C# - Dependency injection theory and .NET Core DI

Orestis Meikopoulos
Azure Cloud & AI Consultant

Agenda

- Dependency Injection - Introduction and Theory
- Dependency Injection in .NET Core
- Hands-on Demo
  - Refactoring code by moving business logic to Services and practically apply DI

Audience Question 1

What is your personal design goals when implementing a software system?

Design Goals

- **Maintainability**
  - The quality of a software system that determines how easily and how efficiently you can update it
- **Testability**
  - The ability to effectively test individual parts of the system
- **Parallel Development**
  - The ability to have separate development teams work on the same system in parallel

What is a dependency?

- A **dependency** is any object that another object requires in order to do its work
- The **MyDependency** class is a dependency of the **IndexModel** class
- The class creates and directly depends on the MyDependency instance

```csharp
public class MyDependency
{
    public MyDependency()
    {
    }

    public void WriteMessage(string message)
    {
        Console.WriteLine(
            $"MyDependency.WriteMessage called. Message: {message}");
    }
}
```

```csharp
public class IndexModel : PageModel
{
    private readonly MyDependency _dependency = new MyDependency();

    public void OnGet()
    {
        _dependency.WriteMessage(
            "IndexModel.OnGet created this message.");
    }
}
```

What is a dependency?

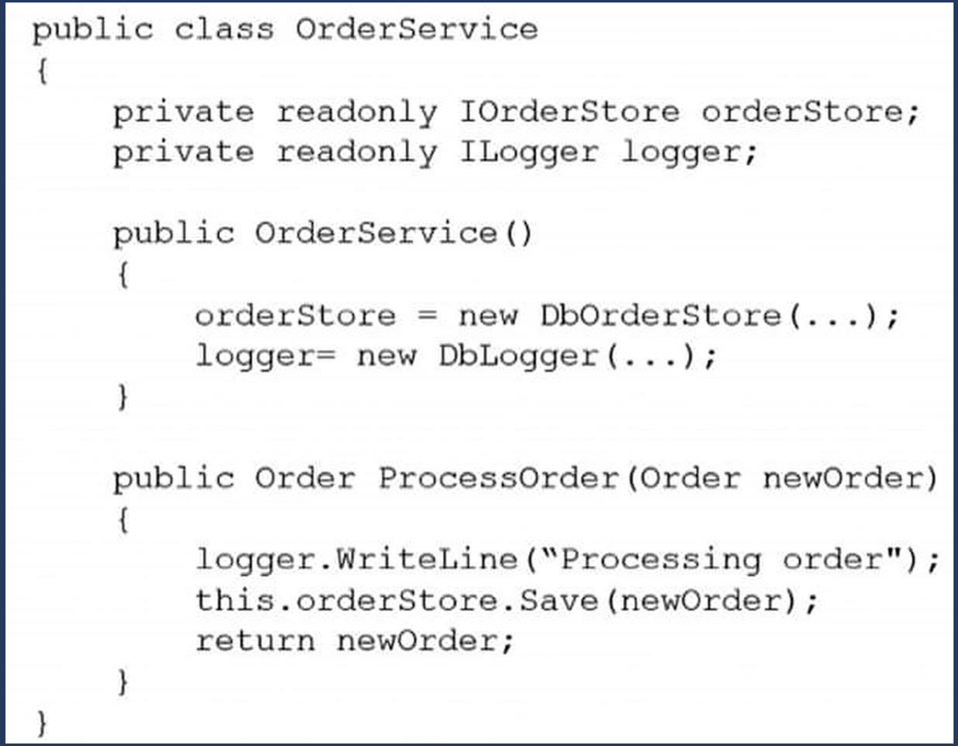
- Code dependencies, such as the previous example, are problematic and should be avoided for the following reasons
  - To replace MyDependency with a different implementation, the class must be modified
  - If MyDependency has dependencies, they must be configured by the class
  - This implementation is difficult to unit test
    - The app should use a mock or stub MyDependency class, which isn't possible with this approach

What problems do direct dependencies create?

