Dependency Injection (DI)

Chapter 1: Foundations

Introduction

As applications grow, classes begin to depend on one another. For example, a ProductService may require a ProductRepository to fetch data from a database. The way these dependencies are created has a major impact on maintainability, flexibility, and testability.

Dependency Injection is a design pattern in which a class receives its dependencies from an external source rather than creating them itself, allowing object creation and dependency management to be separated from business logic.

ASP.NET Core has built-in support for Dependency Injection, making it one of the most commonly used patterns in modern .NET applications.

1. What is a Dependency?

A dependency is any object that another object needs to perform its work.

Examples of common dependencies include ProductRepository, Logger, EmailService, CacheService, PaymentGateway, ConfigurationService, and HTTP clients.

2. What Problem Does Dependency Injection Solve?

Without Dependency Injection, a class is responsible for creating or obtaining its own dependencies, often using the `new` keyword.

This creates tight coupling, making the application difficult to maintain and test.

```
public class ProductService
{
    private ProductRepository repository;

    public ProductService()
    {
        repository = new ProductRepository();
    }
}
```

Problems with this approach:

- Tight coupling to a specific implementation.

- Difficult to replace ProductRepository with another implementation.
- Unit testing becomes difficult because the dependency cannot be substituted.
- Business logic and object creation become mixed together.

3. Tight Coupling vs Loose Coupling

Tight Coupling	Loose Coupling
Creates dependencies internally	Receives dependencies externally
Depends on concrete classes	Depends on abstractions
Hard to test	Easy to test
Hard to replace implementations	Easy to swap implementations
Poor maintainability	Better maintainability

4. Before and After Dependency Injection

Before Dependency Injection

```
public class ProductService
{
    private ProductRepository repository = new
ProductRepository();
}
```

After Dependency Injection

```
public interface IProductRepository
{
    List<string> GetProducts();
}

public class ProductService
{
    private readonly IProductRepository repository;

    public ProductService(IProductRepository repository)
    {
        this.repository = repository;
    }
}
```

The ProductService no longer creates ProductRepository. Instead, the dependency is supplied from outside. This makes the class easier to maintain, extend, and test.

5. Types of Dependency Injection

Constructor Injection

The dependency is passed through the constructor. This is the recommended approach in ASP.NET Core because required dependencies are always available.

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

Setter Injection

The dependency is assigned after the object has been created. This is useful for optional dependencies but is less common in .NET.

```
public IProductRepository Repository { get; set; }
```

Method Injection

The dependency is passed only to the method that requires it.

```
public void PrintProducts(IProductRepository repository)
{
    var products = repository.GetProducts();
}
```

Chapter Summary

In this chapter you learned what a dependency is, why creating dependencies inside a class leads to tight coupling, how Dependency Injection solves this problem, and the three types of Dependency Injection. Constructor Injection is the preferred approach in ASP.NET Core because it makes dependencies explicit and keeps classes easy to maintain.

Chapter 2: Advanced Concepts

1. Dependency Injection vs Inversion of Control (IoC)

Inversion of Control (IoC) is a design principle where the control of creating and managing objects is transferred from application code to an external mechanism. Dependency Injection (DI) is one of the most common ways to achieve IoC.

```
Without IoC:
Application -> Creates Objects

With IoC:
Application -> DI Container -> Creates Objects
```

Dependency Injection	Inversion of Control
A design pattern	A design principle
Supplies dependencies	Delegates object creation and dependency management
One way to implement IoC	Can be achieved using DI, Service Locator, etc.

2. Dependency Injection vs Service Locator

Both patterns provide dependencies, but they do so differently.

```
// Dependency Injection
public ProductService(IProductRepository repository){ }

// Service Locator
var repository = ServiceLocator.Get<IProductRepository>();
```

With Dependency Injection, required dependencies are visible in the constructor. With Service Locator, dependencies are hidden because the class fetches them itself.

Dependency Injection	Service Locator
Dependencies are explicit	Dependencies are hidden
Easy to understand	Harder to identify requirements
Preferred in modern .NET	Generally discouraged

3. Composition Root

The Composition Root is the place where the application's object graph is created and wired together. In ASP.NET Core, Program.cs typically acts as the Composition Root.

```
builder.Services.AddScoped<IProductRepository,
ProductRepository>();
builder.Services.AddScoped<ProductService>();
```

Business classes should not create dependencies. They simply receive them.

