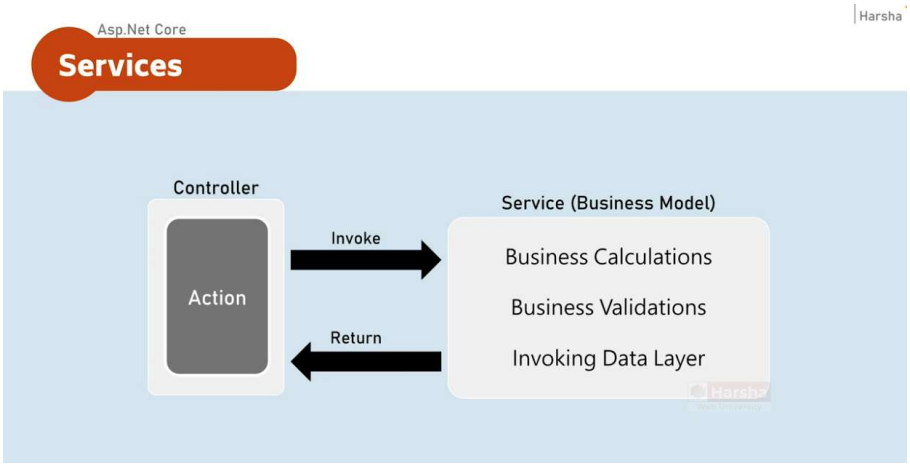
Section 12: Dependency Injection [MVC and Web API]

Friday, January 17, 2025 10:34 PM



Where Should You Write Business Logic?
Business logic should not be written in the **controller** or the **view**. Instead, it should reside in the **service layer**, also known as the **business model**.

What is Business Logic?

Business logic refers to the core logic that deals with:

- **Business calculations**
- **Business validations**
- **Invoking the data layer**

It is specific to the domain and requirements of the client's business. For example:

- In an **insurance application**, core logic includes calculating the premium amounts for policies or the claims amounts.
- In addition, validating conditions like the minimum age required to purchase an insurance policy.

Such domain-specific calculations and validations are collectively referred to as **business logic**. While it may involve calling the data layer, the **service layer** itself is not the **data layer**. The data layer remains a separate entity, and it is invoked by the service layer as needed.

How Business Logic Flows in a Web Application

For example, in a **simple registration form**:

1. **User Interaction**: The user fills in details such as username, password, email, etc., and clicks the **Submit** button.
2. **Controller's Role**: The browser sends a **POST request** to a **controller action method**.
3. **Service Layer's Role**: The controller calls a method in the **service class**, where the following logic is performed:
    - Calculating the user's age based on their date of birth.

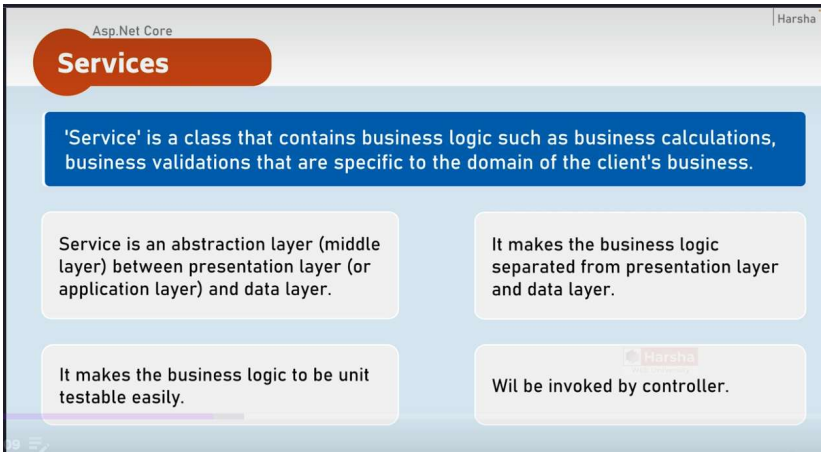
- Checking if the user meets the eligibility criteria (e.g., age requirement).
- Validating the email address and phone number.
- Generating a unique ID for the new user.

4. **Data Layer's Role**: Once all the business logic is complete, the service layer invokes the **data layer** to store the processed registration details in the database.

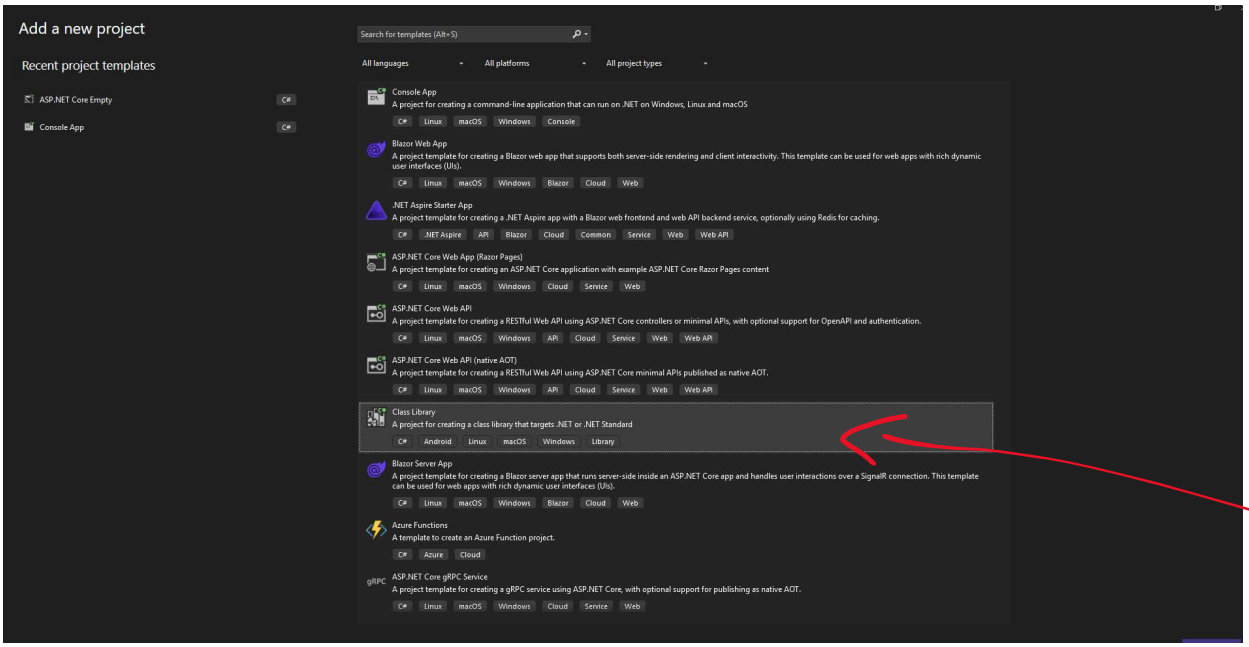
Key Notes:

- The **business logic** should only focus on processing data related to the specific domain of the client's business.
- **Database-related logic** belongs strictly in the **data layer**, not in the **service layer**.
- The **service layer** acts as an intermediary between the **presentation layer** (e.g., views) and the **data layer**, ensuring clean separation of concerns.

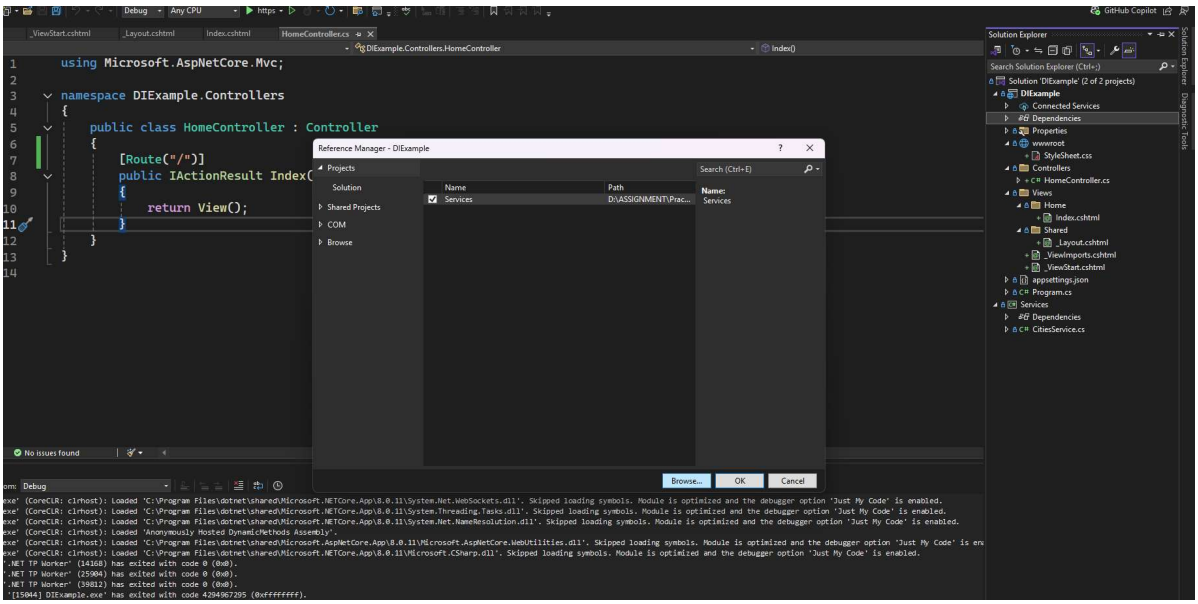
This separation ensures maintainability, scalability, and clarity in your application architecture.

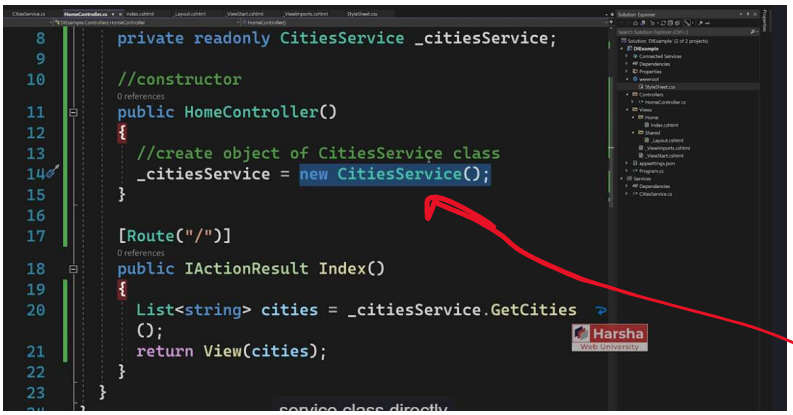


Services can be build independently. So It is wise to create separate class library project for services

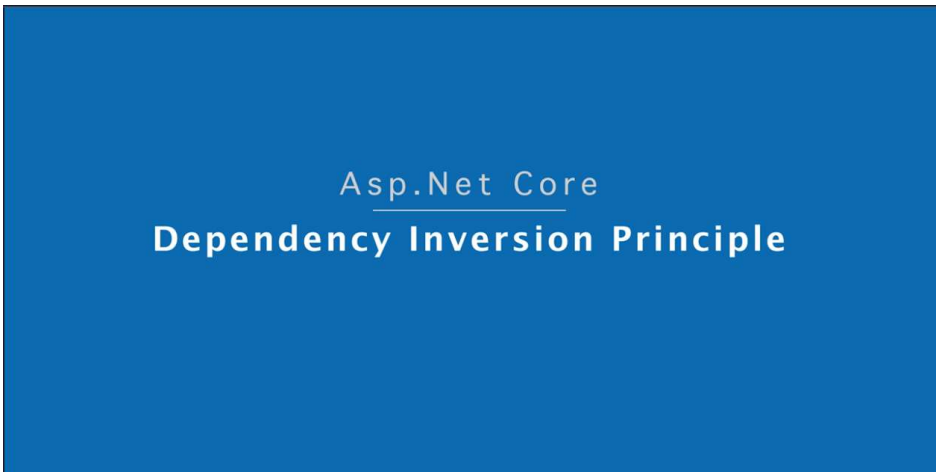


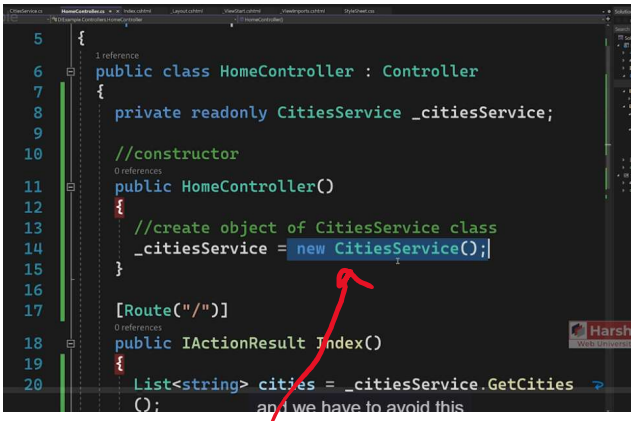
Add Project Reference here





You should never create an object of Service class Directly like this!