- Tightly coupled code
- Difficult to isolate and unit test the class functionality
- Difficult to maintain
  - If I change an X class how do I know what else will be affected? Did I break anything?
  - With unit tests in place this would not be a matter

Audience Question 2

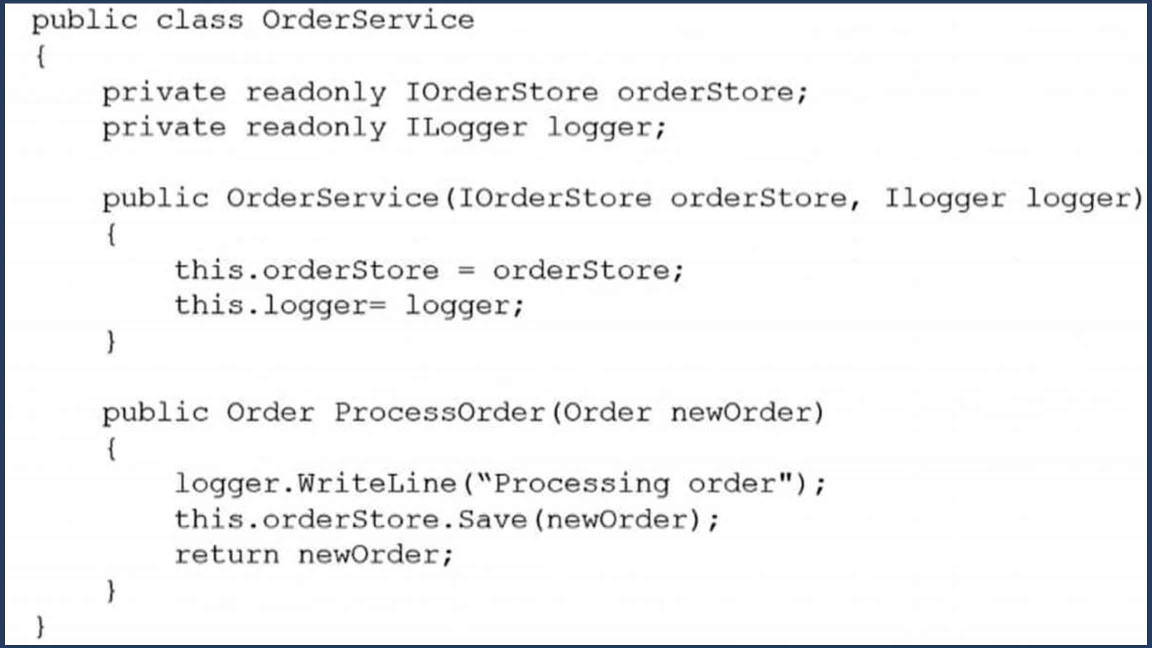
Which one of these classes is tightly coupled which one is loosely coupled to their dependencies?



```
public class OrderService
{
    private readonly IOrderStore orderStore;
    private readonly ILogger logger;

    public OrderService()
    {
        orderStore = new DbOrderStore(...);
        logger= new DbLogger(...);
    }

    public Order ProcessOrder(Order newOrder)
    {
        logger.WriteLine("Processing order");
        this.orderStore.Save(newOrder);
        return newOrder;
    }
}
```

