CS6004NT
Application Development

WEEK - 03

C# Advanced Topics

II

Reflection and Attributes

Reflection

Reflection is the ability of a program to examine its own metadata **at runtime**.

1. Load assemblies and types

```
Assembly assembly = Assembly.Load("MyAssembly");
Type type = assembly.GetType("MyNamespace.MyClass");
object instance = Activator.CreateInstance(type);
```

5. Inspect and manipulate objects

```
Type type = myObject.GetType();
PropertyInfo property = type.GetProperty("MyProperty");
object value = property.SetValue(myObject);
```

9. Create new objects

```
Type type = typeof(MyClass);
object instance = Activator.CreateInstance(type);
```

Reflection

4. Invoke methods:

```csharp
Type type = myObject.GetType();
MethodInfo method = type.GetMethod("MyMethod");
method.Invoke(myObject, null);
```

4. Access properties and fields

```csharp
Type type = myObject.GetType();
PropertyInfo property = type.GetProperty("MyProperty");
object propertyValue = property.GetValue(myObject);
FieldInfo field = type.GetField("MyField");
object fieldValue = field.GetValue(myObject);
```

4. Attribute inspection

```csharp
Type type = typeof(MyClass);
object[] attributes = type.GetCustomAttributes(typeof(MyAttribute), false);
```

Attributes

- Attributes are declarative tags that provide **additional information about code**.
- They are used to add **metadata to types, methods, properties**, and other elements of code.
- They can be used to annotate code with **information that can be used by Reflection**.

Examples

```
using System;
using System.Attribute;
using System.Reflection;

[AttributeUsage(AttributeTargets.Class, AllowMultiple = false)]
public class TableAttribute : Attribute
{
    public TableAttribute(string name)
    {
        Name = name;
    }
    public string Name { get; }
}

[Table("Workers")]
public class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

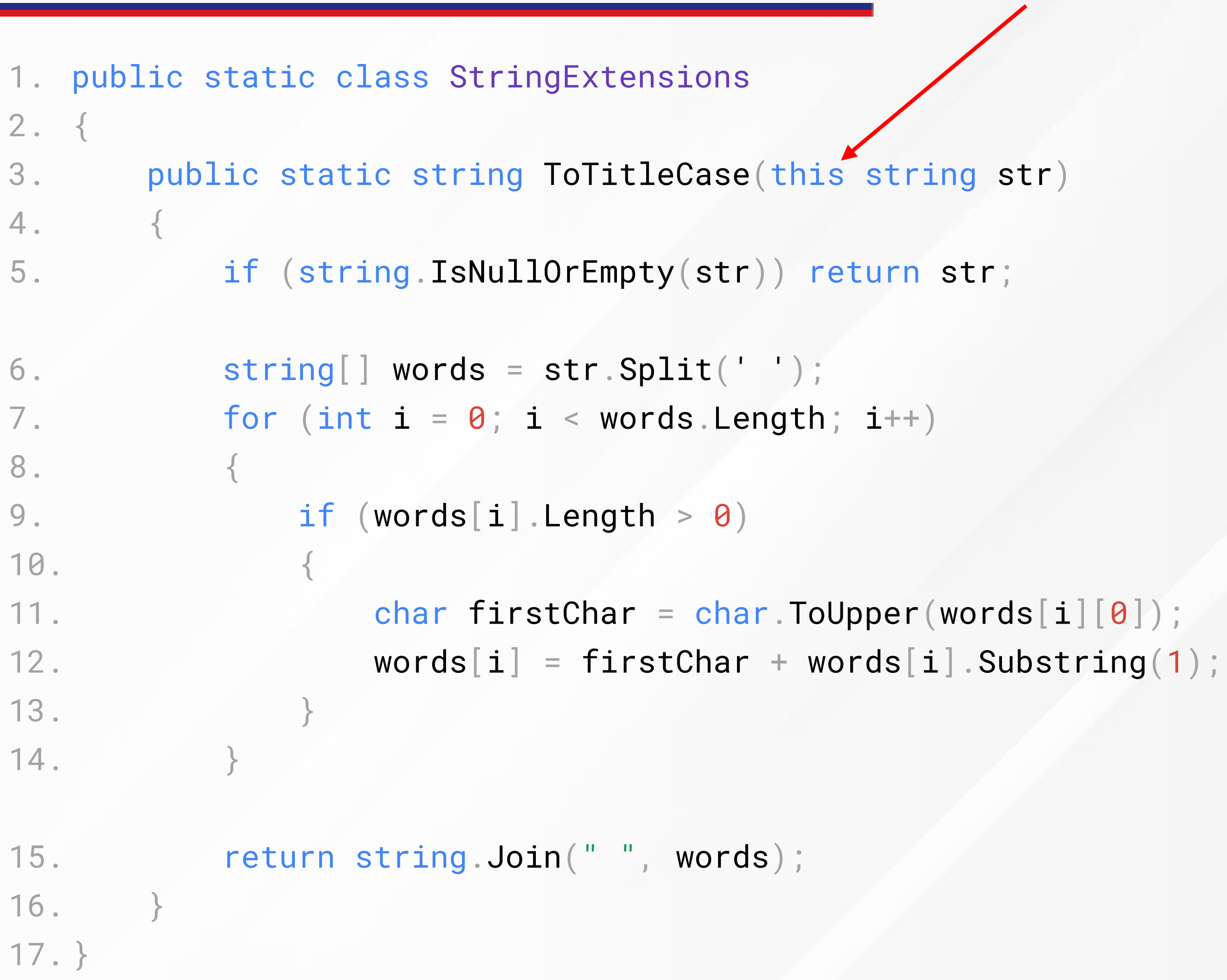
```csharp
public static class DbUtilities
{
    public static void CreateTable<T>()
    {
        Type type = typeof(T);

