Dependency Injection

401 ASP.NET CORE

What is it?

- What is it?
  - Dependency Injection makes it easy to create loosely coupled components, which typically means that components consume functionality defined by interfaces without having any first-hand knowledge of which implementation classes are being used.
- Why is it useful?
  - Dependency injection makes it easier to change the behavior of an application by changing the components that implements the interfaces that define application features. It also results in components that are easier to isolate for unit testing.
- How is it used?
  - The Startup class is used to specify which implementation classes are used to deliver the functionality specified by the interfaces used by the application. When new objects—such as controllers—are created to handle requests, they are automatically provided with instances of the implementation classes they require.

Alternatives?

- Are there any pitfalls or limitations?
  - The main limitation is that classes declare their use of services as constructor arguments, which can result in constructors whose only role is to receive dependencies and assign them to instance fields.
- Are there any alternatives?
  - You don't have to use dependency injection in your own code, but it is helpful to know how it works because it is used by MVC to provide features to developers.
- Has it changed since MVC 5?
  - Previous versions of ASP.NET MVC were designed to enable dependency injection, but you had to select and install a third-party tool to make it work. In ASP.NET Core MVC, a complete DI implementation is included as part of ASP.NET and is used extensively by MVC internally, although it can be replaced with a third-party package.

ADVATAGES

- Helps with adhering to the Dependency Injection Principle (DIP)
- Allows objects to be easily swapped with replacements
- Facilitates the use of the strategy design pattern (SDP)
- Improves testability
- Enables loose coupling of software components

DISADVANTEGES

- DI introduces a learning curve
- DI requires significant overhaul of some existing pr ojects
- Some timelines may not allow for D.I

SOLID Principles

- **Single responsibility principle**: Each class should only be responsible for one primary function. This encourages better class naming and discourages developers from making a class more than it needs to.
- **Open-Closed Principle:** Objects should be open for extension, while remaining closed for modifications. This encourages creation of subclasses for added functionality without breaking existing code
- **Liskov Substitution Principle**: objects should be replicable with appropriate objects. For example, other objects that share the same parent class or common interface. This enables loose coupling of related objects.
- **Interface Segregation Principle** Instead of overusing one generic interface, it is better to have more interfaces, well suited for specific purposes. His encourages you to keep each interface lightweight.
- **Dependency Inversion Principle:** Objects should be decoupled or loosely coupled. This forces classes to depend on the abstract definition of another object instead of a concrete implementation.

What does it look like?

```
using Microsoft.AspNetCore.Mvc;
using DependencyInjection.Models;
using DependencyInjection.Infrastructure;

namespace DependencyInjection.Controllers {

    public class HomeController : Controller {
        private IRepository repository;

        public HomeController(IRepository repo) {
            repository = repo;
        }

        public ViewResult Index() => View(repository.Products);
    }
}
```

How does it work?

1. MVC receives an incoming request to an action method on the Home controller.
2. MVC asks the ASP.NET service provider component for a new instance of the HomeController class.
3. The service provider inspects the HomeController constructor and discovers that it has a dependency on the IRepository interface.
4. The service provider consults its mappings to find the implementation class it has been told to use for dependencies on the IRepository interface.
5. The service provider creates a new instance of the implementation class.
6. The service provider creates a new HomeController object, using the implementation object as a constructor argument.
7. The service provider returns the newly created HomeController object to MVC, which uses it to handle the incoming HTTP request.

Resolving Dependencies

- We need to let the provider know how to resolve the dependencies.

```
public class Startup {

        public void ConfigureServices(IServiceCollection services)
        {
        services.AddTransient<IRepository, MemoryRepository>();
        services.AddMvc();
        }
    ….
}
```

LIFECYCLE MANAGEMENT

In order to manage your dependency's lifecycle in ASP.NET, here is a list of lifetimes you can choose from:

- Transient: A new instance will be created each time the object is needed
- Scoped: A new instance will be created for each web request
- Singleton: A new instance will be created only once at application startup
- Instance (special case of Singleton): Use AddSingleton() and create an instance yourself

In each case, the object will be disposed only after it is no longer needed. To use one or more lifetime settings, you would typically add the your code to the ConfigureServices() method of your Startup.cs file.

Constructor Injection

```
    private IRepository repository;

    public HomeController(IRepository repo) {
        repository = repo;
    }

    public ViewResult Index() => View(repository.Products);
}
```

Action Injection

```
public ViewResult Index([FromServices]ProductTotalizer totalizer) {
        ViewBag.HomeController = repository.ToString();
        ViewBag.Totalizer = totalizer.Repository.ToString();
        return View(repository.Products);
    }
```