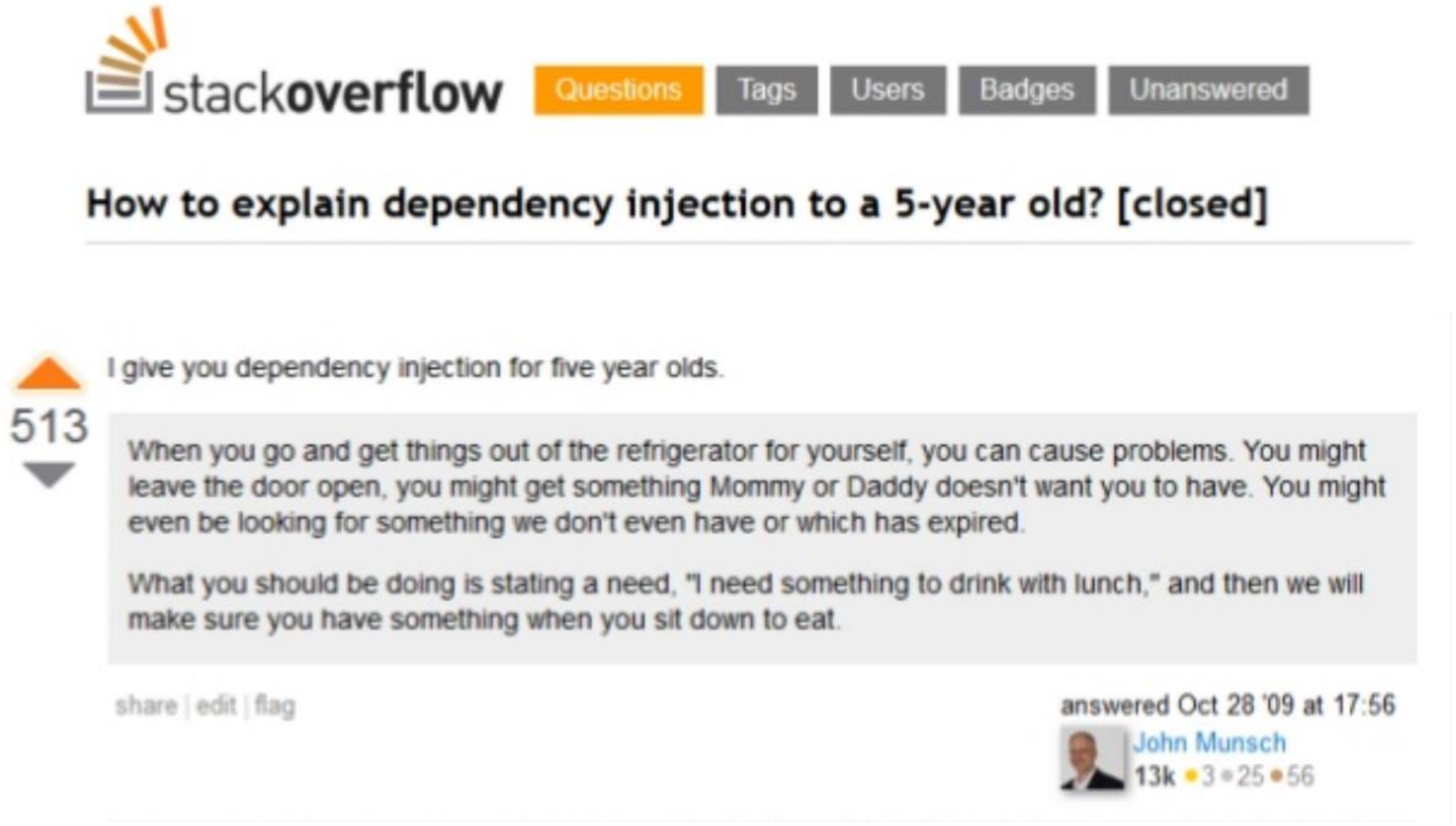


```
public class OrderService
{
    private readonly IOrderStore orderStore;
    private readonly ILogger logger;

    public OrderService(IOrderStore orderStore, Ilogger logger)
    {
        this.orderStore = orderStore;
        this.logger= logger;
    }

    public Order ProcessOrder(Order newOrder)
    {
        logger.WriteLine("Processing order");
        this.orderStore.Save(newOrder);
        return newOrder;
    }
}
```

What is Dependency Injection?



How to explain dependency injection to a 5-year old? [closed]

513

I give you dependency injection for five year olds.

When you go and get things out of the refrigerator for yourself, you can cause problems. You might leave the door open, you might get something Mommy or Daddy doesn't want you to have. You might even be looking for something we don't even have or which has expired.

What you should be doing is stating a need, "I need something to drink with lunch," and then we will make sure you have something when you sit down to eat.

share | edit | flag

answered Oct 28 '09 at 17:56
John Munsch
13k ●3 ●25 ●56

What is Dependency Injection?

- **Dependency Injection** is a **set of practices** that allow you to build **loosely coupled** and testable applications
- Architectural pattern which:
  - Results in decoupled code
  - Promotes testability
  - Makes parallel development easier
  - Results in more maintainable code
  - Typically accomplished with an IoC container

What is Dependency Injection?

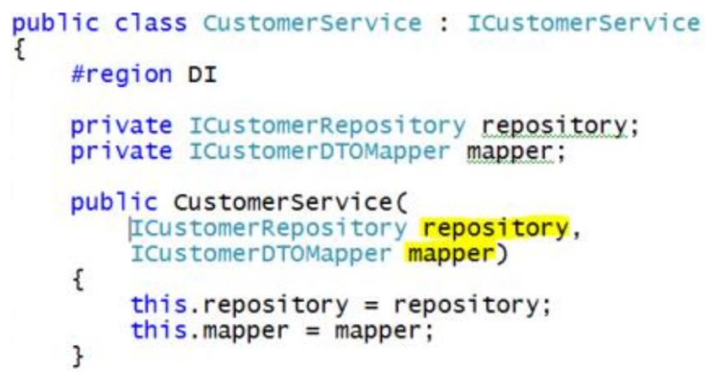
- What does **set of practices** actually mean?
  - It **IS**
    - A way of thinking
    - A way of designing code
    - General guidelines
  - It **IS NOT**
    - A library
    - A framework
    - A tool

What is Dependency Injection?