        // Check if the class is decorated with the TableAttribute
        if (type.IsDefined(typeof(TableAttribute), true))
        {
            var tableAttr = (TableAttribute)type.GetCustomAttribute(typeof(TableAttribute), true);
            string tableName = tableAttr.Name;
            Console.WriteLine($"Creating table named '{tableName}' in the database.");
        }
        else
        {
            throw new Exception($"The class '{type.Name}' is missing the TableAttribute.");
        }
    }
}
```

```csharp
DbUtilities.CreateTable<Employee>();
// Creating table named 'Workers' in the database.
```

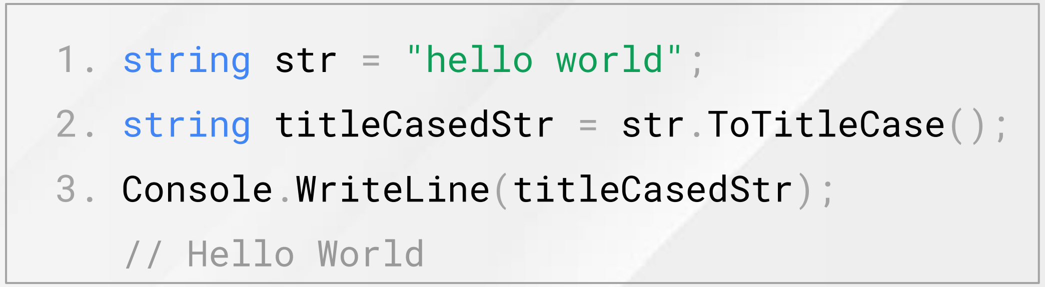
Extension Methods

- An extension method allows us to **add new static methods to an existing class or interface** without modifying its original source code.
- Extension methods are defined in a **separate static class**, and they must be marked with the **this keyword before the first parameter**, indicating which type the method should be available on.

Example



```
public static class StringExtensions
{
    public static string ToTitleCase(this string str)
    {
        if (string.IsNullOrEmpty(str)) return str;

        string[] words = str.Split(' ');
        for (int i = 0; i < words.Length; i++)
        {
            if (words[i].Length > 0)
            {
                char firstChar = char.ToUpper(words[i][0]);
                words[i] = firstChar + words[i].Substring(1);
            }
        }