In this entire application, this is the most problematic statement. We have to avoid this. But why??

There are 2 reasons.

For example, let's say in your real-world project, the developer of the city service has not developed it yet, so there is no CityService class.
In this case, how can you create an object of the CityService class? It's not possible, right?
The developer of the controller must wait until the service development is completed.

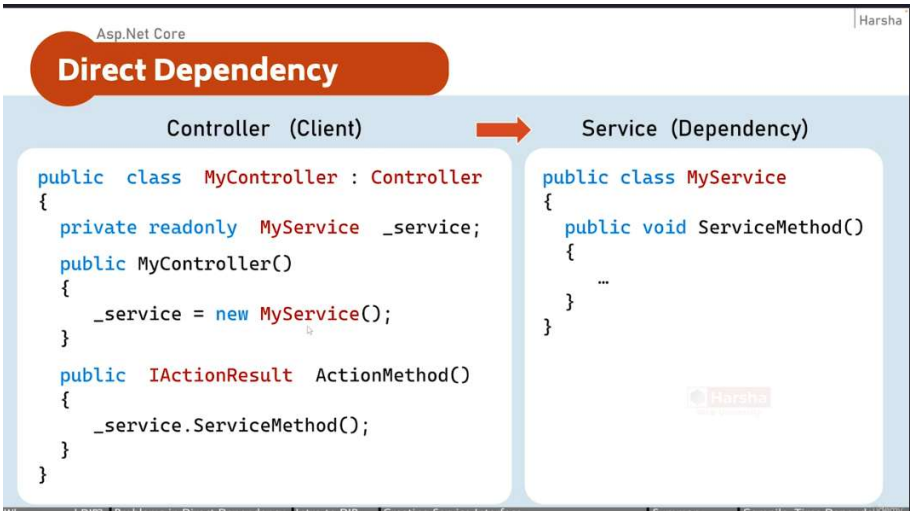
Problems with This Approach:

1. **Dependency on Service Development**:
   The controller depends on the service. The controller developer must wait until the service development is finished.

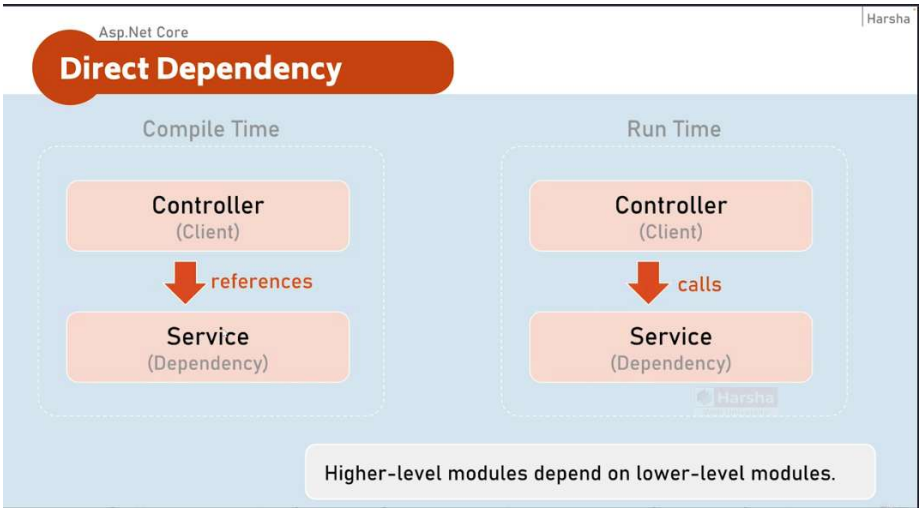
2. **Difficulty in Switching Service Classes**:
   - If you want to use a different class instead of CityService, such as another service class called CityService2 (which retrieves data from a database), you need to change the class name everywhere.
   - For example, in a real-world project, if the same service class is used in over 100 controllers, you would need to change the class name in all those controllers.

This approach makes changing or upgrading services problematic in the future.
While this might not seem like a major issue in small practice-level applications, it becomes a significant problem in real-world projects.

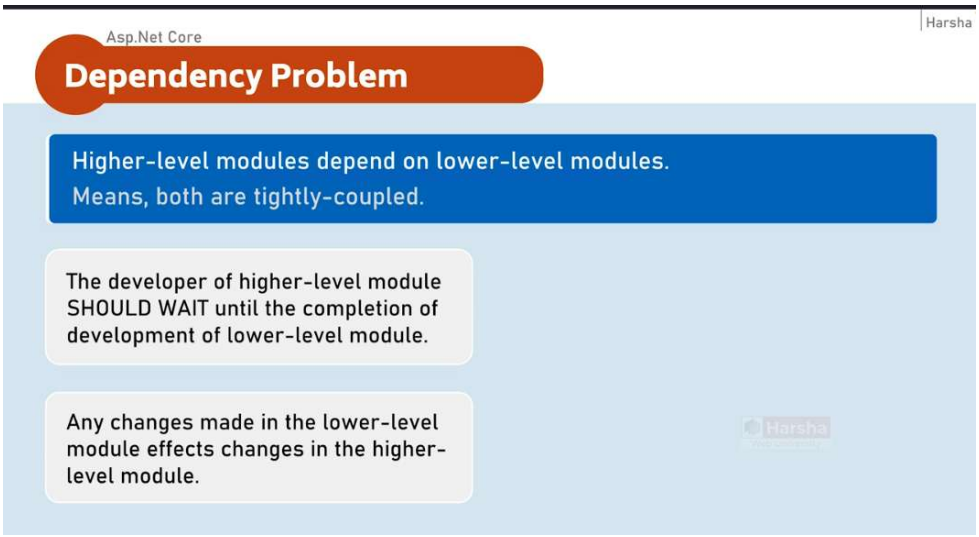
And the solution for this problem is 'Depedency Inversion Principle'.



In the existing code, the controller directly calls the service, meaning it creates an object of the MyService that was already developed.
This is called **direct dependency**, meaning the controller (which is also referred to as the client in this case) depends on the service.
Here, the client does not refer to the browser. Instead, the controller is called a client because it depends on another class that is service and the service class is called as dependency.



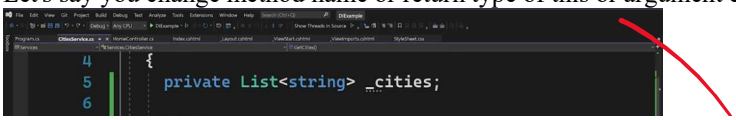
The controller directly depends on the service at both compilation time and runtime.
This means the controller class cannot be compiled if the service class is not defined or does not exist.
In other words, high-level modules depend on low-level modules.
Here, a module refers to a class—it can be a controller class, another service, or any other class that calls this service.
The higher-level module (the controller) depends on the service.
This is the problem.



So, what are the implications of that dependency?

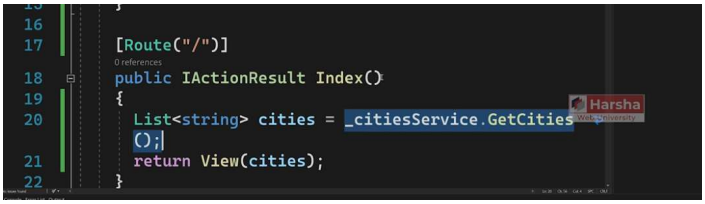
The developer of the higher-level module must wait until the compilation of the lower-level module.
This means the controller developer must wait until the service development is completed.
If there is a delay, the controller developer is forced to wait.
Another problem is that any changes made in the lower-level module directly affect the higher-level module.

Let's say you change method name or return type of this or argument etc will directly affect