- What are **loosely coupled applications**?
  - Small components plugged together to form a bigger system, which are
    - Independent
    - Reusable
    - Interchangeable
- **Benefits** of loosely coupled applications
  - Small classes adhering to Single Responsibility Principle (S in SOLID)
  - Easier to maintain
  - Extensible
  - Testable

What is Dependency Injection?

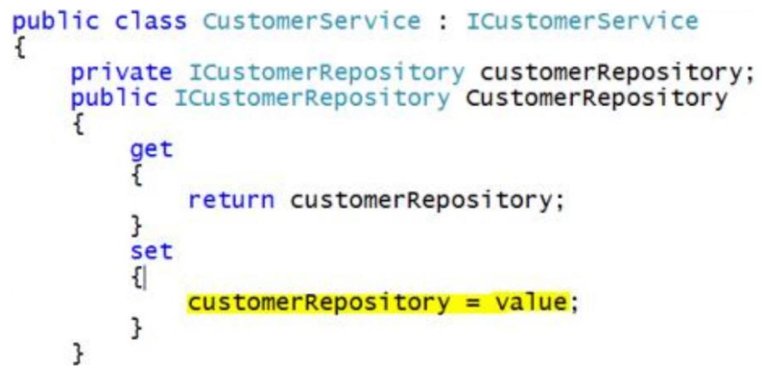
- Types of Dependency Injection
  - Constructor injection (most popular)
  - Property injection
  - Method injection



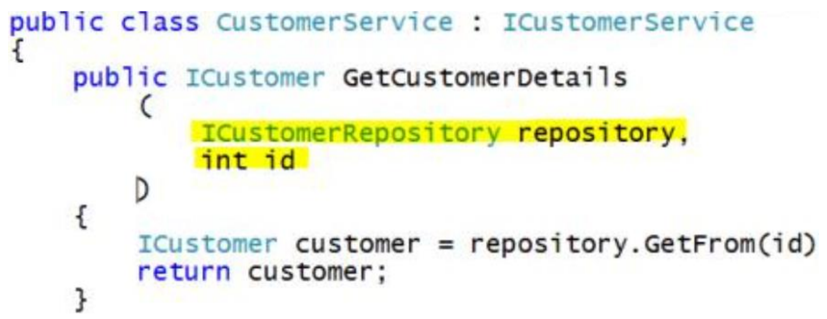
```
public class CustomerService : ICustomerService
{
    #region DI

    private ICustomerRepository repository;
    private ICustomerDTOMapper mapper;

    public CustomerService(
        ICustomerRepository repository,
        ICustomerDTOMapper mapper)
    {
        this.repository = repository;
        this.mapper = mapper;
    }
```



```
public class CustomerService : ICustomerService
{
    private ICustomerRepository customerRepository;
    public ICustomerRepository CustomerRepository
    {
        get
        {
            return customerRepository;
        }
        set
        {
            customerRepository = value;
        }
    }
```



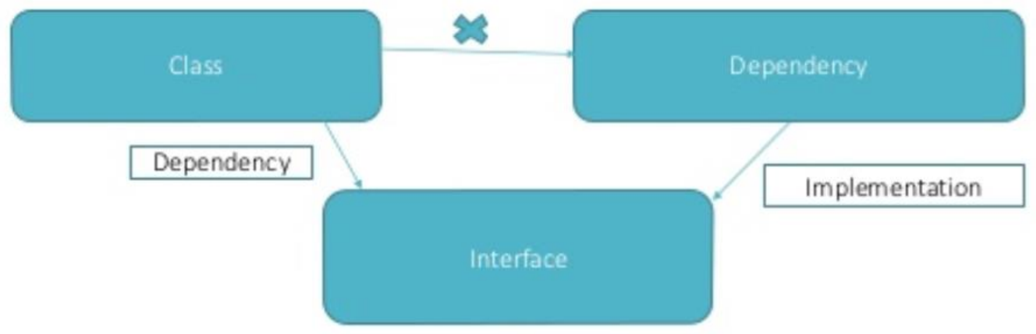
```
public class CustomerService : ICustomerService
{
    public ICustomer GetCustomerDetails
        (
            ICustomerRepository repository,
            int id
        )
    {
        ICustomer customer = repository.GetFrom(id)
        return customer;
    }
```

Audience Question 3

Which software principle does the Dependency Injection design pattern implement, in order to invert the creation of dependent objects?

