**Dependency Injection**

The DI container consists of two main components: the `ServiceCollection` and the `ServiceProvider`.

1. `ServiceCollection`:

- The `ServiceCollection` class is used to register services and dependencies in your application. It provides methods such as `AddTransient`, `AddScoped`, and `AddSingleton` to register services with different lifetimes.

- `AddTransient`: Creates a new instance of the service each time it is requested.

- `AddScoped`: Creates a new instance of the service once per scope. A scope typically corresponds to a request in a web application.

- `AddSingleton`: Creates a single instance of the service and reuses it throughout the application.

- You can also register your own custom services and dependencies using the `Add` method.

- The `ServiceCollection` acts as a container for all the registered services.

2. `ServiceProvider`:

- The `ServiceProvider` class is responsible for resolving the dependencies and creating instances of the requested services.

- It is built based on the `ServiceCollection` and provides the `GetRequiredService` and `GetService` methods to retrieve instances of registered services.

- The `GetRequiredService` method throws an exception if the requested service is not registered, while the `GetService` method returns `null` if the service is not found.

**1)Property DI-** When we supply the dependency of object to Property of the class then it is called as Property Injection.

public class Student

{

public int Id { get; set; }

public string Name { get; set; }

public string Standard { get; set; }

}

public interface IStudentDataAccessLayer

{

public List<Student> GetStudents();

}

public class StudentDataAccessLayer : IStudentDataAccessLayer

{

public List<Student> GetStudents()

{

List<Student> students = new List<Student>();

students.Add(new Student() { Id = 1, Name = "Bharati", Standard = "5th" });

students.Add(new Student() { Id = 2, Name = "HEMA", Standard = "7th" });

students.Add(new Student() { Id = 3, Name = "Neha", Standard = "8th" });

return students;

}

public class StudentBusinessLayer

{

private IStudentDataAccessLayer \_studentDataAccessLayer;

public IStudentDataAccessLayer studentData

{

set

{

\_studentDataAccessLayer = value;

}

get

{

if (\_studentDataAccessLayer == null)

throw new Exception("obj not passed");

else

return \_studentDataAccessLayer;

}

}

public List<Student> GetStudents()

{

return \_studentDataAccessLayer.GetStudents();

}

}

}

class Program

{

static void Main(string[] args)

{

StudentBusinessLayer businessLayer = new StudentBusinessLayer();

businessLayer.studentData = new StudentDataAccessLayer();

var result = businessLayer.GetStudents();

}

}

**2)Constructor DI**-When we supply the dependency of object to the constructor of the class then it is called as Constructor Injection.

using System.ComponentModel;

using System.Runtime.CompilerServices;

using static StudentDataAccessLayer;

public class Student

{

public int Id { get; set; }

public string Name { get; set; }

public string Standard { get; set; }

}

public interface IStudentDataAccessLayer

{

public List<Student> GetStudents();

}

public class StudentDataAccessLayer: IStudentDataAccessLayer

{

public List<Student> GetStudents()

{

List<Student> students = new List<Student>();

students.Add(new Student() { Id = 1, Name = "Bharati", Standard = "5th" });

students.Add(new Student() { Id = 2, Name = "HEMA", Standard = "7th" });

students.Add(new Student() { Id = 3, Name = "Neha", Standard = "8th" });

return students;

}

public class StudentBusinessLayer

{

private IStudentDataAccessLayer \_studentDataAccessLayer;

public StudentBusinessLayer(IStudentDataAccessLayer studentDataAccessLayer)

{

\_studentDataAccessLayer=studentDataAccessLayer;

}

public List<Student> GetStudents()

{

return \_studentDataAccessLayer.GetStudents();

}

}

}

class Program

{

static void Main()

{

IStudentDataAccessLayer dataAccessLayer = new StudentDataAccessLayer();

StudentBusinessLayer businessLayer = new StudentBusinessLayer(dataAccessLayer);

List<Student> students = businessLayer.GetStudents();

foreach (var student in students)

{

Console.WriteLine($"ID: {student.Id}, Name: {student.Name}, Standard: {student.Standard}");

}

Console.ReadLine();

}

}

**3)Method DI-** When we supply the dependency of object to the Method of the class then it is called as Method Injection.

public class Student

{

public int Id { get; set; }

public string Name { get; set; }

public string Standard { get; set; }

}

public interface IStudentDataAccessLayer

{

public List<Student> GetStudents();

}

public class StudentDataAccessLayer : IStudentDataAccessLayer

{

public List<Student> GetStudents()

{

List<Student> students = new List<Student>();

students.Add(new Student() { Id = 1, Name = "Bharati", Standard = "5th" });

students.Add(new Student() { Id = 2, Name = "HEMA", Standard = "7th" });

students.Add(new Student() { Id = 3, Name = "Neha", Standard = "8th" });

return students;

}

public class StudentBusinessLayer

{

public List<Student> GetStudents(IStudentDataAccessLayer \_studentDataAccessLayer)

{

return \_studentDataAccessLayer.GetStudents();

}

}

}

class Program

{

static void Main(string[] args)

{

StudentBusinessLayer businessLayer = new StudentBusinessLayer();

var result = businessLayer.GetStudents(new StudentDataAccessLayer());

}

}

**Mapster**

What is the function of Mapster?