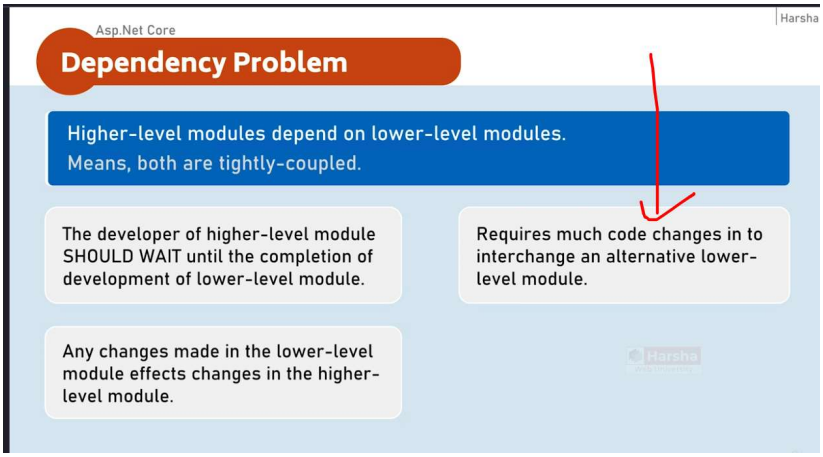
```
{
  private List<string> _cities;

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

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

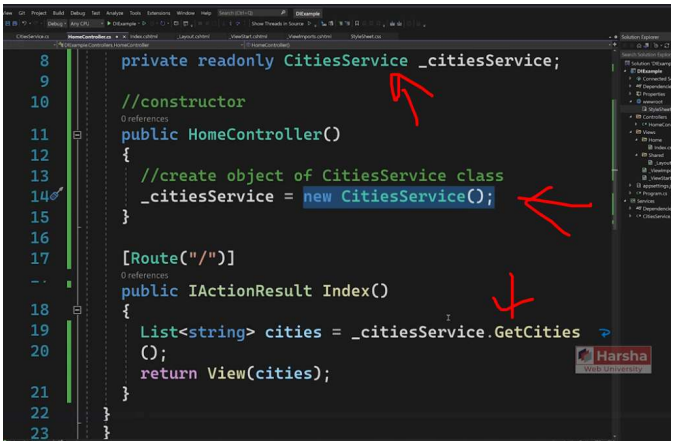
The calling portion here.



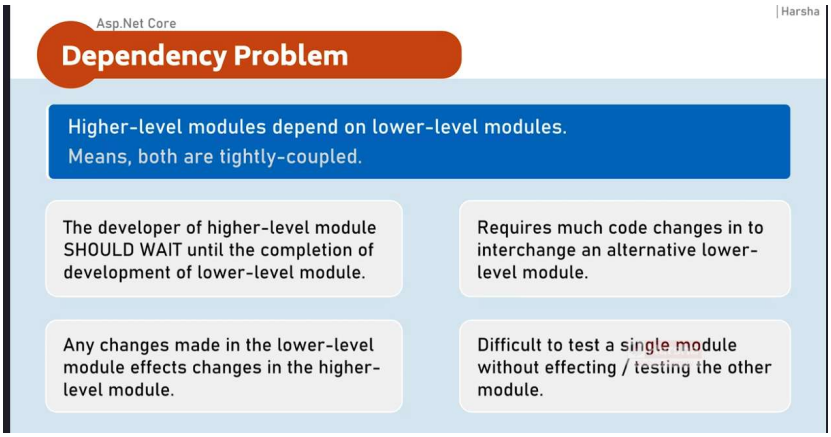
```
[Route("/")]
0 references
public IActionResult Index()
{
  List<string> cities = _citiesService.GetCities
  ();
  return View(cities);
}
```

Next Problem is when you want to use alternative service in a controller instead of existing service, it requires huge changes.

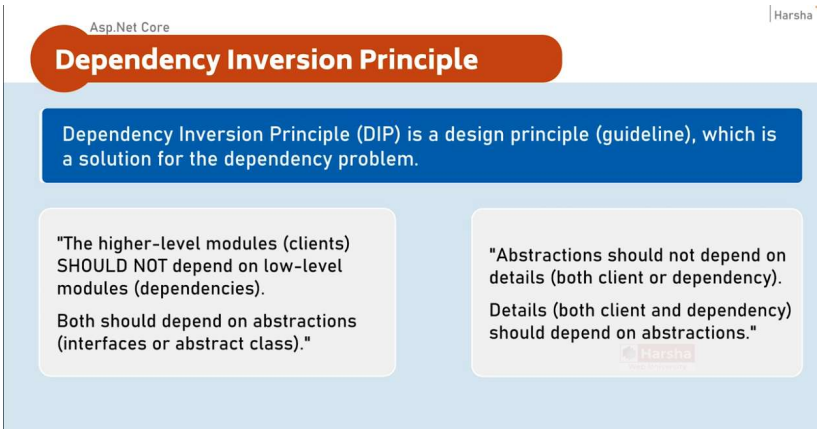




```
private readonly CitiesService _citiesService;

//constructor
0 references
public HomeController()
{
  //create object of CitiesService class
  _citiesService = new CitiesService();
}

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



Solution for this problem is 'Dependency Inversion Principle'

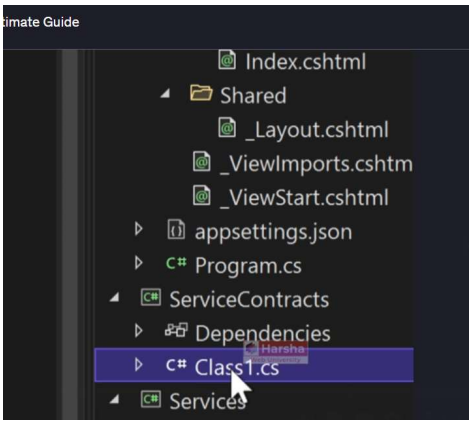


It is just a design principle, meaning it is a guideline that doesn't provide any predefined library or functionality or even tell you how to implement this principle.
The principle is just a guideline that outlines a set of rules but doesn't specify how to implement them.
To implement this, you need to use Inversion of Control (IoC).
We will learn about Inversion of Control in the next lecture, so hold on for now.
First, try to understand the Dependency Inversion Principle.
It states that higher-level modules should not depend on lower-level modules; both should depend on abstractions.
So, what does it mean?

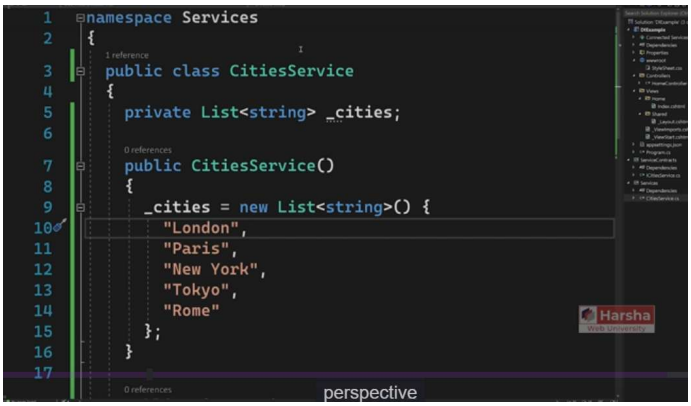
Both the client, meaning the controller class, and the service class, meaning the dependency class, should depend on abstractions, but neither of them should depend on the other directly.
For example, the HomeController should never depend on the service class directly but instead should depend on abstraction.
What is meant by abstraction here?
Abstraction refers to creating an interface or abstract class.
In most cases, interfaces are preferred because they are flexible and even support multiple inheritance.

Let's create interface that contains the list of methods that must be implemented by the service. Here the interface is being developed by the

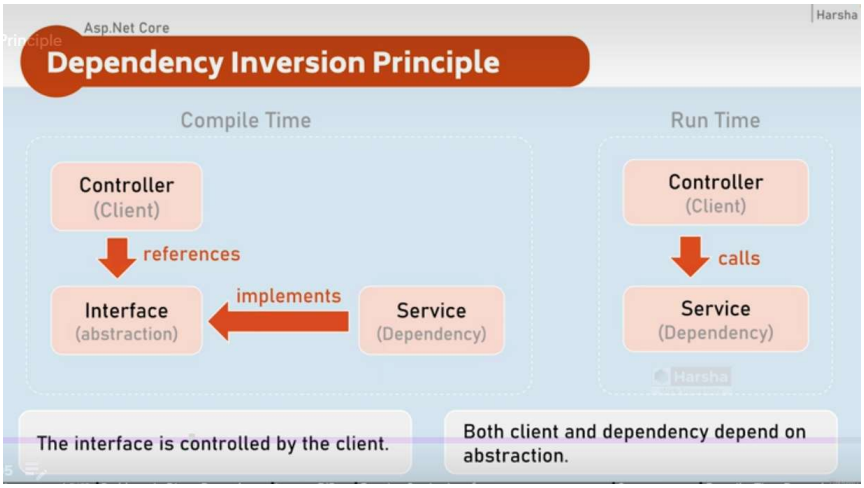
First developer that is developer A. (The developer of the Controller) .



Let's understand developer B (Service class Developer) perspective. He doesn't know from which controller this service should be called.



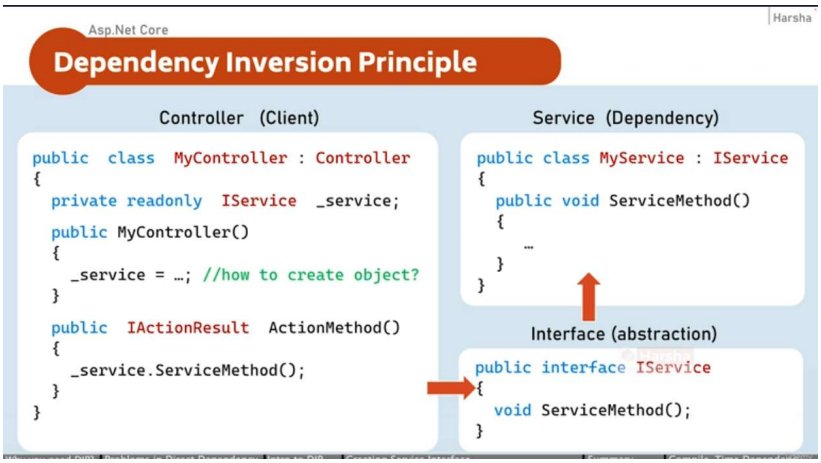
His responsibility is to implement interface 'ICitiesService', so both developer can work independently. Even they can work in parallel



This is the graphical picture. At runtime, Controller depends on Service. That is true. At runtime, the controller calls the methods of service.

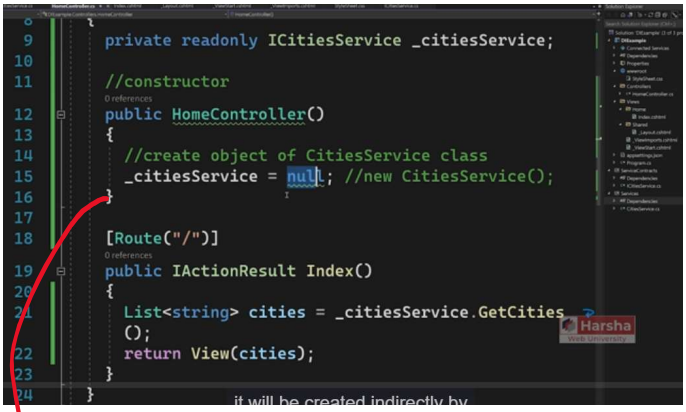
But at least at compilation time, they are loosely coupled. Both controller and service are not knowing or calling each other.

The Controller reference the interface and service implements the interface. So indirectly they are connected but not directly.

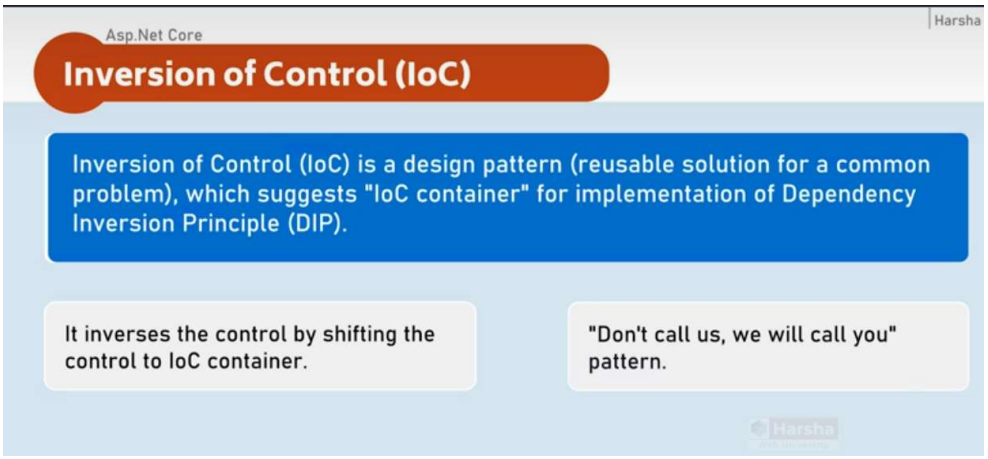


But how do we an object? With 'inversion of control' and 'dependency injection'. The 'Inversion of Control' takes the responsibility of creating object

And adding that object into this reference variable is the responsibility of dependency injection.



Now the question is who will create an object of 'CitiesService' here? It will be creating by implementing 'inversion of control'.

