Dependency Injection

.NET

*A **dependency** is anything that an object requires in order to function properly. The **Dependency injection (DI)** design pattern is a technique used to achieve **Inversion of Control (IoC)** between classes and their dependencies.*

HTTPS://DOCS.MICROSOFT.COM/EN-US/ASPNET/CORE/FUNDAMENTALS/DEPENDENCY-INJECTION?VIEW=ASPNETCORE-5.0

Dependencies Explained

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-5.0

IndexModel class depends on functionality provided by the ***MyDependency*** class.

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

    public async Task OnGetAsync()
    {
        await _dependency.WriteMessage(
            "IndexModel.OnGetAsync created this message.");
    }
}
```

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

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

        return Task.FromResult(0);
    }
}
```

An instance of the ***MyDependency*** class can be created in the ***IndexModel*** class to make WriteMessage() available to that class.

Dependency Inversion – Overview

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-5.0

Code dependencies can be problematic and should be avoided.

- To replace ***MyDependency*** with a different implementation, the class would have to be modified.
- If ***MyDependency*** has dependencies, they must be configured by the class.
- The below implementation is difficult to unit test.

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

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

        return Task.FromResult(0);
    }
}
```

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

    public async Task OnGetAsync()
    {
        await _dependency.WriteMessage(
            "IndexModel.OnGetAsync created this message.");
    }
}
```

Service Type Lifetimes

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-5.0#transient

Service	Description
Transient	***Transient*** services (AddTransient) are created each time they're requested from the service container. Best for lightweight, stateless services.
Scoped	***Scoped*** lifetime services (AddScoped) are created once per HTTP request (connection).
Singleton	***Singleton*** services (AddSingleton) are created the first time they're requested (or when Startup.ConfigureServices() is run). Every subsequent request uses the same instance.

Dependency Injection – “This is the way”

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-5.0

Dependency injection helps provide services that classes need:

- ASP.NET Core provides a built-in ***service container*** called ***IServiceProvider***.
- Dependencies are registered in Startup.ConfigureServices().
- An ***interface*** (or base class) is used to abstract the dependency implementation.
- Inject the service into the constructor of the class where it's used.

.NET framework takes on the responsibility of creating an instance of the dependency and disposing of it when it's no longer needed.

Dependency Injection – Step by Step(1/2)

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-5.0

```
public interface IMyDependency
{
    Task WriteMessage(string message);
}
```

1) Create an interface where you declare a method that you want to make available through Dependency Injection.

2)Define the method in a class that implements the Interface.

```
public class MyDependency : IMyDependency
{
    private readonly ILogger<MyDependency> _logger;

    public MyDependency(ILogger<MyDependency> logger)
    {
        _logger = logger;
    }

    public Task WriteMessage(string message)
    {
        _logger.LogInformation(
            "MyDependency.WriteMessage called. Message: {MESSAGE}",
            message);

        return Task.FromResult(0);
    }
}
```

Dependency Injection – Step by Step(2/2)

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-5.0

```
public void ConfigureServices(IServiceCollection services)
{
    se

    services.AddScoped<IMyDependency, MyDependency>();
    services.AddTransient<IOperationTransient, Operation>();
    services.AddScoped<IOperationScoped, Operation>();
    services.AddSingleton<IOperationSingleton, Operation>();
    services.AddSingleton<IOperationSingletonInstance>(new Operation(Guid.Empty));

    // OperationService depends on each of the other Operation types.
    services.AddTransient<OperationService, OperationService>();
}
```

3)Add the dependency to ***ConfigureServices()*** with:
services.[desiredScope]<[interface], [class]>();

4) Inject the dependency into the constructor of the dependent class and assign it to a ***private*** variable of the ***interface*** type.