As the name suggests, Mapster is a library that maps one object type to a different object type. It is a convention-based mapper that is easy to configure and use. Writing code to map one object to another can be very repetitive and boring. Because of this, Mapster frees us from writing error-prone boilerplate code.

public class Person

{

public string FirstName { get; set; }

public string LastName { get; set; }

public int Age { get; set; }

}

public class PersonDto

{

public string FullName { get; set; }

public int Age { get; set; }

}

}

class Program

{

static void Main(string[] args)

{

TypeAdapterConfig<Person, PersonDto>.NewConfig()

.Map(dest => dest.FullName, src => src.FirstName+" "+ src.LastName);

var person = new Person

{

FirstName = "Bharati",

LastName = "NH",

Age = 30

};

var personDto = person.Adapt<PersonDto>();

Console.WriteLine($"Full Name: {personDto.FullName}");

Console.WriteLine($"Age: {personDto.Age}");

Console.ReadLine();

}

}

}

**Object Mappers**

**AutoMapper:**AutoMapper in C# is a library used to map data from one object to another in web development. It acts as a mapper between two objects and transforms one object type into another.

public class Person

{

public string? Title { get; set; }

public string? FirstName { get; set; }

public string? LastName { get; set; }

public DateTime? DateOfBirth { get; set; }

public Address? Address { get; set; }

}

public class PersonDto

{

public string? Title { get; set; }

public string? FirstName { get; set; }

public string? LastName { get; set; }

public DateTime? DateOfBirth { get; set; }

}

public static class MapperConfig

{

public static IMapper InitializeAutomapper()

{

var config = new MapperConfiguration(cfg =>

**{**

**cfg.CreateMap<Person, PersonDto>();**

**});**

**return config.CreateMapper();**

**}**

**}**

**}**

**class Program**

**{**

**static void Main(string[] args)**

**{**

**Person person = new Person**

**{**

**Title = "Mr",**

**FirstName = "John",**

**LastName = "Doe",**

**DateOfBirth = new DateTime(1990, 5, 15),**

**Address = new Address**

**{**

**Street = "123 Main St",**

**City = "London",**

**PostCode = "SW1A 1AA",**

**Country = "UK"**

**}};**

**var mapper = MapperConfig.InitializeAutomapper();**

**PersonDto personDto = mapper.Map<PersonDto>(person);**

**Console.WriteLine("Title: " + personDto.Title);**

**Console.WriteLine("First Name: " + personDto.FirstName);**

**Console.WriteLine("Last Name: " + personDto.LastName);**

**Console.WriteLine("Date of Birth: " + personDto.DateOfBirth);**

**Console.ReadLine();**

**TypeAdapterConfig typeAdapterConfig = new TypeAdapterConfig();**

**var p1 = new Person();**

**//var p2 = new Person();**

**var newObj = p1.Adapt<PersonDto>();**

**}}}**

**Repository pattern**

**The repository pattern is a design pattern that isolates the data layer from the rest of the app. The data layer refers to the part of your app, separate from the UI, that handles the app's data and business logic, exposing consistent APIs for the rest of your app to access this data.**

**public interface IOrderRepository**

**{**

**Order Add(Order order);**

**// ...**

**}**

**public interface OrderRepository : IOrderRepository**

{

// ...

}

**HTTP Methods**

1. **GET:** It perform theRead Operation
2. **POST:** It perform the Write Operation
3. **PUT:** It perform the Update Operation
4. **HEAD:** The HEAD method asks for a response identical to that of a GET request, but without the response body.
5. **DELETE:** It perform the **Delete** Operation
6. **PATCH:** The PATCH HTTP method is used to modify the values of the resource properties.
7. **OPTIONS:** The HTTP OPTIONS method is used to describe communication options for the target resource.
8. **CONNECT:** The CONNECT method is for making end-to-end connections between a client and a server

**DDD Model**

In DDD, this notion is called model-driven design. Your understanding of the model is evolved in your code. Domain-driven designers would rather not bother with reams of documentation or heavy diagramming tools. They seek, instead, to imbue their sense of domain understanding directly into their code.

**Entity FrameWork**

Object Relational Mapping (ORM) is a technique used in creating a "bridge" between object-oriented programs and, in most cases, relational databases.

[Dapper](https://www.learndapper.com/) is a micro-ORM. It is mainly used to map queries to objects. This tool doesn't do most of the things an ORM tool would do like SQL generation, caching results, lazy loading, and so on.

**What are the Approaches to Migrate Data to DB**

1. **ADO.NET**: ADO.NET is a core data access technology in .NET that allows you to interact with databases directly. You can use the **SqlConnection** class to connect to the source data (e.g., CSV, Excel, another database), read the data using **DataReader** or **DataAdapter**, and then insert or update the data in the target database using SQL commands.
2. **Entity Framework (EF)**: Entity Framework is an Object-Relational Mapping (ORM) framework that simplifies data access in .NET applications. It allows you to work with databases using strongly-typed .NET objects, and it can handle the migration of data from one database to another. You can use EF's **DbContext** to read data from the source database, map it to entities, and then save the entities to the target database.
3. **SQL Server Integration Services (SSIS)**: If you are working with SQL Server as your source and target databases, you can use SQL Server Integration Services (SSIS) to create data migration packages. SSIS provides a visual development interface to design data flows and transformations, making it easier to migrate large volumes of data between databases.