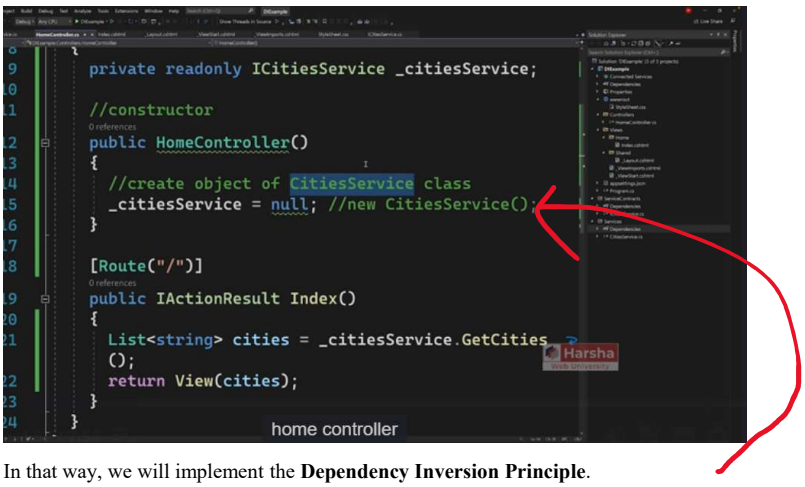


Here, the big question is: who will create the object of this CityService class?
It will be created indirectly by implementing **Inversion of Control (IoC)**.

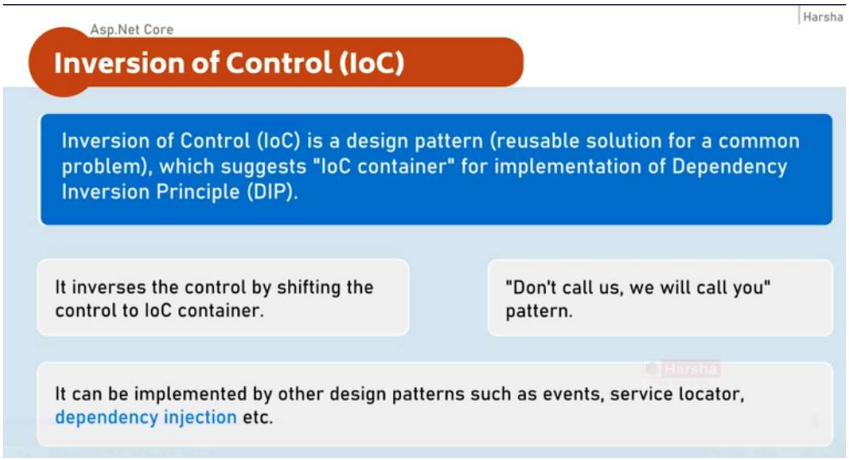
So, what is the exact meaning of Inversion of Control (IoC)?

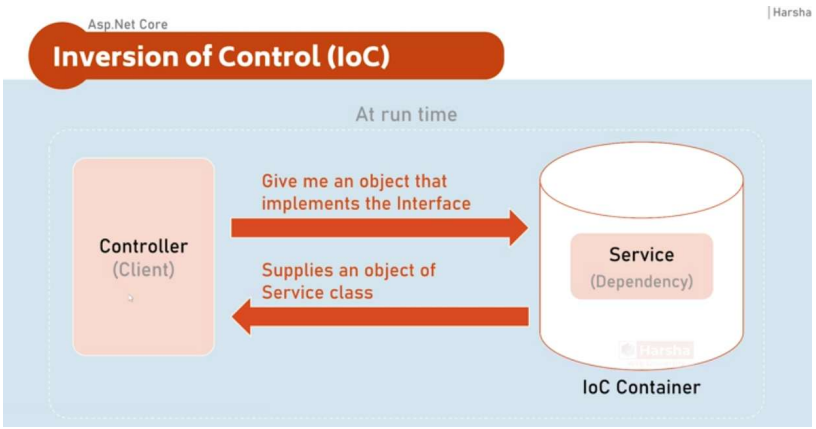
It is a design pattern, meaning a reusable solution for common problems in design.
IoC guides you on how to implement the **Dependency Inversion Principle**.
In the last lecture, we implemented the Dependency Inversion Principle. That means we created an interface, and it was created by the developer of the controller.

But to fulfill and implement the Dependency Inversion Principle, an **IoC container** is one of the ways.
It inverts the control by shifting the responsibility into the IoC container.
This means that instead of the developer creating an object of the CityService class directly, the IoC container takes on the responsibility of creating the object of the service class.

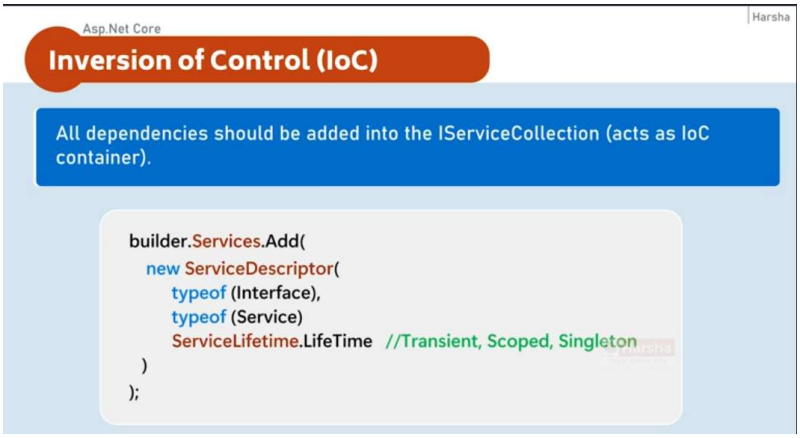


In that way, we will implement the **Dependency Inversion Principle**.
It follows the pattern of "Don't call us; we will call you."
For example, if you have applied for a job, the HR department might say, "Don't call us for a job inquiry; we will call you when we need you."
This is the intention of this pattern.
The controller should not call the CityService class directly.
Instead, the **IoC container** creates an object of the CityService class and supplies that object to the HomeController.





At runtime, the controller says, "Hey IoC container, please give me a service."
The service is already registered as part of the IoC container, so the IoC container creates an object of the service, returns it, and supplies it to the controller.
The controller then receives it and calls the necessary methods as needed.
This is how it works. But before than that we have to add the service class particularly to the ioc container and this is the statement for that.



Here is program.cs file, by default there is ioc container which is of IServiceCollection type.

```
var builder = WebApplication.CreateBuilde
builder.Services.AddControllersWithViews(

var app = builder.Build();

app.UseStaticFiles();
app.UseRouting();
app.MapControllers();
```

builder.Services is your IOC container. When you want to add your services into the IOC container, you have to add them into these services.

Here you can add essential namespaces and services.

```
using ServiceContracts;
using Services;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllersWithViews();
builder.Services.Add(new ServiceDescriptor() { });

var app = builder.Build();

app.UseStaticFiles();
app.UseRouting();
app.MapControllers();

app.Run();
```

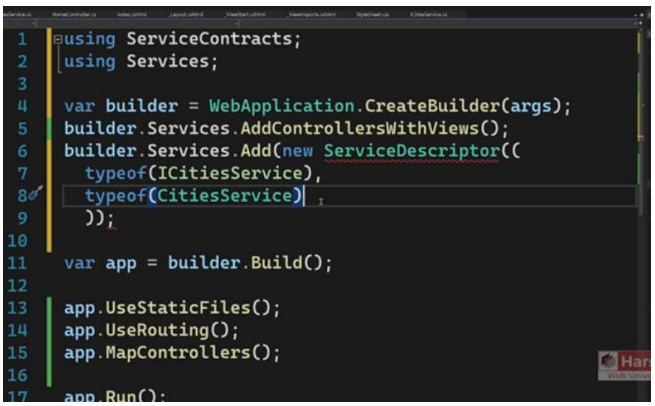
Hey IoC container, I would like to add a service.
While adding the service, its details are represented as an object of the **ServiceDescriptor** class, which is a predefined class. We create an instance of this class to hold the details of the service, such as:

- The type of interface (e.g., ICityService)
- The lifetime of the service
- The service class name (e.g., CityService)

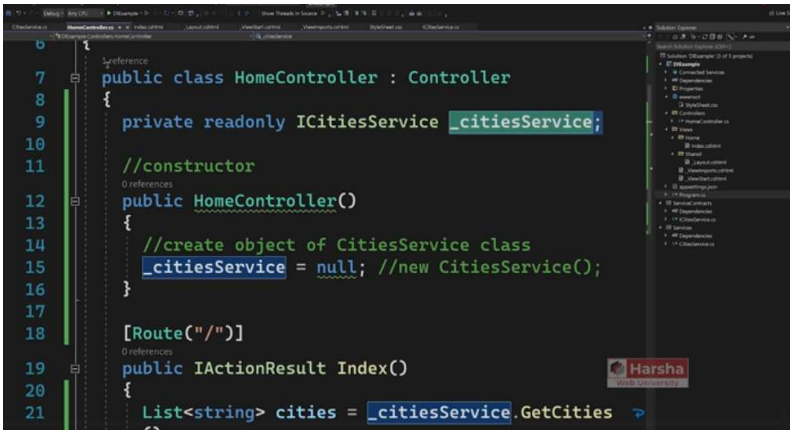
When creating the ServiceDescriptor, the constructor requires the following:
1. The first argument: the interface type (e.g., ICityService).
2. The second argument: the service class type (e.g., CityService).

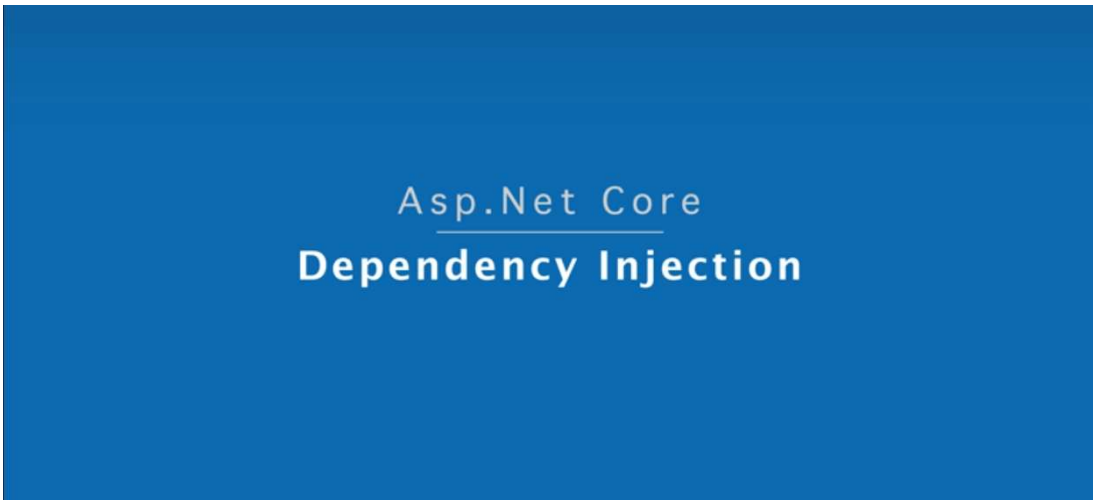
This setup tells the IoC container:

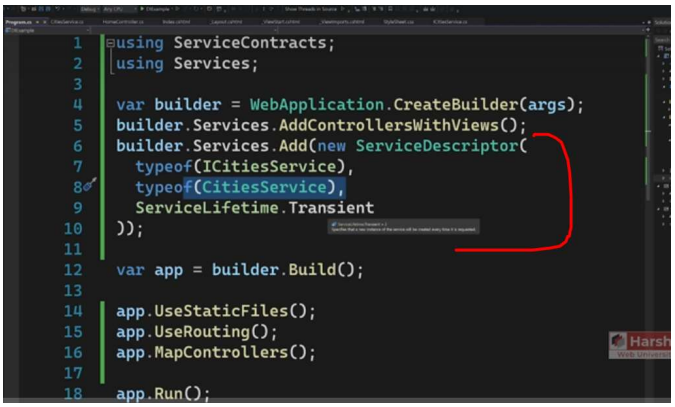
"Hey, whenever someone requests an object that implements ICityService, create an instance of CityService."
For example, if the HomeController or any other service requests an object implementing ICityService, the IoC container will automatically create an object of the CityService class and supply it to the HomeController.
This reference will then be stored in a field of the HomeController.
How does the HomeController receive it? That's what we will learn next.



