3. Dependency Injection

3.1 Built-in IoC Container

- The built-in container is represented by IServiceProvider implementation that supports constructor injection by default. The types (classes) managed by built-in IoC container are called services.
- There are basically two types of services in ASP.NET Core:
  - Framework Services :
    Services which are a part of ASP.NET Core framework such as IApplicationBuilder, IHostingEnvironment, ILoggerFactory etc.
  - Application Services :
    The services (custom types or classes) which we as a programmer creates for our application.
- In order to let the IoC container automatically inject our application services, we first need to register them with IoC container.

3.2 Registering Application Service

- ASP.NET Core allows us to register our application services with IoC container, in the ConfigureServices method of the Startup class. The ConfigureServices method includes a parameter of IServiceCollection type which is used to register application services.
- Example :

```
public class Startup
{
        public void ConfigureServices(IServiceCollection services)
        {
                services.Add(new ServiceDescriptor(typeof(ILog), new
                MyConsoleLogger()));
        }
}
```

- As we can see, Add() method of IServiceCollection instance is used to register a service with an IoC container. The ServiceDescriptor is used to specify a service type and its instance. We have specified ILog as service type and MyConsoleLogger as its instance. This will register ILog service as a singleton by default. Now, an IoC container will create a singleton object of MyConsoleLogger class and inject it in the constructor of classes wherever we include ILog as a constructor or method parameter throughout the application.

- Thus, we can register our custom application services with an IoC container in ASP.NET Core application.

3.3 Understanding Service Lifetime

- Built-in IoC container manages the lifetime of a registered service type. It automatically disposes a service instance based on the specified lifetime.
- The built-in IoC container supports three kinds of lifetimes:
    - Singleton :
      IoC container will create and share a single instance of a service throughout the application's lifetime.
    - Transient :
      The IoC container will create a new instance of the specified service type every time you ask for it.
    - Scoped :
      IoC container will create an instance of the specified service type once per request and will be shared in a single request.

3.4 Extension Methods for Registration

- ASP.NET Core framework includes extension methods for each types of lifetime; AddSingleton(), AddTransient() and AddScoped() methods for singleton, transient and scoped lifetime respectively.
- The following example shows the ways of registering types (service) using extension methods.
- Example :

```
public void ConfigureServices(IServiceCollection services)
{
        services.AddSingleton<ILog, MyConsoleLogger>();
        services.AddSingleton(typeof(ILog), typeof(MyConsoleLogger));

        services.AddTransient<ILog, MyConsoleLogger>();
        services.AddTransient(typeof(ILog), typeof(MyConsoleLogger));

        services.AddScoped<ILog, MyConsoleLogger>();
        services.AddScoped(typeof(ILog), typeof(MyConsoleLogger));
}
```

3.5 Constructor Injection

- Once we register a service, the IoC container automatically performs constructor injection if a service type is included as a parameter in a constructor.
- Example :

```
public class HomeController : Controller
{
        ILog _log;

        public HomeController(ILog log)
        {
                _log = log;
        }

        public IActionResult Index()
        {
                _log.info("Executing /home/index");
                return View();
        }
}
```

- In the above example, an IoC container will automatically pass an instance of MyConsoleLogger to the constructor of HomeController. We don't need to do anything else. An IoC container will create and dispose an instance of ILog based on the registered lifetime.
- Action Method Injection
- Sometimes we may only need dependency service type in a single action method. For this, use [FromServices] attribute with the service type parameter in the method.
- Example :

```
public class HomeController : Controller
{
        public HomeController()
        {
        }

        public IActionResult Index([FromServices] ILog log)
        {
                log.info("Index method executing");
                return View();
        }
}
```