**Dependency Injection (DI)**

Dependency Injection (DI) es un patrón de diseño que facilita la gestión de dependencias en una aplicación. Su objetivo principal es desacoplar los componentes del sistema, permitiendo que las dependencias sean inyectadas en lugar de ser creadas internamente. Esto mejora la mantenibilidad, escalabilidad y testabilidad del código.

* **Principios Fundamentales**
* **Inversión de control (IoC):** En lugar de que un componente cree sus propias dependencias, estas son proporcionadas externamente.
* **Bajo acoplamiento:** Al desacoplar las dependencias, los módulos pueden evolucionar de manera independiente.
* **Alta testabilidad:** Las dependencias pueden ser fácilmente simuladas (mocked) en pruebas unitarias.
* **Mantenibilidad:** Facilita la modificación y extensión del código sin afectar otras partes del sistema.
* **Formas de Inyección de Dependencias**

Existen tres formas principales de implementar Dependency Injection:

### **Inyección por Constructor (Constructor Injection)**

Se pasa la dependencia a través del constructor de la clase.

public class MiServicio

{

private readonly IRepositorio \_repositorio;

public MiServicio(IRepositorio repositorio)

{

\_repositorio = repositorio;

}

}

### **Inyección por Método o Propiedad (Setter Injection)**

La dependencia se asigna a través de un método o propiedad pública.

public class MiServicio

{

public IRepositorio Repositorio { private get; set; }

public void Ejecutar()

{

Repositorio?.ObtenerDatos();

}

}

### **Inyección por Interfaz (Interface Injection)**

La clase implementa una interfaz que define un método para recibir la dependencia.

public interface IInyectable

{

void SetRepositorio(IRepositorio repositorio);

}

public class MiServicio : IInyectable

{

private IRepositorio \_repositorio;

public void SetRepositorio(IRepositorio repositorio)

{

\_repositorio = repositorio;

}

}

# **Ejemplo Implementado en el Proyecto**

En el proyecto, se aplicó Dependency Injection para desacoplar los servicios y mejorar la testabilidad.

### **Registro de Servicios en Program.cs**

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

builder.Services.AddSingleton<IConfigurationService, ConfigurationService>();

builder.Services.AddSingleton<IFileService, FileService>();

var app = builder.Build();

Aquí se registra IConfigurationService y IFileService, permitiendo que sean inyectados en cualquier parte del proyecto.

### 

### 

### 

### 

### **Inyección de ILogger en FileService**

public class FileService : IFileService

{

private readonly ILogger<FileService> \_logger;

private readonly FileManager \_fileManager;

public FileService(ILogger<FileService> logger)

{

\_logger = logger;

\_fileManager = new FileManager("StoredFiles");

}

public void SaveFile(string fileName, string content)

{

try

{

\_fileManager.WriteToFile(fileName, content);

\_logger.LogInformation("File saved: {FileName}", fileName);

}

catch (Exception ex)

{

\_logger.LogError(ex, "Error saving file: {FileName}", fileName);

}

}

}

El uso de ILogger<T> nos permite registrar eventos sin necesidad de crear instancias manualmente.

* **Beneficios de DI en .NET**
* Facilita la reutilización de código al desacoplar componentes.
* Simplifica la administración de dependencias en aplicaciones grandes.
* Mejora la escalabilidad al permitir cambiar implementaciones sin modificar la lógica del negocio.
* Favorece las pruebas unitarias al facilitar el uso de mocks y stubs.