        return string.Join(" ", words);
    }
}
```



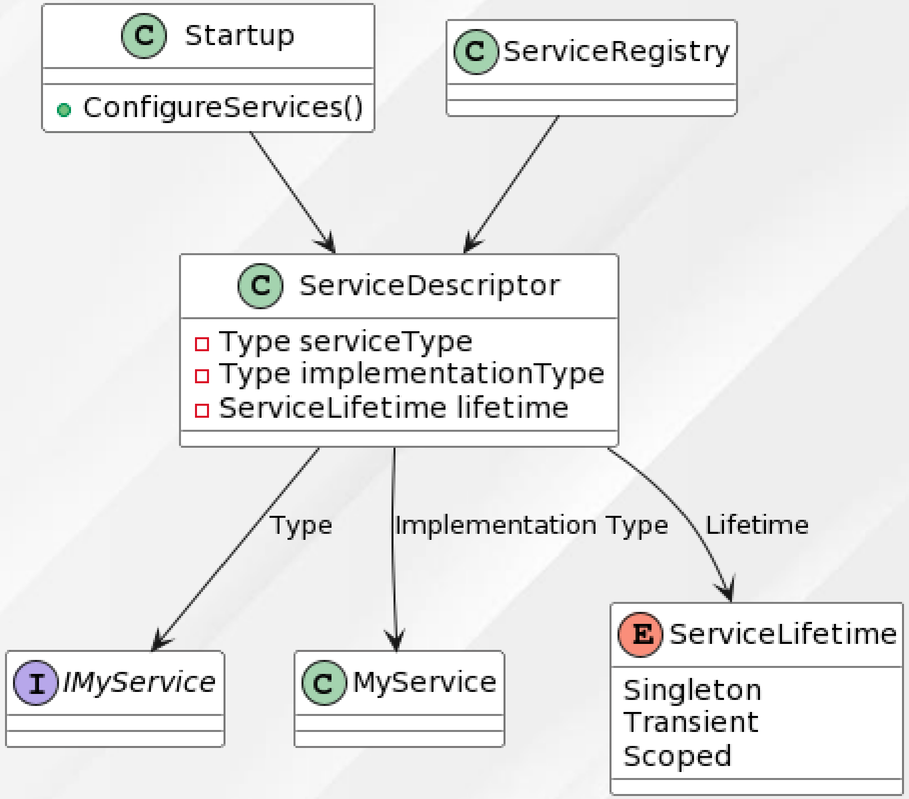
```
string str = "hello world";
string titleCasedStr = str.ToTitleCase();
Console.WriteLine(titleCasedStr);
// Hello World
```

Dependency Injection

- Dependency Injection (DI) is a design pattern used to **remove dependencies between objects**, making code more modular, reusable, and easier to maintain.
- It is a way of **injecting dependencies into a class** or component from an external source, **rather than creating those dependencies inside** the class or component itself.
- DI promotes loose coupling between objects, making code more flexible and easier to change.
