A screen shot of a computer

Description automatically generatedA screen shot of a computer program

Description automatically generated

A screen shot of a computer

Description automatically generated

The first one controller is dependent on interface and second one is on object. So, we are going on discussion on dependency injection.

The constructor of the BookController class is parameterized, meaning it contains list of parameters (interface).

**Note => When any class requests an object through its constructor using an interface, then services provide object related to that class. And manually assigns these objects to each private fields. So that action method interacts with the respective private fields.**

In the internally, at runtime through RegisterAllDependencies => Interface related Class are instantiated to the constructor of BookController when an instance of BookController is created. The private fields hold each class instantiate when handling requests in controller.

So, imagine you're in your workshop and you're about to start working on a car. Before you start, you gather all the tools you need, right? Similarly, when you create an instance of BookController, you gather all the dependencies it requires and pass them to its constructor.

When we talk about "instantiating" a controller, we're creating an object that represents that controller class in memory. It's like creating a physical object that embodies (Murta rupa dincha) all the properties and behaviors defined by the controller class.

Imagine you have a blueprint for a car. This blueprint contains all the instructions on how to build a car, but it's not an actual car itself. When you want to drive a car, you need to build it based on this blueprint. Instantiating a controller is like building a car based on the blueprint.

So, when we say we're "instantiating" a controller in .NET Core, we're creating an object based on the blueprint (the controller class) that contains all the instructions on how to handle incoming requests. This object will then be responsible for processing a specific request when it comes in, using the methods and actions defined in the controller class.

Imagine you have a blueprint for building a car. This blueprint is like a set of instructions detailing how to construct a car, but it's not an actual car itself. Now, when you want to drive a car, you can't just use the blueprint. You need to follow those instructions and build the car. That's where instantiation comes in.

So, when we talk about "instantiating" a controller in .NET Core, it's like taking that blueprint (which is the controller class) and actually building something real based on it. We're creating an object that knows how to handle requests coming into our application.

This object, just like a car built from a blueprint, is equipped with all the methods and actions defined in the controller class. It's ready to process specific requests that come its way, following the instructions laid out in the blueprint (or the controller class).

Imagine you're building a car, and you need various tools to assemble it, like a wrench, a screwdriver, etc. Each tool/task is essential for the BookController to assemble the car/software properly. So, managing users, handling files, encrypting data, etc., are all tasks that the BookController must be able to perform effectively.

Now, when you're ready to put everything together and start using your tools to build the car, you need to gather all your tools first, right? That's what the constructor does. It's like a checklist where you list down all the tools you need. In this case, the constructor takes all the necessary tools (dependencies) as parameters.

The private fields in the code are like storage spaces where you keep your tools handy. When you get your tools (dependencies) from the constructor, you store them in these private fields so you can easily access them whenever you need to use them during the construction process (when handling requests in your controller).

So, the private fields hold onto the tools (dependencies), and the constructor gathers these tools and puts them in the right places so you can use them effectively when you're working on your car (handling requests in your controller). This setup helps keep your code organized and makes it easier to manage all the different tasks your controller needs to handle.

Just like gathering tools before building a car ensures a smooth construction process, gathering dependencies before instantiating a controller ensures that it's well-equipped to handle requests effectively. It's all about having the right tools (dependencies) at hand to carry out the tasks required by the controller. And just like storing tools in designated spaces keeps them organized and easily accessible during construction, storing dependencies in private fields ensures they're readily available for use within the controller's methods and actions. This approach indeed helps maintain code organization and facilitates efficient handling of various tasks within the controller.

A screenshot of a computer program

Description automatically generated

The constructor of the PublicController class receives several dependencies through dependency injection. Means, the constructor receives several classes instantiate at the run time from outside of class internally.

The action methods (GetRelationshipList) within the controller handle incoming HTTP requests and delegate the actual processing to the appropriate business logic and services.

**When developing applications** using .NET Core, one often encounters the need to manage the lifecycle of objects and services. This is where dependency injection comes into play. Dependency injection helps in creating loosely coupled components by providing dependencies to an object from an external source, rather than creating them within the object itself.

A close-up of text

Description automatically generated

**Singleton**

The Singleton services ensures that only one instance of a service is created and shared throughout the application’s lifetime. This means that whenever a request for the service is made, the same instance is returned. Singleton instance is created only when it's first requested. It has memory efficient as they are created once reused everywhere.

Imagine you have a favorite pen that you always use. No matter how many times you need a pen, you always grab that same one. In programming, a Singleton is like that favorite pen. It ensures that there's only one instance (or copy) of a particular service throughout your entire application. So, whenever your code needs that service, it always gets the same instance. It's like having one go-to pen that you use every time.

A close-up of a diagram

Description automatically generated

**Scoped**

The Scoped services ensures that with every HTTP request, get a new instance. The same instance is provided for the entire scope of that request.

Now, think about going to a coffee shop. When you order a drink, they give you a cup, and that cup is yours for the time you're in the shop. No matter how many sips you take, you're always using the same cup until you leave. Scoped services in programming work similarly. With every new HTTP request (like ordering a new drink), you get a new instance of a service. And just like the cup you get at the coffee shop, that instance stays with you throughout the entire request.

A diagram of several different types of instances

Description automatically generated with medium confidence

**Transient**

The Transient services ensures that with every HTTP request, a new instance is created, perfect for short lived and stateless classes, like services implementing the business logic.

A diagram of several steps

Description automatically generated

Lastly, let's talk about transient services. Imagine you're at a party and someone offers you a disposable cup for your drink. You use it for a short while, then throw it away when you're done. Transient services are like those disposable cups. Every time you need a service, a new instance is created for you. And once you're done using it (like after a single HTTP request), it's thrown away. These are perfect for things that don't need to stick around for long, like temporary calculations or short-lived tasks.

