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.