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

    public IndexModel(
        IMyDependency myDependency,
        OperationService operationService,
        IOperationTransient transientOperation,
        IOperationScoped scopedOperation,
        IOperationSingleton singletonOperation,
        IOperationSingletonInstance singletonInstanceOperation)
    {
        _myDependency = myDependency;
        OperationService = operationService;
        TransientOperation = transientOperation;
        ScopedOperation = scopedOperation;
        SingletonOperation = singletonOperation;
        SingletonInstanceOperation = singletonInstanceOperation;
    }

    public OperationService OperationService { get; }
    public IOperationTransient TransientOperation { get; }
    public IOperationScoped ScopedOperation { get; }
    public IOperationSingleton SingletonOperation { get; }
    public IOperationSingletonInstance SingletonInstanceOperation { get; }

    public async Task OnGetAsync()
    {
        await _myDependency.WriteMessage(
            "IndexModel.OnGetAsync created this message.");
    }
}
```

Alternative to Constructor Injection [FromServices] Attribute

https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/dependency-injection?view=aspnetcore-5.0#action-injection-with-fromservices

---

After registering a service with the ***service container***, the [FromServices] attribute enables injecting the registered service directly into an ***Action Method*** without using constructor injection in the ***Controller***.

```
public IActionResult About([FromServices] IDateTime dateTime)
{
    ViewData["Message"] = $"Current server time: {dateTime.Now}";

    return View();
}
```

Alternative to Constructor Injection .GetService<>()

https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/dependency-injection?view=aspnetcore-5.0#action-injection-with-fromservices

---

If you are unable to obtain an instance of a registered service by **Dependency Injection**, .GetService<> can be used to get a service object.

```
public class MyClass()
{
    public void MyMethod()
    {
        var optionsMonitor =
            _services.GetService<IOptionsMonitor<MyOptions>>();
        var option = optionsMonitor.CurrentValue.Option;

        ...
    }
}
```

*Do not invoke GetService to obtain a service instance when you can use DI instead.

Dependency Injection - Examples of Scopes.

```
public void ConfigureServices(IServiceCollection services)
{


    services.AddScoped<IMyDependency, MyDependency>();
    services.AddTransient<IOperationTransient, Operation>();
    services.AddScoped<IOperationScoped, Operation>();
    services.AddSingleton<IOperationSingleton, Operation>();
    services.AddSingleton<IOperationSingletonInstance>(new Operation(Guid.Empty));

    // OperationService depends on each of the other Operation types.
    services.AddTransient<OperationService, OperationService>();
}
```

Dependency Injection - .addDbContext

https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.dependencyinjection.entityframeworkservicecollectionextensions.adddbcontext?view=efcore-5.0
https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-5.0#entity-framework-contexts

---

The ***Entity Framework*** context is usually added to the ***service container*** using the ***scoped*** lifetime because web app database operations are normally scoped to the client request. The default lifetime is ***scoped*** if a lifetime isn't specified by an *AddDbContext<TContext>()* overload when registering the database context. Services of a given lifetime shouldn't use a database context with a shorter lifetime than the service.

```
public void ConfigureServices(IServiceCollection services)
{
    ...

    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    ...
}
```

DI – Best Practices (1/2)

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-5.0#design-services-for-dependency-injection

---

Best design practices are to:

- Design services to use ***dependency injection*** to obtain their dependencies.
- Avoid stateful, static classes and members.
- Design apps to use ***singleton*** services, which avoid creating global state.
- Avoid direct instantiation of dependent classes within services. Direct instantiation couples the code to a particular implementation.
- Make classes small, well-factored, and easily tested.
- If a class seems to have too many injected dependencies, it's a sign that the class has too many responsibilities and is violating the ***Single Responsibility Principle (SOLID)***.

DI – Best Practices (1/2)

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-5.0#design-services-for-dependency-injection
https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-5.0#recommendations

- ***IServiceProvider*** requires a public constructor for Dependency Injection.
- ***Dependency Injection*** is an alternative to static or ***global*** object access patterns. You may not be able to realize the benefits of ***DI*** if you mix it with static object access.
- The ***IServiceProvider service container*** is designed to serve the needs of most consumer apps. Use the ***IServiceProvider container*** unless you need a specific feature that ***IServiceProvider*** doesn't support.