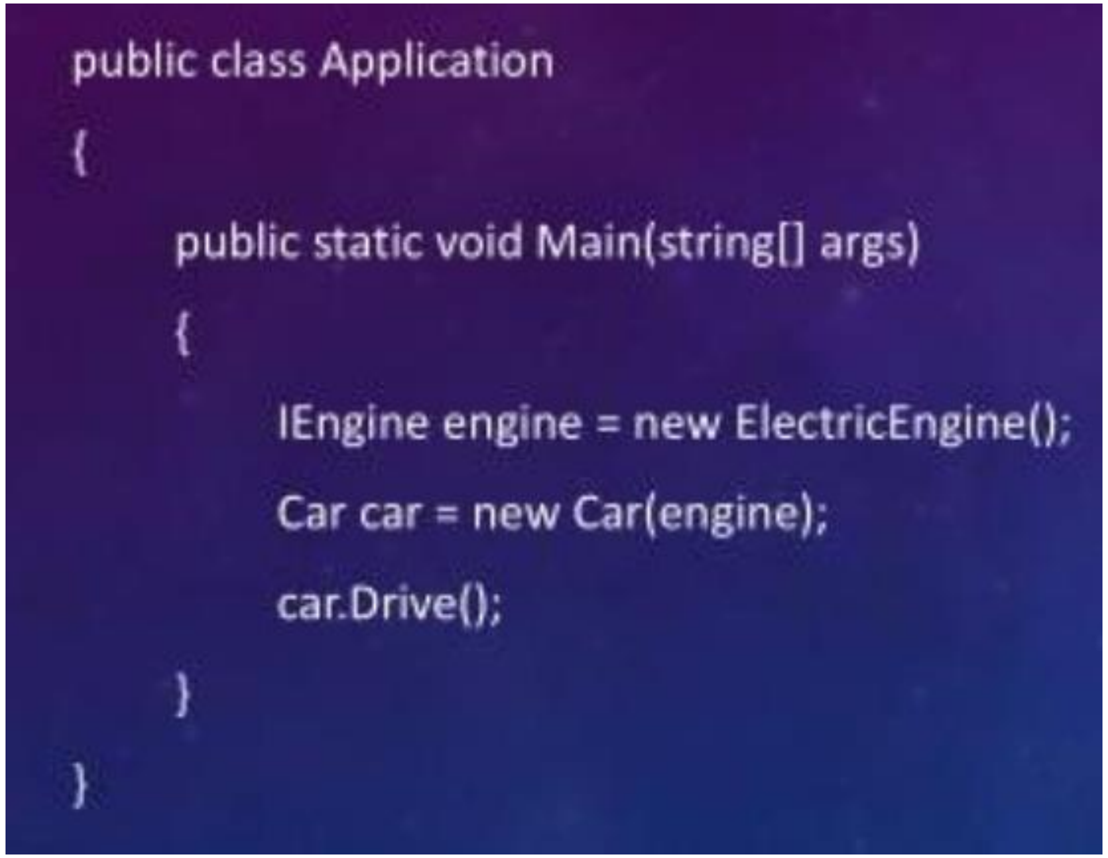
Inversion of Control (IoC)

- Referred to as the "**Hollywood Principle**"
  - "Don't call us, we will call you."
- Sometimes also referred to as **Dependency Inversion Principle** (D in SOLID)
  - Higher level modules should not depend on lower level modules. Both should depend on abstractions



Inversion of Control (IoC)

- Dependency Injection removes the need for building dependencies from the class which depends on them
- The problem is just shifted to the caller
- In the example, we need to know the concrete implementation of the IEngine interface at compile time
- With IoC, we delegate the management of the entire lifecycle of the objects to an **IoC (or DI) container**



```
public class Application
{
    public static void Main(string[] args)
    {
        IEngine engine = new ElectricEngine();
        Car car = new Car(engine);
        car.Drive();
    }
}
```

IoC (DI) Containers

- IoC Container (a.k.a. DI Container) is a framework for implementing automatic dependency injection
- It manages object creation and it's life-time and also injects dependencies to the class
- The IoC container creates an object of the specified class and also injects all the dependency objects through a constructor, a property or a method at run time and disposes it at the appropriate time
- This is done so that we don't have to create and manage objects manually

How IoC (DI) Containers work

- Mapping Abstractions to Concrete Types
  - Usually initialized on app start
  - Methods like Register<IAbstraction>().As<ConcreteType>()
- Method Resolve<TRequired>()
  - E.g. container.Resolve<IAbstraction>();
- Recursively resolves dependencies reading constructor parameters by using reflection