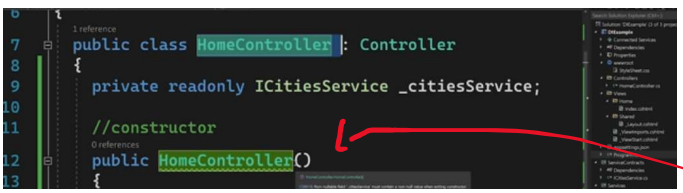
```
using ServiceContracts;
using Services;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllersWithViews();
builder.Services.Add(new ServiceDescriptor((
  typeof(ICitiesService),
  typeof(CitiesService)
  ));

var app = builder.Build();

app.UseStaticFiles();
app.UseRouting();
app.MapControllers();

app.Run();
```

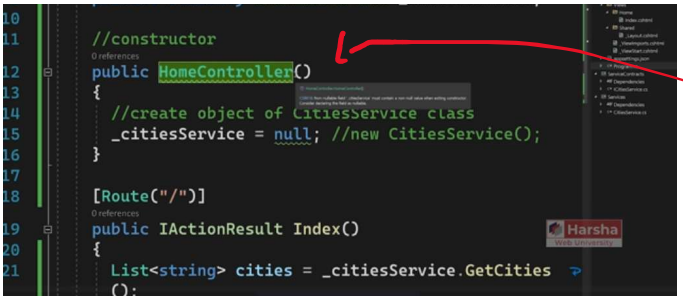






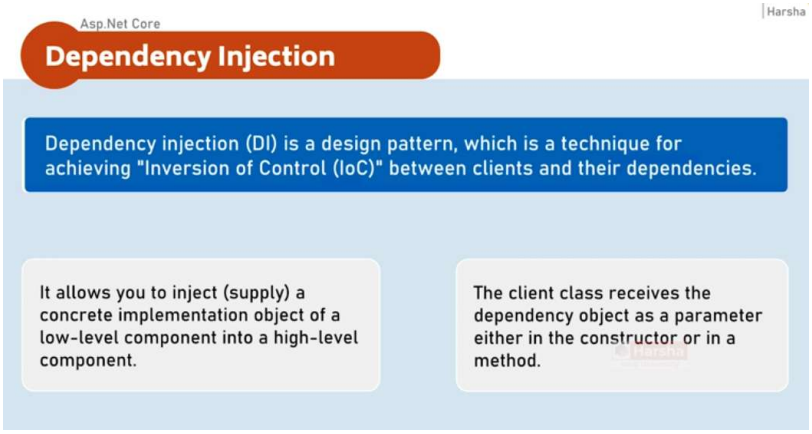
We have just understood that, the ioc container which is implemented here creates an object of City Service.



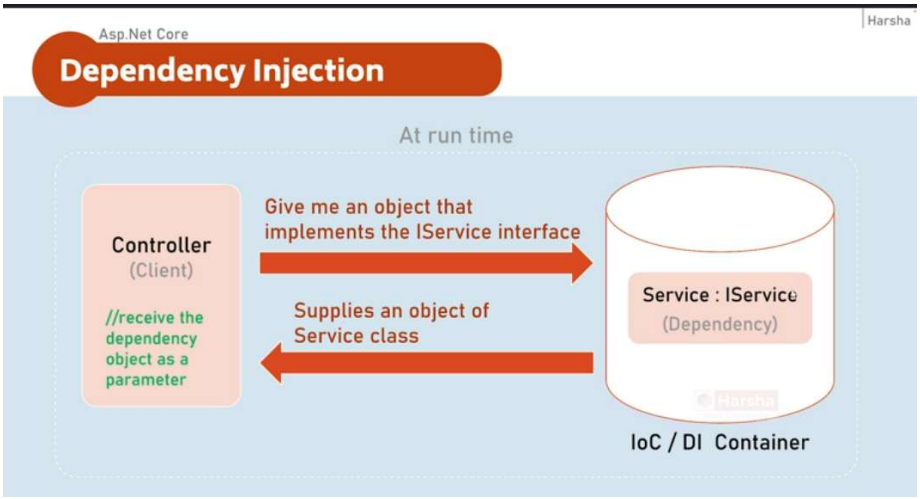


But is 'HomeController' ready to receive that object? As of now, no!

Because you have to implement **dependency injection** concept in this controller to receive the service instance.

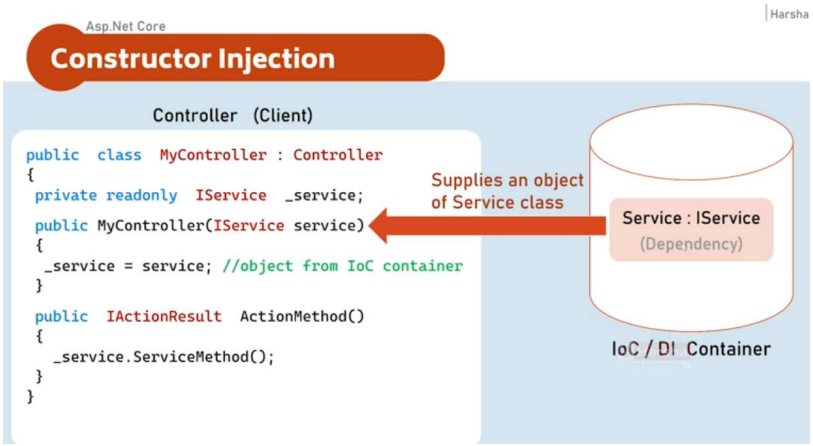


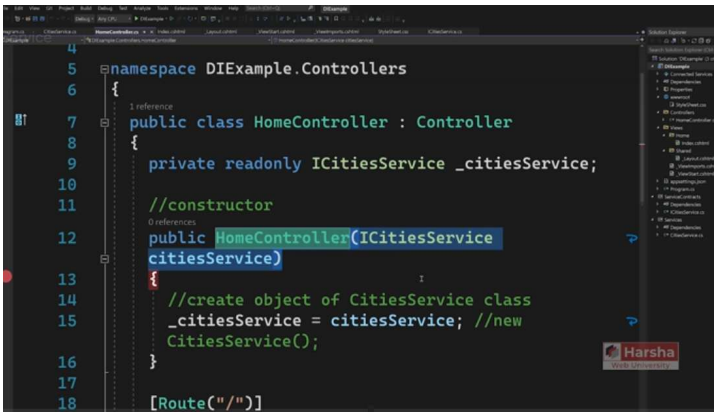
It is a design pattern that implements inversion of control.
Dependency injection is one of the ways to implement inversion of control.
It is a technique to receive the dependency object into the client.
In other words, it is a technique to receive the object of the CityService into the HomeController in our example.
The high-level component receives an object of the low-level component supplied by the IoC container.
According to the dependency injection concept, the service object can be received either through a constructor or a method parameter.



That means at runtime, the controller asks the IoC container, which is built into ASP.NET Core:
"Hey, ASP.NET Core IoC container, please supply me an object that implements a particular interface, for example, ICityService."
The IoC container, also called the DI container, already knows which service implements which interface.
Since the controller doesn't specify a specific class name but just asks for the interface, the container picks the corresponding service class based on the interface name.
For example, whenever the controller asks for ICityService, the IoC container picks the CityService class, creates an object of that service class

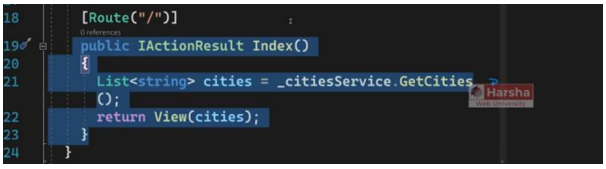
(also called the dependency class), and supplies it to the controller at runtime.
The client, or the controller, can receive the object either as a parameter in the constructor or as a parameter in the method.

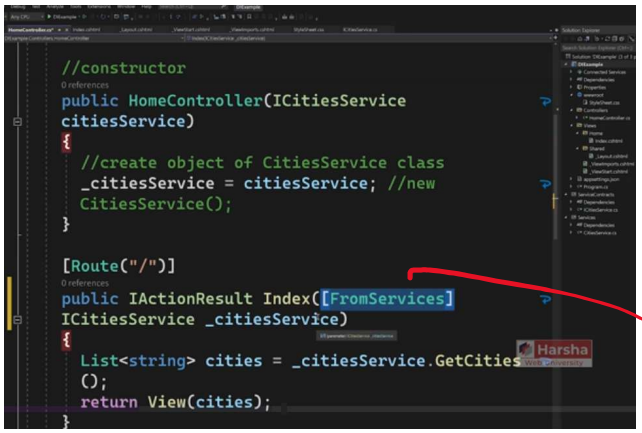




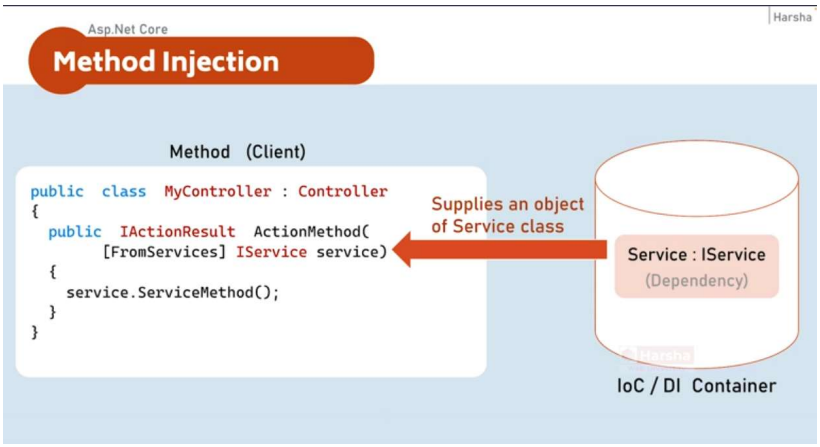
Currently, in this existing code, you are injecting the service into the constructor of this client class called controller. But suppose if you feel that the service instance is required only for a specific method, not for all the method of the same class in that case instead of injecting it into the constructor, you can inject the service into a specific method.

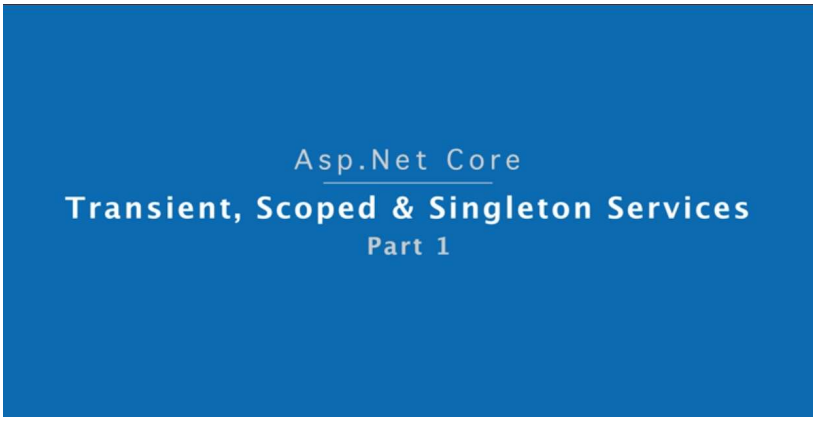
For example 'Index()' method. So that, that service instance is available only for the same action method but not for other methods in the same class.