4. Dependency Injection Container

A DI container automatically creates objects, resolves their dependencies, and manages their lifetimes.

Responsibilities of a DI container:

- Register services
- Create objects
- Resolve dependencies recursively
- Manage object lifetimes
- Reuse instances when appropriate

```
builder.Services.AddTransient<IEmailService, EmailService>();
builder.Services.AddScoped<IProductRepository,
ProductRepository>();
builder.Services.AddSingleton<ILogger, Logger>();
```

5. Object Lifetimes

Lifetime	Meaning	New Instance	Typical Use
Transient	Created every time requested	Yes	Stateless helpers
Scoped	One instance per scope.	Per Scope	Repositories, DbContext
Singleton	One shared instance	No	Logging, configuration

Rule of thumb:

- Use Transient for lightweight, stateless services.
- Use Scoped for request specific services.
- Use Singleton only for shared, thread safe services.

6. Common Mistakes

- Creating dependencies using new inside services.
- Injecting too many dependencies into one class.
- Using Singleton for services that hold request specific data.
- Creating circular dependencies.
- Confusing IoC with Dependency Injection.

7. Best Practices

- Prefer constructor injection.
- Depend on interfaces rather than concrete classes when appropriate.
- Keep services focused on a single responsibility.
- Register services in one place (Program.cs).
- Choose the correct lifetime based on usage.

Chapter 3: Dependency Injection in ASP.NET Core (.NET)

1. Why ASP.NET Core Uses Dependency Injection

ASP.NET Core has a built-in Dependency Injection container. Instead of manually creating objects with the new keyword, services are registered once, and the framework injects them wherever they are required. This keeps code loosely coupled and easier to maintain.

2. Project Structure

```
Controllers/
    ProductController.cs

Services/
    IProductService.cs
    ProductService.cs

Repositories/
    IProductRepository.cs
    ProductRepository.cs

Program.cs
```

3. Repository Layer

The repository is responsible for interacting with the data source.

```
public interface IProductRepository
{
    List<string> GetProducts();
}

public class ProductRepository : IProductRepository
{
    public List<string> GetProducts()
    {
        return new List<string>
        {
            "Laptop",
            "Phone",
            "Keyboard"
        };
    }
}
```

4. Service Layer

The service contains business logic. Notice that it receives the repository through constructor injection.

```
public interface IProductService
{
    List<string> GetProducts();
}

public class ProductService : IProductService
{
    private readonly IProductRepository _repository;

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

    public List<string> GetProducts()
    {
        return _repository.GetProducts();
    }
}
```

5. Controller Layer

The controller receives ProductService through Dependency Injection.

```
[ApiController]
[Route("api/products")]
public class ProductController : ControllerBase
{
    private readonly IProductService _service;

    public ProductController(IProductService service)
    {
        _service = service;
    }

    [HttpGet]
    public IActionResult Get()
    {
        return Ok(_service.GetProducts());
    }
}
```

6. Registering Services in Program.cs

The DI container must know which implementation should be created for each interface.

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

builder.Services.AddScoped<IProductRepository,
ProductRepository>();
builder.Services.AddScoped<IProductService, ProductService>();

builder.Services.AddControllers();

var app = builder.Build();

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

7. How Dependency Injection Works

```
HTTP Request

↓

Controller

↓

Service

↓

Repository

↓

DbContext

↓

SQL Server
```

When a request reaches ProductController, ASP.NET Core sees that it needs IProductService. It creates ProductService, notices that ProductService needs IProductRepository, creates ProductRepository, injects it into ProductService, and finally injects ProductService into ProductController.

8. What the DI Container Does

- Stores service registrations.
- Creates objects only when needed.
- Resolves constructor dependencies automatically.

- Manages service lifetimes (Transient, Scoped, Singleton).
- Reuses instances according to the configured lifetime.

9. Real World Example

Imagine an e-commerce application.

```
OrderController
        |
        ▼
OrderService
        ├── IOrderRepository
        ├── IEmailService
        ├── ILogger<OrderService>
        └── IPaymentGateway
```

OrderService focuses only on business logic. The framework injects all required services, allowing each implementation to be replaced independently.