- It also allows for easier unit testing, as dependencies can be mocked or replaced with test doubles.

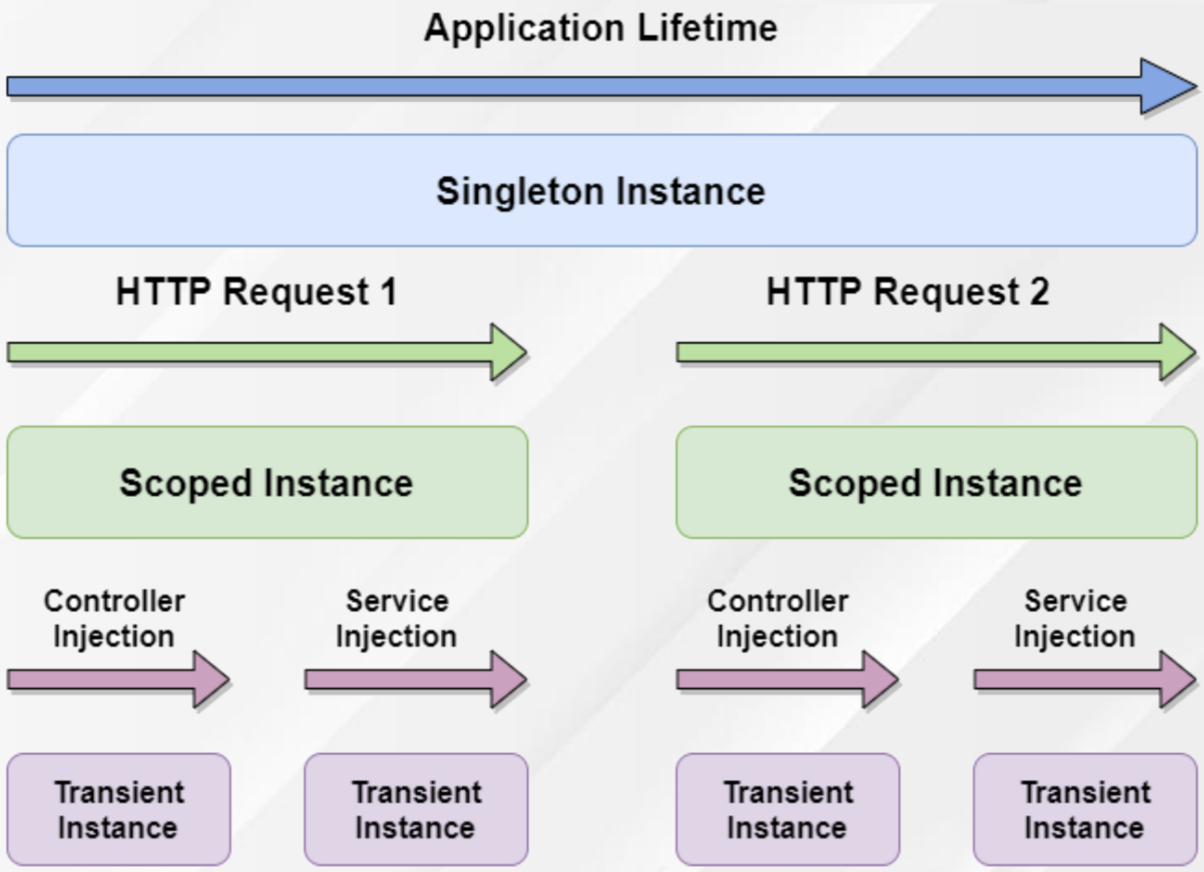
Service Registration

- The Startup/Program is used to register services with the **DI container**.
- **ServiceDescriptors** represent registered services in the DI container.
- **ServiceRegistry** is used to store ServiceDescriptors.



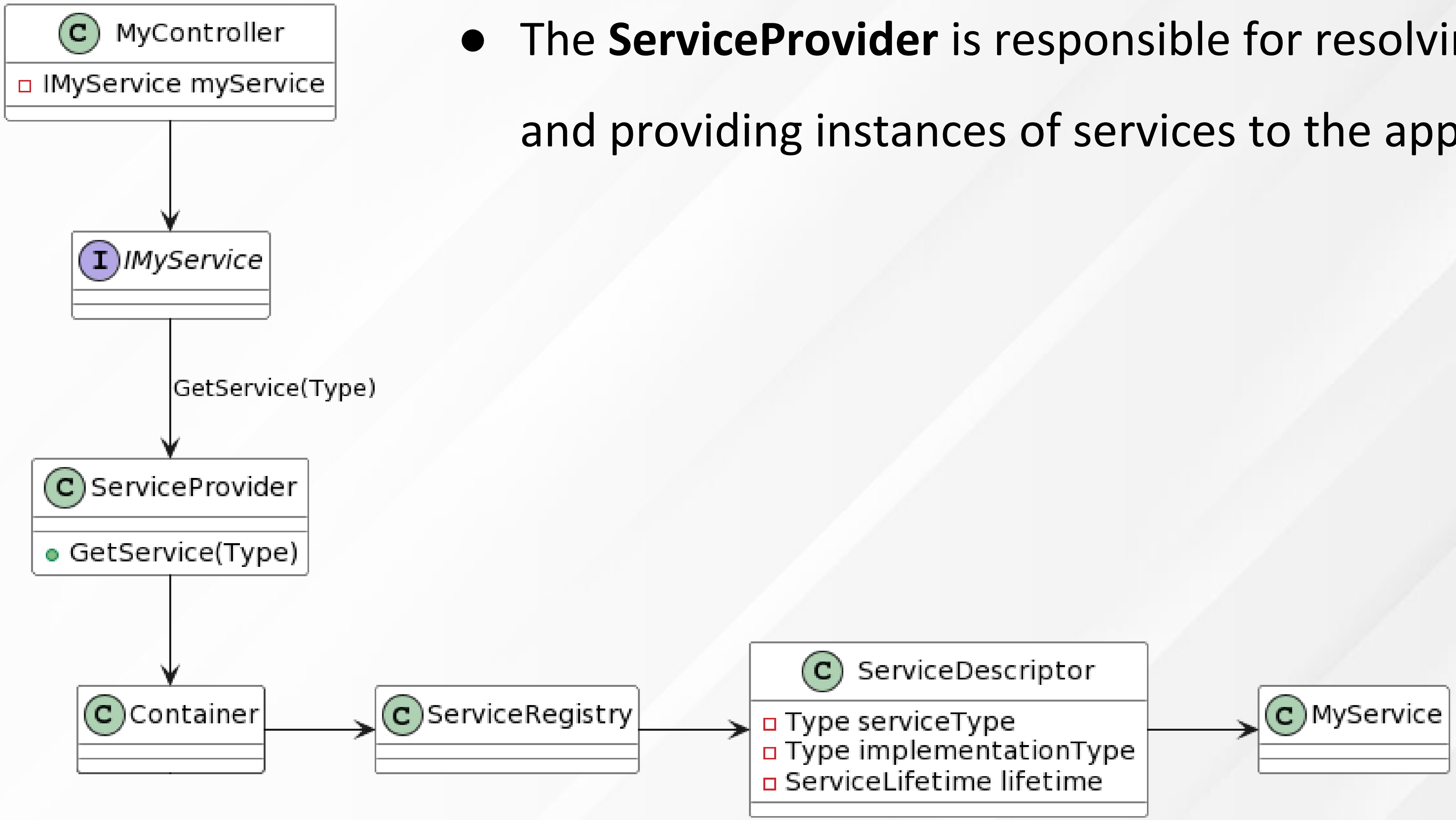
Service Registration

- **Service Instance lifetimes**:
  - **Singleton:** A single instance of the service is created and used for the lifetime of the application.