**In short**, assuming a simple layered ASP.NET Core application that creates a scope for each request, and a business operation that creates a shopping cart order. In terms of dependencies, it would be something like this:

A screenshot of a computer program

Description automatically generated

As you can see, the logger is a singleton, because it's thread safe and will be shared across all services saving some memory, the DbContext is scoped ensure database state is shared inside that HTTP request, and the others can be transient because they are stateless and short lived.

**What is IOC/DI Container?**

An Inversion of Controller container or Dependency Injection container is a framework that helps with Dependency Injection. It

1. creates
2. and injects dependencies for us automatically.

**What is Dependency Injection?**

Dependency Injection is a software design pattern that allow us to register services (dependencies) and their implementations at application startup. These services are often defined as interfaces, which are then implemented by concrete classes. When the application needs an instance of a service, it requests it from the DI container, which then resolves the appropriate implementation based on the registered mappings.

**Services (Dependencies):** These are abstractions or interfaces that define what function or methods a component need. For example, there is a service called **IUserService** that defines methods for user authentication, registration, and so on.

**Implementations:** These are the concrete classes that provide the actual implementation of the services/interfaces. Going back to the **IUserService** example, there exist a class like **UserService** that implements the methods defined in the **IUserService** interface. This class contains the actual code that performs the authentication, registration, etc.

Let's say your application has a service interface **IUserService** and two concrete implementations:

**UserService** and **AdminUserService**. During application startup, you register these services with the DI container:

**services.AddScoped<IUserService, UserService>();**

**services.AddScoped<IUserService, AdminUserService>();**

Now, when some part of your application requests an instance of **IUserService**, the DI container looks at the registered mappings. Depending on the scope and other factors, it chooses the appropriate implementation to instantiate and provide back to the application. If it's a regular user accessing the service, it might provide an instance of **UserService**, but if it's an administrative task, it might provide an instance of **AdminUserService**.

**Example:** Imagine you're the manager of a restaurant. Your restaurant has several departments, such as the kitchen, the bar, and the front-of-house staff. Each department has specific tasks and responsibilities.

Now, let's relate this scenario to dependency injection:

**Services (Dependencies):**

**Kitchen Service:** Responsible for preparing food.

**Bar Service:** Responsible for preparing drinks.

**Front-of-House Service:** Responsible for greeting customers, taking orders, and serving food.

**Implementations:**

**Head Chef:** Implements the Kitchen Service. They oversee food preparation and ensure quality.

**Bartender:** Implements the Bar Service. They mix drinks and handle the bar area.

**Waitstaff:** Implements the Front-of-House Service. They interact with customers, take orders, and serve food.

**Dependency Injection:**

At the beginning of the day (**application startup**), you assign roles to your staff members. You specify who will be the Head Chef, Bartender, and Waitstaff.

During the day, when a customer arrives and places an order, the Waitstaff doesn't directly cook the food or prepare drinks. Instead, they request the required service from the assigned staff members (Head Chef for food, Bartender for drinks).

The Waitstaff doesn't need to know how the Head Chef or Bartender prepares the food or drinks. They simply interact with them through their defined roles (services), which are implemented by specific staff members (implementations).

If a staff member is unavailable (e.g., the Head Chef goes on break), you can easily swap them out with a replacement without affecting the overall operation of the restaurant. This is similar to how dependency injection allows you to swap out implementations of services in software without changing the consuming components.

In this analogy, the roles (services) define what needs to be done (prepare food, make drinks), while the staff members (implementations) actually carry out those tasks. Dependency injection helps keep the different components of the restaurant (or software application) loosely coupled, allowing for flexibility, maintainability, and scalability.

**What is Inversion of Control?**

Inversion of control is a broad term but for software developer it is a pattern used for decoupling components and layers in the system.

For example, your application has a text editor component, and you want to provide spell checking. Standard code would look like this something

A screenshot of a computer program

Description automatically generated ]

What we've done here creates a dependency between the TextEditor and the SpellChecker. In an IoC scenario we would instead do something like this:

A screenshot of a computer program

Description automatically generated

In traditional programming, when a class needs another class or object to perform certain tasks, it usually creates that object itself. For example, if a TextEditor class needs a SpellChecker to check the spelling of text, it would typically create a SpellChecker object within its own code.

The TxtEditor class receive an instance of IocSpellChecker from whoever creates an instance of TextEditor. This means that the responsibility of creating the IocSpellChecker instance is delegated to another part of the code, outside of the TextEditor class.

This delegation of responsibility is what we call "inversion of control" (IoC). Instead of the TextEditor class controlling the creation of its dependencies (in this case, the IocSpellChecker), the control is inverted or shifted to the code that creates the TextEditor instance. The TextEditor class simply relies on the fact that it will be provided with an IocSpellChecker instance when it's constructed, but it doesn't care about how that instance is created.

**Why we use Dependency Injection through Constructor Injection in dot net core?**

Constructor injection in .NET Core is all about keeping our code neat and easy to work with.

This method makes it simple to test our code because we can easily swap out real parts with fake ones, kind of like using toy tools instead of real ones to see if everything fits together correctly.

This setup also helps keep our code flexible and we can make changes without causing a big mess.

Overall, it enhances code readability, flexibility.

**How can we achieve dependency injection through constructor injection?**

we can achieve dependency injection through constructor injection which involves defining dependencies as interfaces, accepting them as parameters in the constructor of class, configuring the .NET core DI container to map interfaces to concrete implementations during application startup and utilizing the injected dependencies within the class.