DI Basics

DI in NET Core - Overview

- Dependency injection is provided out of the box in ASP.NET Core web applications
- You register all your dependencies to the .NET Core **IoC service container** at application startup and get them injected in the client code when required
- ASP.NET Core provides a built-in service container called **IServiceProvider**
- Services are registered in the app's **Startup.ConfigureServices** method

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddScoped<IMyDependency, MyDependency>();

    services.AddRazorPages();
}
```

```
public class Index2Model : PageModel
{
    private readonly IMyDependency _myDependency;

    public Index2Model(IMyDependency myDependency)
    {
        _myDependency = myDependency;
    }

    public void OnGet()
    {
        _myDependency.WriteMessage("Index2Model.OnGet");
    }
}
```

DI in NET Core - Overview

- With the use of DI pattern in the current example
  - The Razor PageModel class doesn't use the concrete type MyDependency, only the interface IMyDependency
  - That makes it easy to change the implementation that the class uses without modifying it
  - The Razor PageModel class doesn't create an instance of MyDependency, it's created by the DI container

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddScoped<IMyDependency, MyDependency>();

    services.AddRazorPages();
}
```

```
public class Index2Model : PageModel
{
    private readonly IMyDependency _myDependency;

    public Index2Model(IMyDependency myDependency)
    {
        _myDependency = myDependency;
    }

    public void OnGet()
    {
        _myDependency.WriteMessage("Index2Model.OnGet");
    }
}
```

DI in NET Core - Overview

- Dependency injection can be used in a chained fashion
- Each requested dependency in turn requests its own dependencies
- The container resolves the dependencies in the graph and returns the fully resolved service
- The collective set of dependencies that must be resolved is typically referred to as a **dependency tree**, **dependency graph**, or **object graph**

```
//Startup.cs
public void ConfigureServices(IServiceCollection services)
{
    //with framework provided extension methods
    services.AddDbContext<MyDbContext>(opt => opt.UseInMemoryDatabase("MyDbName"));
    services.AddMvc(); //inject all services related to MVC
    //simple scoped service registration
    services.AddScoped<IRepository, Repository>();
    services.AddTransient<ISomeService, SomeService>();
    services.AddSingleton<IConfigBuilder, FileConfigBuilder>();
    //any type that needs injection/to be injected
    services.AddScoped<JustAnotherEntity>();
    //instance registration example, sp is IServiceProvider
    services.AddTransient<MyService>(sp => new MyService(
        "Some string argument",
        sp.GetService<ISomeService>()));
}
```

DI in NET Core - Overview

- The standard and ideal way to get a service instance injected in a class is to use **Constructor Injection**
  - Add the dependencies as parameter to the constructor of the class and then the **NET Core DI framework** will inject the actual instances as arguments at runtime
- Constructor injection is the ideal way to inject dependencies in a class as it is conventional, very descriptive and readable
  - It also makes the intention very clear that the code will not function if those dependencies are not provided

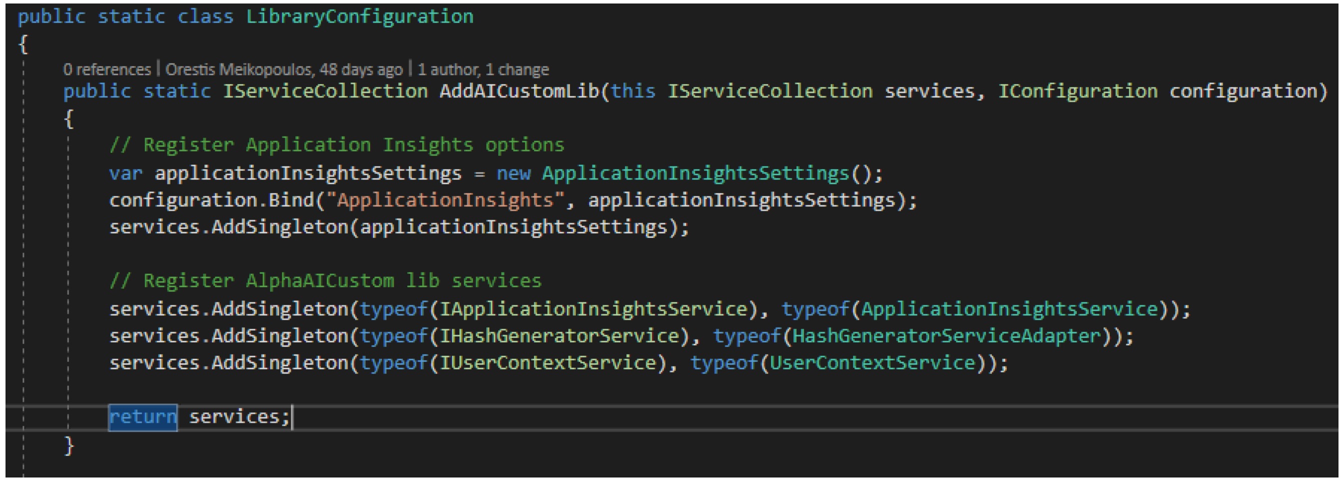
```
public class TestController : Controller
{
    ISomeService _someService;

    //constructor injection
    public TestController(ISomeService service)
    {
        _someService = service;
    }

    public IActionResult Get()
    {
        //use injected service
        var result = _someService.SomeMethod();
        return new OkObjectResult(result);
    }
}
```