  - **Scoped:** A new instance of the service is created for each scope.
  - **Transient:** A new instance of the service is created each time it is requested.



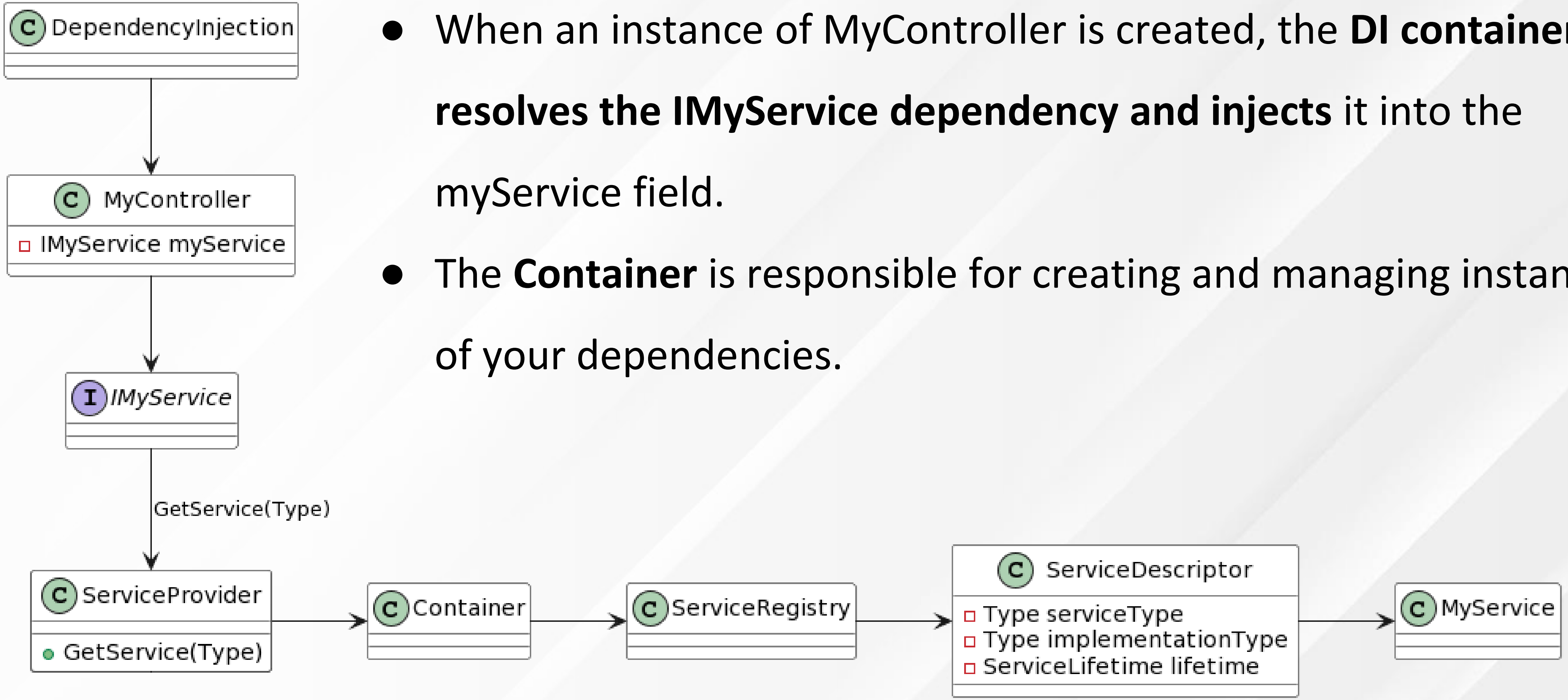
Service Resolution

- The **ServiceProvider** is responsible for resolving dependencies and providing instances of services to the application.



Service Injection

- When an instance of MyController is created, the **DI container resolves the IMyService dependency and injects** it into the myService field.
- The **Container** is responsible for creating and managing instances of your dependencies.



Example