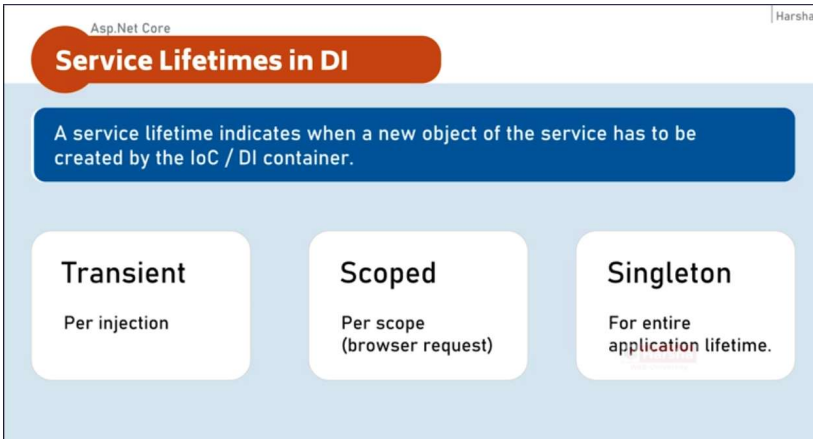


It instructs the IOC container to give an object of the city service class that implements ICitiesService.





```
var builder = WebApplication.CreateBuilde
builder.Services.AddControllersWithViews(
builder.Services.Add(new ServiceDescripto
  typeof(ICitiesService),
  typeof(CitiesService),
  ServiceLifetime.Transient
));

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



While adding your service into the IoC container in the Program.cs file, we create a new object of the ServiceDescriptor that represents a service. This is the interface of the service, and this is the name of the service—that's fine.
But what about the service lifetime? We have just given it as Transient. What about the others, like Scoped and Singleton? What's the difference? Let's try to understand.

A service lifetime in dependency injection represents when a service object should be created and when it should be destroyed by the IoC container.
In dependency injection, the IoC container (or DI container) is responsible for creating service objects and disposing of them.
But when exactly the service objects are created and when they are destroyed or disposed depends on the service lifetime: Transient, Scoped, or Singleton.

Transient

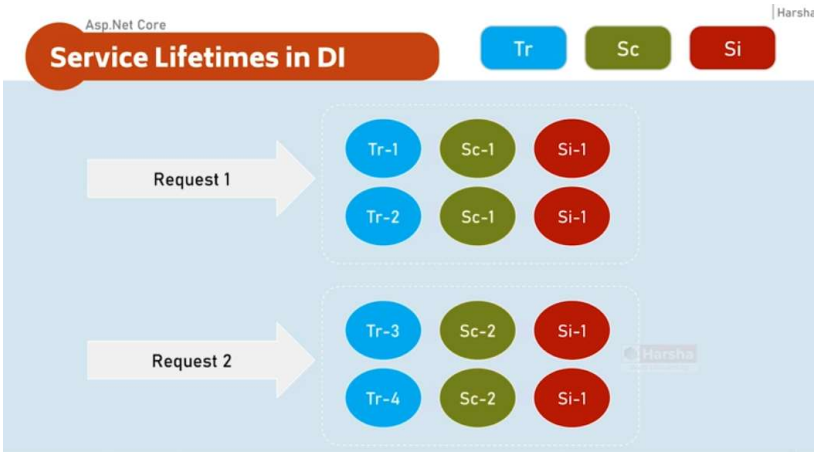
In the case of Transient, a new service object is created every time the service is injected.
For example, if we inject the same Transient service into three different controllers or services, a new object will be created each time the service is injected.
For instance, if the same CityService in our example is injected three times, three separate objects of the CityService class will be created internally.
That is why it is called **"per injection"**.

Scoped

For Scoped, a new service object is created for every scope.
The **scope lifetime** in general corresponds to a browser request.
This means that when the application encounters a new browser request, the IoC container creates a new scope.
At the end of the request (when the response is completed), the scope ends.
However, within the same request, it is possible to create multiple scopes.
In that case, a new service object is created for each scope.

Singleton

For Singleton, the service object is created only once—when it is injected into any class for the first time.
After that, the same service object is reused every time.
New objects of the Singleton service will not be created unless the application restarts.



Let's visualize the same graphically.

For example, we have created three services: one is of Transient type, one is Scoped type, and one is Singleton.
So, we have three services. In the same request, you may inject the services in any order, whenever you want, and any number of times—no rules.
When the first request is received by the application, assume we have injected these three services. For each service, a new object will be created. The first object here is represented as one:

- Transient service object 1
- Scoped service object 1
- Singleton service object 1

But maybe within the same request, in the same class or in another class, we have injected these three services again, a second time.
For each invocation, the Transient service should be instantiated, so you will get another object of the Transient service.
However, you will get the **same object** of the Scoped service because, by default, the entire request processing is treated as a single scope unless you create custom scopes.
Since you're injecting the same service a second time in the same scope, a new object of the Scoped service will not be created.
For Singleton, there will always be one and only one object, so undoubtedly, you will get the same object for the second invocation as well.
Now assume we have received a second request from the browser and injected all three services again.
As usual, you will get another object of the Transient service for each injection. Every time you call the service, you get a new object—that's about Transient services.
For the second request, a new scope is created. Within this scope, you will get another object of the Scoped service, called Scoped service object 2.
Always, undoubtedly, for the Singleton service, there will be a single object for all requests and all scopes.
If we inject the same services one more time in the same request, another object of the Transient service will be created, the same object of the Scoped service will be reused, and exactly one and only object of the Singleton service will be returned.
This is how the objects of the services are created.
It is also possible to inject a service into another service.

For example, at our HomeController, we have injected 'CitiesService'

```
namespace DIExample.Controllers
{
    0 references
    public class HomeController : Controller
    {
        /*private readonly ICitiesService _citiesService;

        //constructor
        public HomeController(ICitiesService citiesService)
        {
            //create object of CitiesService class
            _citiesService = citiesService; //new CitiesService();
        }
        */
        [Route("/")]
        0 references
        public IActionResult Index([FromServices]ICitiesService
        _citiesService)
        {
            List<string> cities =  citiesService.GetCities();
```

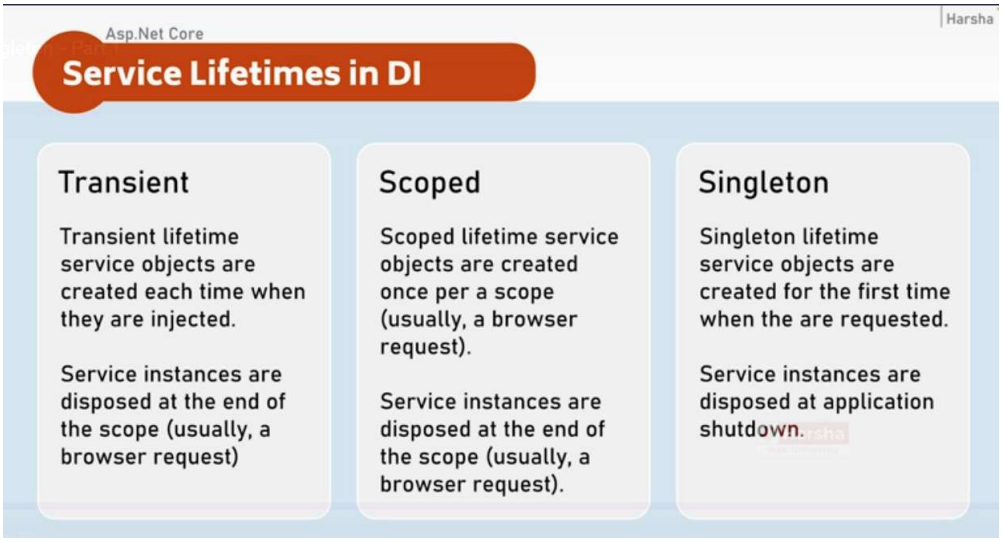
Maybe if needed within this city service you may inject another service and that service you may inject one or more other service
And internally the IOC container will automatically maintain all these service relations as service graph or dependency injection graph.
It's internally maintained by the IOC container. The developer has nothing to do with it.

```
{
    2 references
    public class CitiesService : ICitiesService
    {
        private List<string> _cities;

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

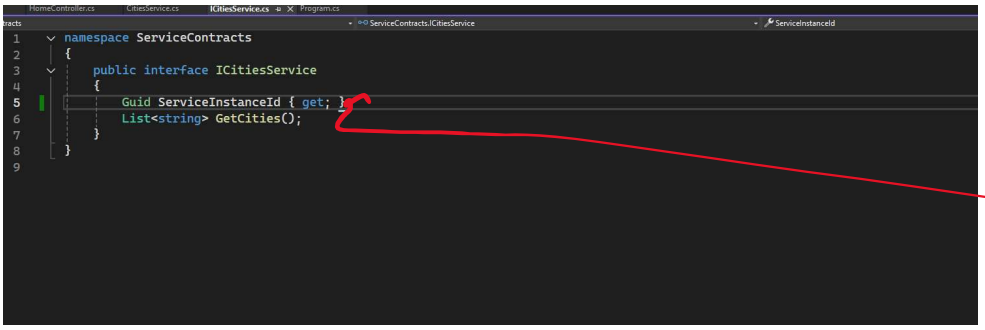
And it is also important to understand when the service objects get destroyed.

In the case of ***Transient services,*** the Transient service objects are created for every injection, but they will not be destroyed when the class execution is completed.
The Transient service instances will be destroyed at the **end of the scope**, not when their execution is completed.
For example, if you are injecting a Transient service called CityService into the HomeController, at the end of the Home Controller execution, the city service object will not be destroyed. It will be destroyed when the scope ends. That means the request processing has been Completed.

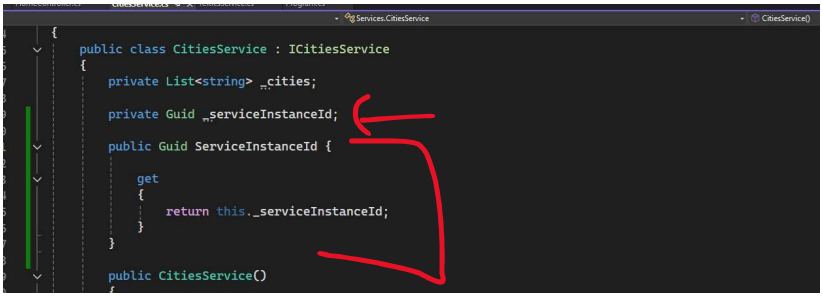
```
    0 references
    public class HomeController : Controller
    {
        /*private readonly ICitiesService _citiesService;

        //constructor
        public HomeController(ICitiesService citiesService)
        {
            //create object of CitiesService class
            _citiesService = citiesService; //new CitiesService();
        }
        */
        [Route("/")]
        0 references
        public IActionResult Index([FromServices]ICitiesService
        _citiesService)
        {
            List<string> cities = _citiesService.GetCities();
            return View(cities);
```

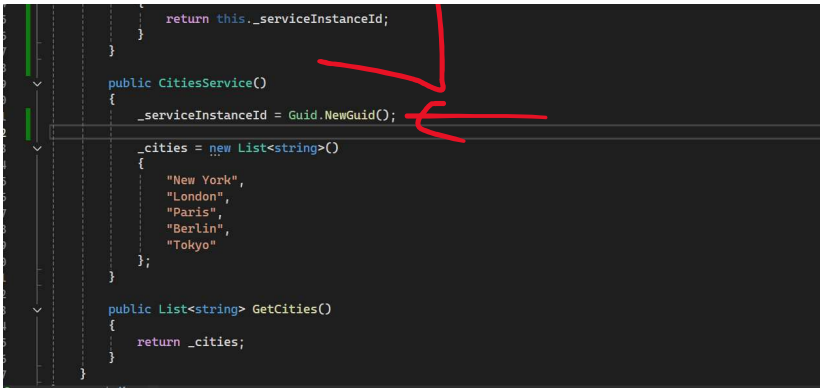


Let's understand these three service lifecycle practically.