DI in NET Core - Register services with extension methods

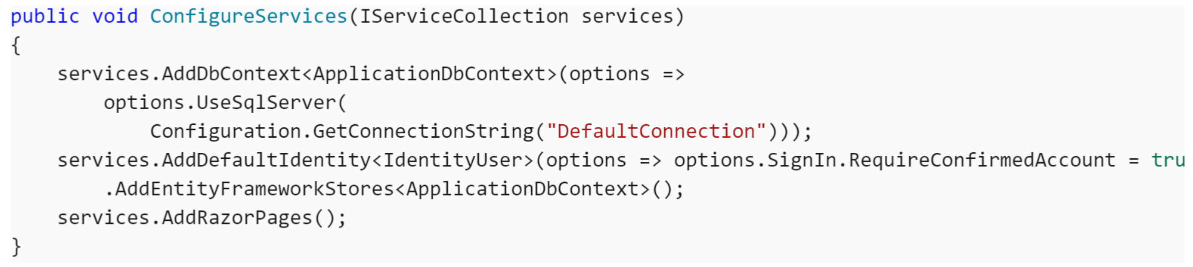
- Related groups of registrations can be moved to an extension method to register services
  - Each services.Add{SERVICE_NAME} extension method adds and potentially configures services
- For example
  - **AddControllersWithViews** adds the services that MVC controllers with views require
  - **AddRazorPages** adds the services that Razor Pages require



```
public static class LibraryConfiguration
{
    0 references | Orestis Meikopoulos, 48 days ago | 1 author, 1 change
    public static IServiceCollection AddAICustomLib(this IServiceCollection services, IConfiguration configuration)
    {
        // Register Application Insights options
        var applicationInsightsSettings = new ApplicationInsightsSettings();
        configuration.Bind("ApplicationInsights", applicationInsightsSettings);
        services.AddSingleton(applicationInsightsSettings);

        // Register AlphaAICustom lib services
        services.AddSingleton(typeof(IApplicationInsightsService), typeof(ApplicationInsightsService));
        services.AddSingleton(typeof(IHashGeneratorService), typeof(HashGeneratorServiceAdapter));
        services.AddSingleton(typeof(IUserContextService), typeof(UserContextService));

        return services;
    }
```



```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = tru
        .AddEntityFrameworkStores<ApplicationDbContext>();
    services.AddRazorPages();
}
```

DI in NET Core - Service Lifetimes

- **Transient**
  - Transient lifetime services (**AddTransient**) are created each time they're requested from the service container
  - This lifetime works best for lightweight, stateless services
  - In apps that process requests, transient services are disposed at the end of the request
- **Scoped**
  - Scoped lifetime services (**AddScoped**) are created once per client request (connection)
  - In apps that process requests, scoped services are disposed at the end of the requests