```
public class MyService
{
    private readonly MyDependency _dependency;

    public MyService(MyDependency dependency)
    {
        _dependency = dependency;
    }

    public void DoSomething()
    {
        _dependency.DoSomethingElse();
    }
}
```

```
public class MyDependency
{
    public void DoSomethingElse()
    {
        Console.WriteLine("Doing something else...");
    }
}
```

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

// Register services
builder.Services.AddScoped<MyDependency>();
builder.Services.AddScoped<MyService>();

var app = builder.Build();
app.MapGet("/", (MyService myService) =>
myService.DoSomething());
app.Run();
```

Program.cs

Web API

Web API

REST API

REST API (**Representational State Transfer Application Programming Interface**) is a widely used standard for building web-based applications using **HTTP requests and responses**.

RESTful constraints:

1. **Client-Server Architecture:** the client and server are separate components that communicate through HTTP requests and responses.
2. **Statelessness:** each request must contain all of the information needed for the server to understand and respond to it.

REST API

3. **Cacheability:** Responses should include information on whether they can be cached or not, and clients should be able to cache them.
4. **Layered System:** this means that intermediate servers such as load balancers or gateways can be used without affecting the interface between the client and the server.
5. **Uniform Interface:** The interface should be consistent and should use HTTP methods and URIs to interact with resources.
6. **Code on Demand:** It is an optional constraint that allows the server to send code to the client. For example, a RESTful service might send a script to the client that can be used to display data in a certain way.

REST API

HTTP Methods:

HTTP methods are used to perform CRUD (Create, Read, Update, Delete) operations on resources.

1. **GET:** Retrieves resource information.
2. **POST:** Creates new resources.
3. **PUT:** Updates existing resources.
4. **DELETE:** Deletes resources.

URIs (Uniform Resource Identifiers):

URIs are used to identify resources in the system.

1. URI should contain **only nouns, and no verbs or actions**.
2. URI should use **plural nouns for collections and singular for individual** resources.

Minimal APIs in .NET

Minimal APIs is a new feature introduced in .NET 6, which is a lightweight and simplified way to create APIs. It is suitable for small to medium-sized projects.