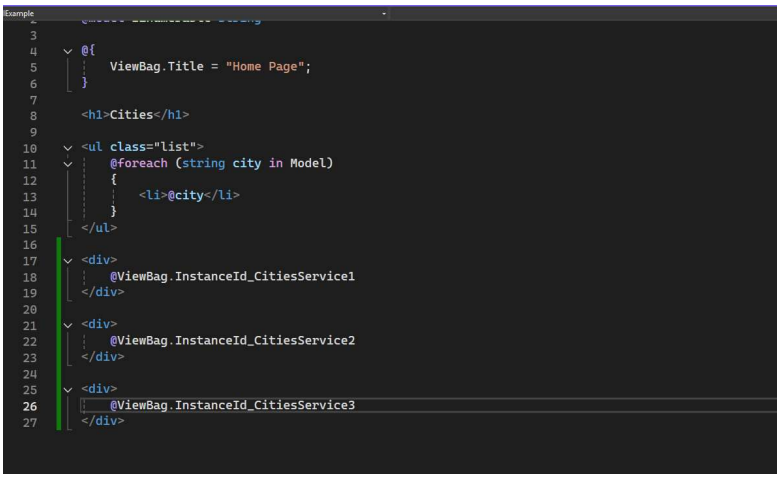
We have added Guid for understanding.



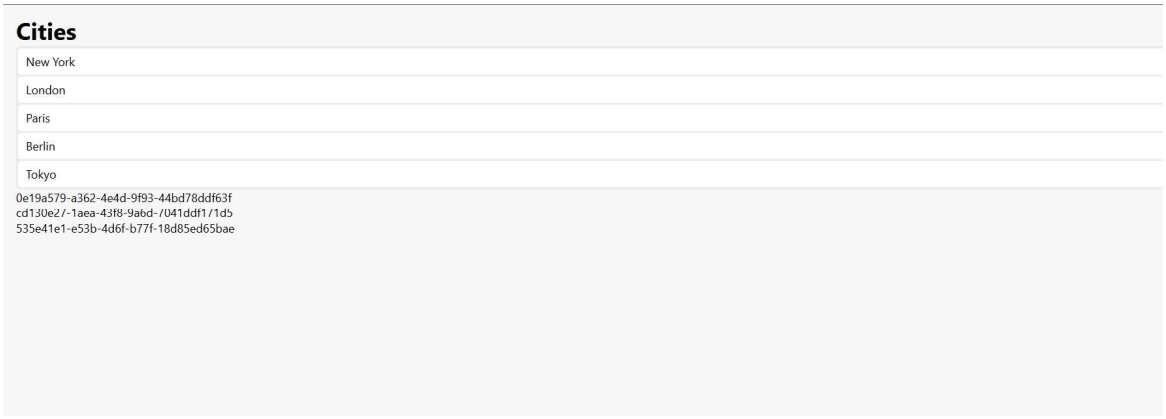


```
            return this._serviceInstanceId;
        }
    }

    public CitiesService()
    {
        _serviceInstanceId = Guid.NewGuid();

        _cities = new List<string>()
        {
            "New York",
            "London",
            "Paris",
            "Berlin",
            "Tokyo"
        };
    }

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

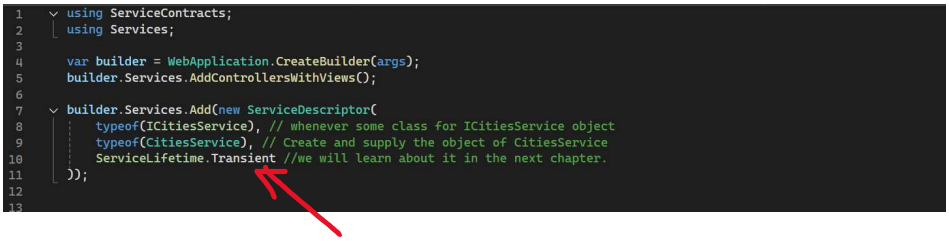
```
namespace DIExample.Controllers
{
    public class HomeController : Controller
    {
        public readonly ICitiesService _citiesService1;
        public readonly ICitiesService _citiesService2;
        public readonly ICitiesService _citiesService3;

        //constructor injection
        public HomeController(ICitiesService citiesService1, ICitiesService citiesService2, ICitiesService citiesService3)
        {

            _citiesService1 = citiesService1;
            _citiesService2 = citiesService2;
            _citiesService3 = citiesService3;
        }


        [Route("/")]
        public IActionResult Index()
        {
            List<string> cities = _citiesService1.GetCities();

            ViewBag.InstanceId_CitiesService1 = _citiesService1.ServiceInstanceId;
            ViewBag.InstanceId_CitiesService2 = _citiesService2.ServiceInstanceId;
            ViewBag.InstanceId_CitiesService3 = _citiesService3.ServiceInstanceId;

            return View(cities);
        }
    }
```



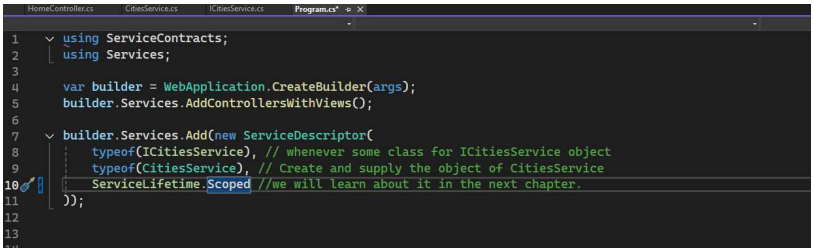
```
Example
3
4   @{
5       ViewBag.Title = "Home Page";
6   }
7
8   <h1>Cities</h1>
9
10  <ul class="list">
11      @foreach (string city in Model)
12      {
13          <li>@city</li>
14      }
15  </ul>
16
17  <div>
18      @ViewBag.InstanceId_CitiesService1
19  </div>
20
21  <div>
22      @ViewBag.InstanceId_CitiesService2
23  </div>
24
25  <div>
26      @ViewBag.InstanceId_CitiesService3
27  </div>
```



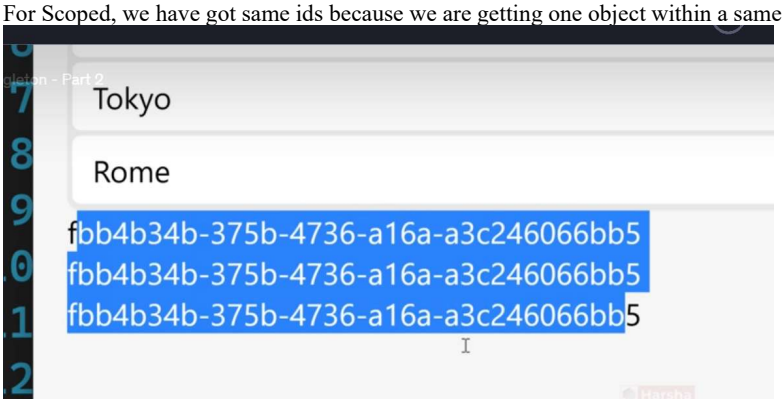
We have three different ids for 3 services because we have used 'transient' services.



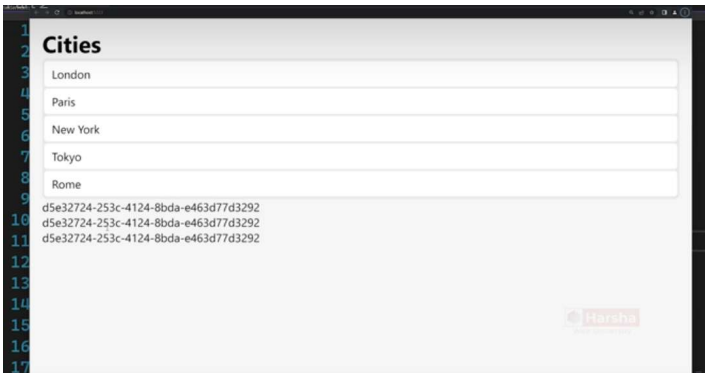
Now, let's explore 'Scoped' life cycle.



For Scoped, we have got same ids because we are getting one object within a same request scope.



Now, reload the tab and make another request and see we have got different object.

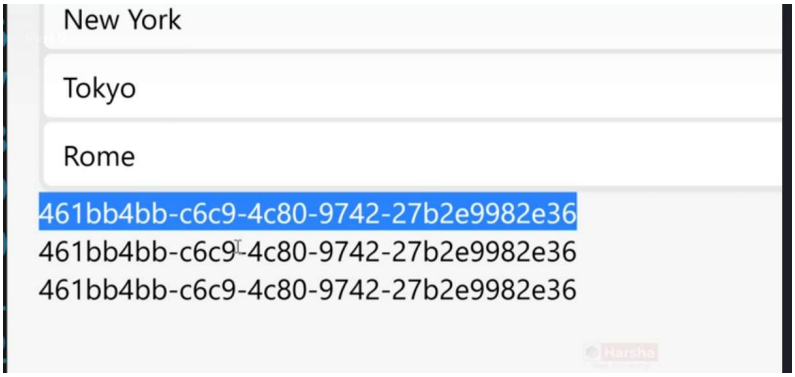


Singleton:

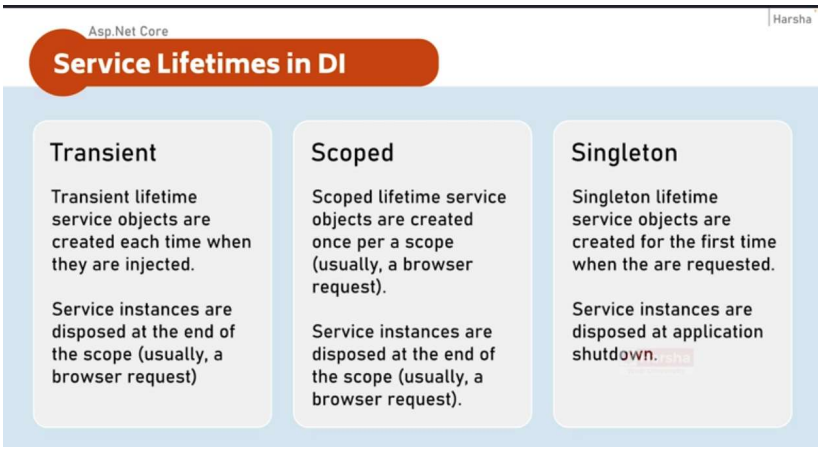
```
using ServiceContracts;
using Services;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllersWithViews();
builder.Services.Add(new ServiceDescriptor(
  typeof(ICitiesService),
  typeof(CitiesService),
  ServiceLifetime.Singleton
));