10. Why Interfaces?

Without interface

ProductService

↓

ProductRepository

With interface

ProductService

↓

IProductRepository

↓

ProductRepository

Benefits

- Loose coupling
- Easy replacement
- Mocking
- Different implementations

10. Benefits in Real Projects

Without DI	With DI
Uses new throughout the code	Centralized service registration
Hard to swap implementations	Easy to replace implementations
Difficult to maintain	Cleaner architecture
Tightly coupled	Loosely coupled
More boilerplate	The framework manages object creation

Chapter Summary

ASP.NET Core's built-in Dependency Injection container automatically creates and injects dependencies based on the services registered in Program.cs. Controllers receive services, services receive repositories, and each layer focuses only on its own responsibility. This results in a cleaner, more maintainable, and scalable application.

Chapter 4: Interview Questions, Revision & Cheat Sheet

Top 25 Interview Questions

1. What is Dependency Injection?

Answer: A design pattern where a class receives its dependencies from an external source instead of creating them.

2. What is a dependency?

Answer: Any object another object requires to perform its work.

3. Why is DI used?

Answer: To reduce coupling, improve maintainability, and simplify testing.

4. What is tight coupling?

Answer: When a class directly depends on a concrete implementation.

5. What is loose coupling?

Answer: When a class depends on abstractions and receives dependencies externally.

6. What is Constructor Injection?

Answer: Dependencies are passed through the constructor. This is the preferred approach.

7. What is Setter Injection?

Answer: Dependencies are provided through a property or setter method.

8. What is Method Injection?

Answer: Dependencies are passed only to the method that needs them.

9. What is IoC?

Answer: A principle where object creation and control are delegated to an external mechanism.

10. DI vs IoC?

Answer: DI is one way to achieve IoC.

11. What is a DI Container?

Answer: A framework component that creates objects and resolves dependencies.

12. What is a Composition Root?

Answer: The place where dependencies are registered and wired together.

13. Where is the Composition Root in ASP.NET Core?

Answer: Typically in Program.cs.

14. What is Transient lifetime?

Answer: A new instance is created every time it is requested.

15. What is Scoped lifetime?

Answer: One instance is created per HTTP request.

16. What is Singleton lifetime?

Answer: A single shared instance exists for the application's lifetime.

17. When should Singleton be avoided?

Answer: For services that store request specific or mutable state.

18. Why prefer Constructor Injection?

Answer: It guarantees required dependencies are available and makes them explicit.

19. DI vs Service Locator?

Answer: DI pushes dependencies into a class. Service Locator makes the class pull them.

20. Can DI work without interfaces?

Answer: Yes. Interfaces are helpful but not mandatory.

21. Can DI exist without a container?

Answer: Yes. Dependencies can be injected manually.

22. What happens if a dependency isn't registered?

Answer: ASP.NET Core throws a runtime exception when resolving it.

23. Can one interface have multiple implementations?

Answer: Yes.

ASP.NET Core allows multiple implementations to be registered. You can inject IEnumerable<IService> to receive all implementations or resolve a specific one using keyed services (available in newer .NET versions) or custom selection logic.

24. What are the main benefits of DI?

Answer: Loose coupling, flexibility, maintainability, scalability, and easier testing.

25. Which injection type is recommended in .NET?

Answer: Constructor Injection.

Quick Revision Notes

- Dependency: An object required by another object.
- Dependency Injection: Providing dependencies from outside the class.
- Goal: Reduce coupling between classes.
- Preferred injection type: Constructor Injection.
- ASP.NET Core has a built-in DI container.
- Register services in Program.cs.
- Controllers receive Services; Services receive Repositories.
- DI is a design pattern. IoC is a design principle.
- Composition Root is where dependencies are wired together.
- Choose the correct lifetime based on the service's purpose.

Comparison Table

Concept	Meaning	Remember
Dependency	Required object	Needed to perform work
DI	Provides dependencies	Reduces coupling
IoC	Transfers control	DI implements IoC
Transient	New instance	Stateless services
Scoped	One per request	Repositories, DbContext
Singleton	One application instance	Configuration, logging

Final Cheat Sheet

✓ Dependency = Object needed by another object.
✓ DI = Receive dependencies instead of creating them.
✓ Constructor Injection is the recommended approach.
✓ Register services in Program.cs.
✓ DI Container creates and injects dependencies automatically.

✓ Transient = New every resolution.
✓ Scoped = One per request.
✓ Singleton = One for the application lifetime.
✓ DI improves flexibility, maintainability, and scalability.
✓ DI is one implementation of IoC.