Minimal APIs project using the .NET CLI:

1. Open a terminal or command prompt and navigate to the directory where you want to create the project.
2. Run the following command to create a new .NET 6 Minimal APIs project:

   ```
   dotnet new web -o MyMinimalApi --framework net6.0
   ```

1. Navigate into the project directory using the following command:

   ```
   cd MyMinimalApi
   ```

1. Run the following command to restore the project dependencies:

   ```
   dotnet restore
   ```

1. Run the project using the following command:

   ```
   dotnet run
   ```

Minimal APIs in .NET

Understanding Minimal APIs project:

1. **MyMinimalApi.csproj:** This file contains the project configuration, including the list of dependencies and the build settings.
2. **Program.cs:** This file is the entry point of the application. It contains the Main method, which sets up the web host and runs the application.
3. **Properties:** This directory contains the launch settings for the application, such as the ports to use and any environment variables that need to be set.
4. **appsettings.json:** This file contains the application configuration settings, such as connection strings and other settings that can be changed without modifying the application code.
5. **appsettings.Development.json:** This file contains the development-specific configuration settings, which override the values in the appsettings.json file.
6. **obj:** This directory contains the intermediate files generated during the build process..
7. **bin:** This directory contains the compiled executable files and any other files generated during the build process.

Minimal APIs in .NET

Understanding Program.cs:

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

➔ *Initializes a new instance of the WebApplicationBuilder class.*

```csharp
var app = builder.Build();
```

➔ *Creates an instance of the WebApplication class which is used to configure the HTTP pipeline, and routes.*

```csharp
app.MapGet("/", () => "Hello World!");
```

➔ *Maps a GET HTTP request to the root endpoint "/" of the web application, and returns the string "Hello World!" in the response.*

```csharp
app.Run();
```

➔ *Runs the application by starting the web host and listening for incoming requests.*

Controller Based APIs in .NET

Controller Based APIs are web APIs that use controllers for handling web API requests. Controllers are classes that derive from ControllerBase class.

Controller based APIs project using the .NET CLI:

1. Open a terminal or command prompt and navigate to the directory where you want to create the project.
2. Run the following command to create a new .NET 6 Controller based APIs project:

```
dotnet new webapi -n MyControllerApi --framework net6.0
```

1. Navigate into the project directory using the following command:

```
cd MyControllerApi
```

1. Run the following command to restore the project dependencies:

```
dotnet restore
```

1. Run the project using the following command:

```
dotnet run
```

Controller Based APIs in .NET

Understanding Controllers:

```
[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
{
    private static readonly string[] Summaries = new[] {"Freezing", "Bracing", ...};
    private readonly ILogger<WeatherForecastController> _logger;

    public WeatherForecastController(ILogger<WeatherForecastController> logger)
    {
        _logger = logger;
    }

    [HttpGet(Name = "GetWeatherForecast")]
    public IEnumerable<WeatherForecast> Get()
    {
        // code that returns random weather forecasts
    }
}
```

Controller Based APIs in .NET

What’s new Program.cs:

1. `builder.Services.AddControllers();`

   ➔ *Registers the controllers with the dependency injection system. This enables the application to inject dependencies into the controller class constructor using constructor injection.*

1. `builder.Services.AddEndpointsApiExplorer();`

   ➔ *Adds the endpoint API explorer to the application's services. This service generates the OpenAPI specification for the application's endpoints.*

1. `builder.Services.AddSwaggerGen();`

   ➔ *This line adds the Swagger generator to the application's services. The Swagger generator uses the OpenAPI specification to generate the Swagger UI.*

Controller Based APIs in .NET

What’s new Program.cs:

4. `app.UseSwagger();`
5. `app.UseSwaggerUI();`

   ➔ *The application uses the Swagger middleware to generate the Swagger UI and make it available at the default route of "/swagger".*