var app = builder.Build();

app.UseStaticFiles();
app.UseRouting();
```

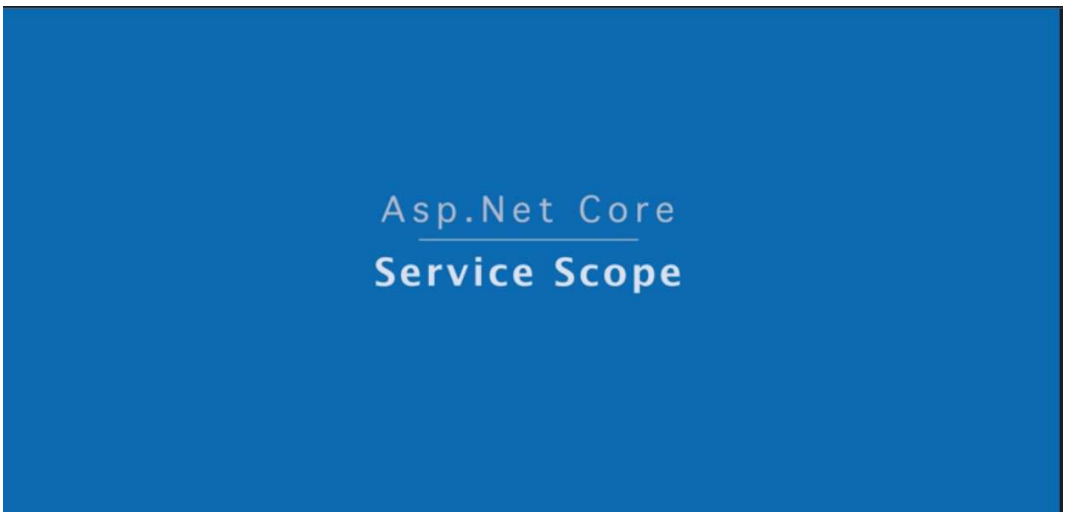
Now make it 'Singleton' service.

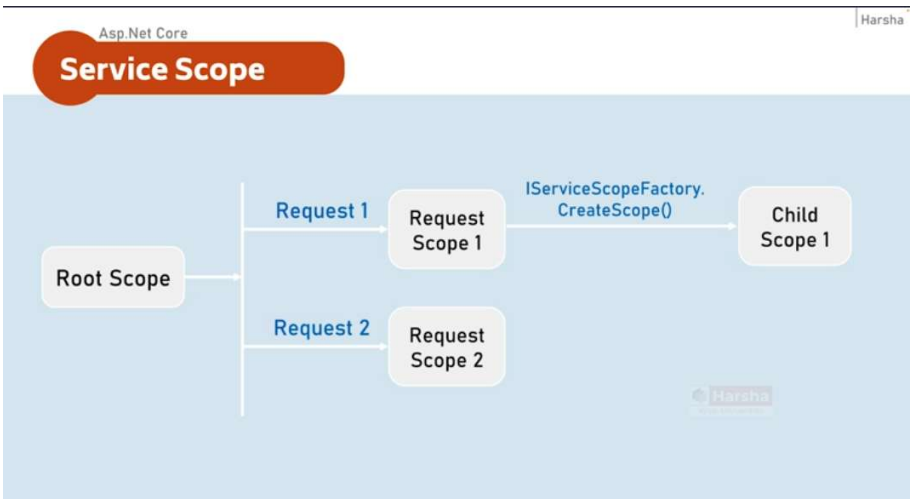
It looks like 'Scoped' service but the point here is the service object once created for the first time and the same will be persisted throughout the entire application lifetime. [For all the requests and all the scopes and all the injections]



Reload the tab, you will see the same result.







By default, whenever an application runs, a root scope gets created automatically in ASP.NET Core. For each request, a new scope gets created in ASP.NET Core, and you can call them request scopes.

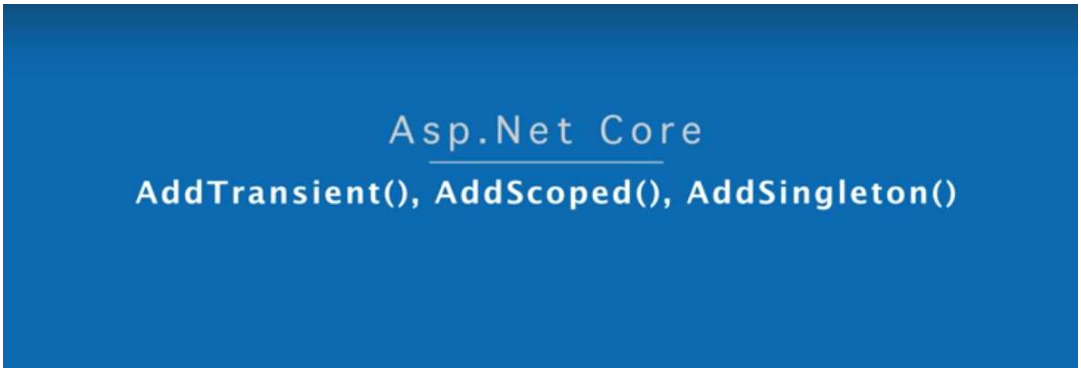
For example, if there are 10 requests, 10 request scopes will be created. That means for each request, a new request scope gets created.
It is also possible to create child scopes within the same request scope.
But what does a scope exactly represent?

A scope represents the lifetime of the scoped services and also the transient services.
Even though transient services get instantiated for each injection, they are disposed of at the end of the same scope.
It is also possible to create one or more child scopes within the same request scope by using the CreateScope method of the IServiceScopeFactory.

When Should You Create a Child Scope?

For example, you have created a disposable service that implements the IDisposable interface.

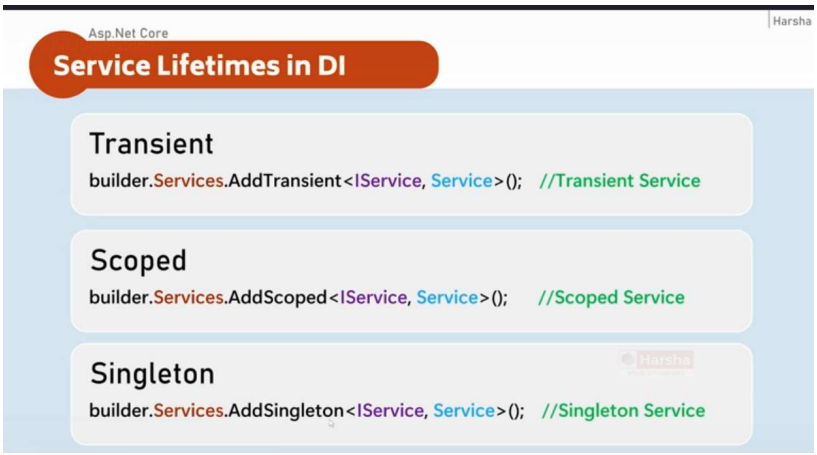
If your service creates a database connection in the constructor and closes that database connection in the Dispose method, the service must be short-lived.
The database connection should not remain open until the end of the request. It should be opened, and once your work is done, the database connection should be closed, meaning the service should be disposed of as soon as possible.
For that scenario, you will use the child scopes concept.



Currently we are adding the services into the service collection by using this add method by creating a new object of the service descriptor class.

```
1  using ServiceContracts;
2  using Services;
3
4  var builder = WebApplication.CreateBuilder(args);
5  builder.Services.AddControllersWithViews();
6  builder.Services.Add(new ServiceDescriptor(
7    typeof(ICitiesService),
8    typeof(CitiesService),
9    ServiceLifetime.Scoped
10 ));
11
12 var app = builder.Build();
13
```

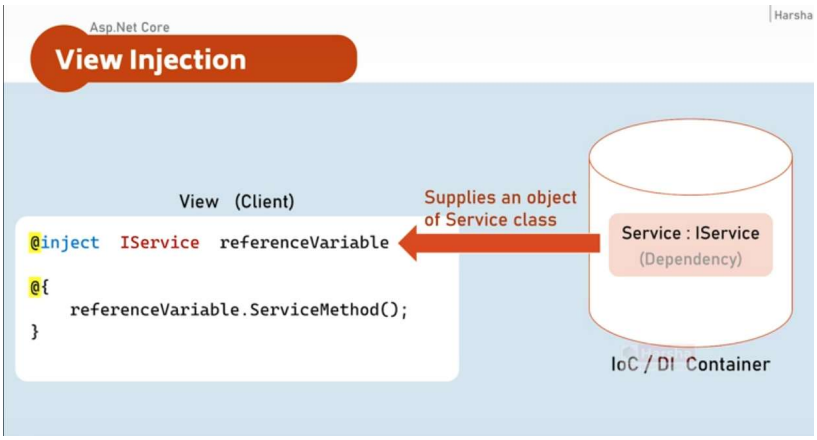
There are more shorthand method to add the service into the service collection.

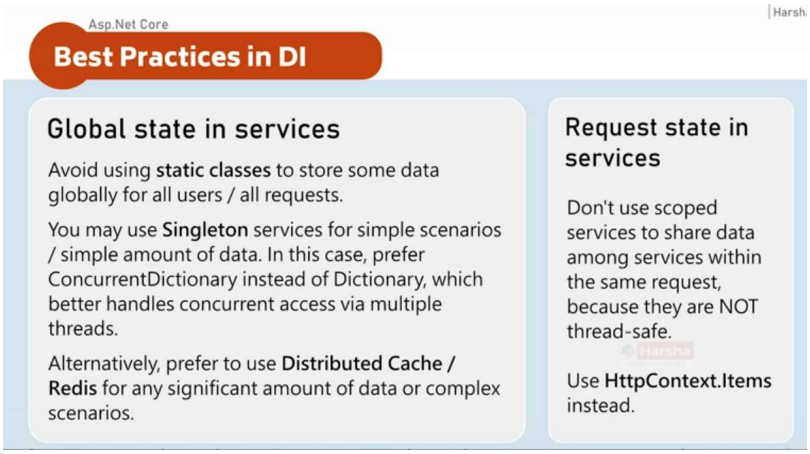



```
1  using ServiceContracts;
2  using Services;
3
4  var builder = WebApplication.CreateBuilder(args);
5  builder.Services.AddControllersWithViews();
6  builder.Services.Add(new ServiceDescriptor(
7    typeof(ICitiesService),
8    typeof(CitiesService),
9    ServiceLifetime.Transient
```

```
builder.Services.Add(new ServiceDescriptor(
  typeof(ICitiesService),
  typeof(CitiesService),
  ServiceLifetime.Transient
));
builder.Services.AddTransient<ICitiesService, CitiesService>();

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

In ASP.NET Core, the views are compiled as their classes.
That means when you create a view and compile it, it will automatically generate a class that inherits from a predefined class called RazorPage.
Just like you can inject services into controllers or other service classes, you can also inject services into the views.
This is possible by using the @inject directive within the view.
Let me show that.




Asp.Net Core
Best Practices in DI
Global state in services
Avoid using static classes to store some data globally for all users / all requests.
You may use Singleton services for simple scenarios / simple amount of data. In this case, prefer ConcurrentDictionary instead of Dictionary, which better handles concurrent access via multiple threads.
Alternatively, prefer to use Distributed Cache / Redis for any significant amount of data or complex scenarios.
Request state in services
Don't use scoped services to share data among services within the same request, because they are NOT thread-safe.
Use HttpContext.Items instead.