autofac over the built-in container and showcase your knowledge of its registration syntax.
- **Troubleshooting:** Explain how you would diagnose common DI issues (e.g., circular dependencies, incorrect lifetimes).