4. `app.UseHttpsRedirection();`

   ➔ *Redirects all HTTP requests to HTTPS.*

4. `app.UseAuthorization();`

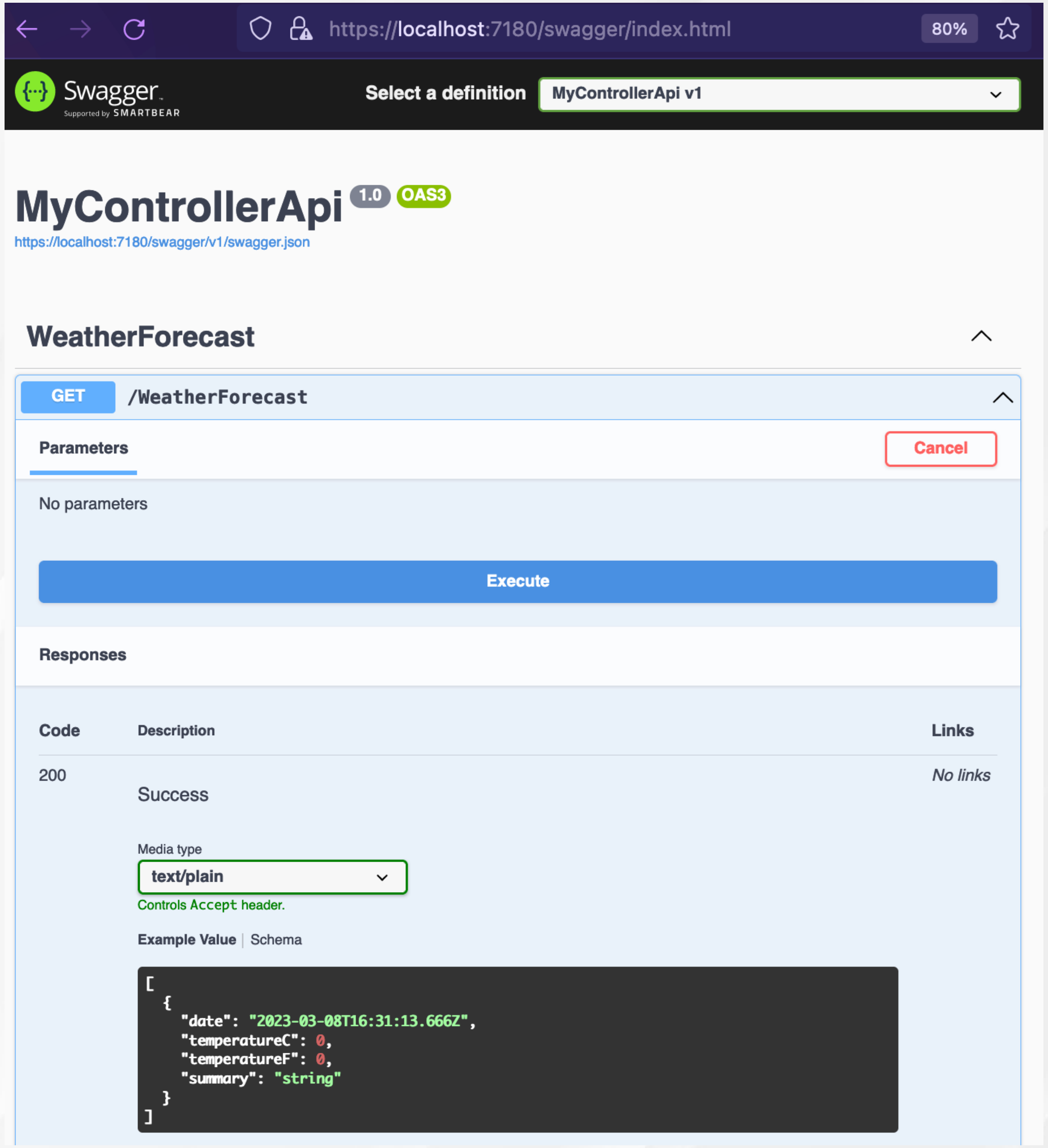
   ➔ *This line enables authorization middleware, which validates that the incoming request has the appropriate authentication and authorization.*

4. `app.MapControllers();`

   ➔ *This line maps the incoming request to the appropriate controller action method based on the route defined in the [Route] attribute of the controller and action methods.*

Swagger UI

1. **Interactive API documentation:** This documentation includes descriptions of each endpoint, the parameters it accepts, and the responses it returns.
2. **Easy testing:** This makes it easy to quickly test directly from the documentation page and iterate on API changes.
3. **Improved collaboration:** Easy to share the API documentation with other developers or stakeholders.
4. **Consistency and standardization:** It enforces the use of standardized parameter and response formats, and provides a consistent UI for interacting with your API.



Select a definition: MyControllerApi v1

MyControllerApi 1.0 OAS3

https://localhost:7180/swagger/v1/swagger.json

WeatherForecast

GET /WeatherForecast

No parameters

Responses

Code	Description	Links
200	Success	*No links*

Media type: text/plain

Controls Accept header.

Example Value | Schema

```
[
  {
    "date": "2023-03-08T16:31:13.666Z",
    "temperatureC": 0,
    "temperatureF": 0,
    "summary": "string"
  }
]
```

Thank You