DI in NET Core - Service Lifetimes

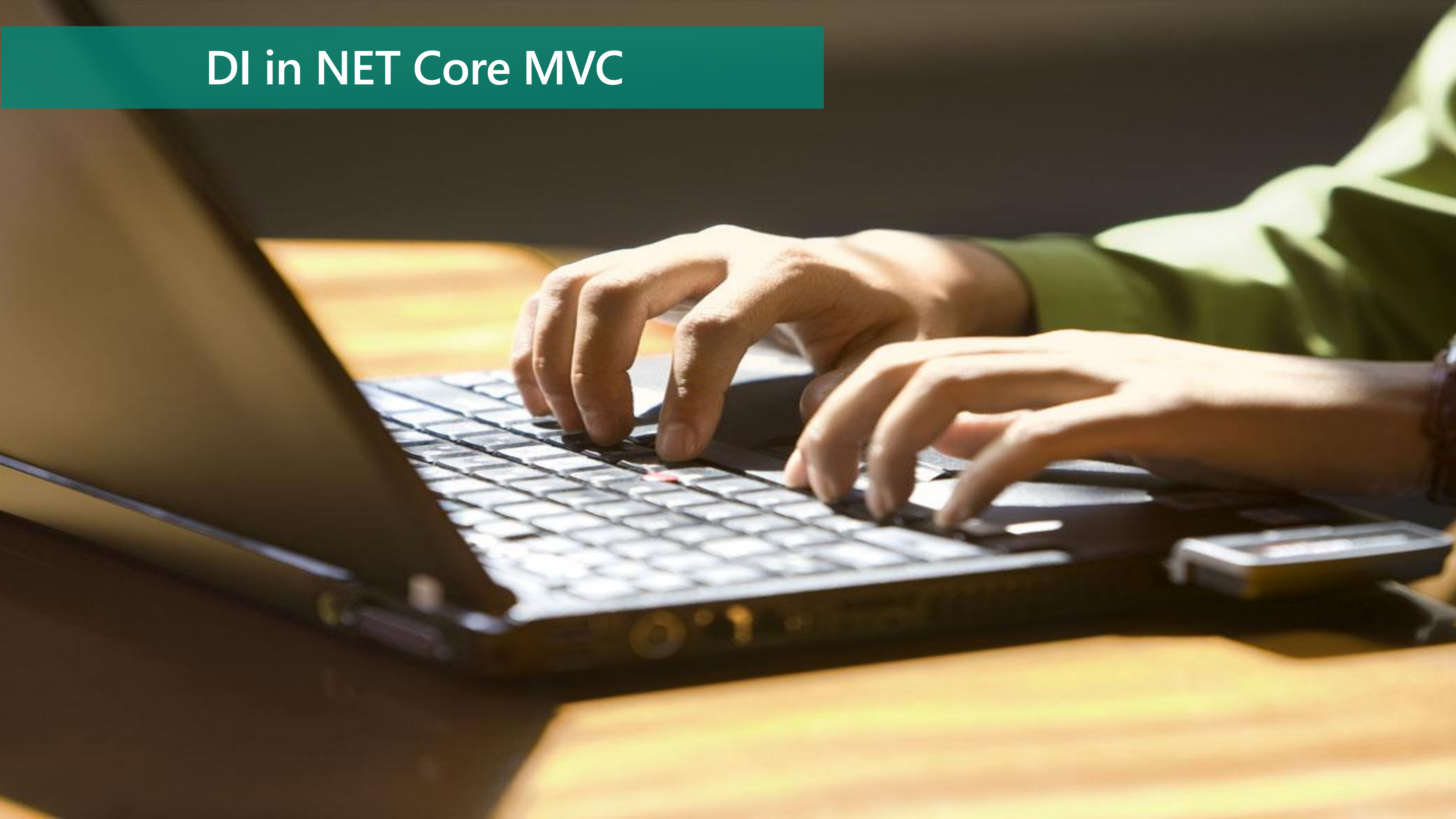
- **Singleton**
  - Singleton lifetime services (**AddSingleton**) are created the first time they're requested
  - Every subsequent request uses the same instance
  - If the app requires singleton behavior, allow the service container to manage the service's lifetime
    - Don't implement the singleton design pattern and provide code to dispose of the singleton
  - In apps that process requests, singleton services are disposed when the ServiceProvider is disposed on application shutdown
    - Because memory is not released until the app is shut down, memory use with a singleton must be considered

DI Service Lifetimes

DI in NET Core - Best Practices

- Design services to use dependency injection to obtain their dependencies
- Avoid stateful, static classes and members
- Avoid direct instantiation of dependent classes within services
  - Direct instantiation couples the code to a particular implementation
- Make app classes small, well-factored, and easily tested
- A class seems with too many injected dependencies, maybe has too many responsibilities and is violating the **Single Responsibility Principle (SRP)**
  - Attempt to refactor the class by moving some of its responsibilities into a new class
  - Keep in mind that **Razor Pages page model** classes and **MVC controller classes** should focus on **UI concerns**

DI in NET Core MVC



Microsoft

Thank You

- For the opportunity
- For participating
- For listening

© Copyright Microsoft Corporation. All rights reserved.