ASP.NET: Dependency Injection

Dependency Injection

When a *service* (dependency object) is created outside of its *client* (the object that depends on it) that service can be passed into, or *injected*, into the client, typically through the client's constructor.
This process is called *dependency injection*, where services are not created by the clients that use (and depend on) them, but rather are created and managed in other code, and are *injected* into the client.

```
//The client class that depends on a service
class Reviewer
{
  public EmailSender _sender {get; set;}

  //EmailSender is injected into the Reviewer object
  public Reviewer(EmailSender sender)
  {
    _sender = sender;
  }
  public SendReview(string lesson, string comments)
  {
    _sender.SendReview(lesson, comments);
  }
}

//The dependency/service class
class EmailSender
{
  public SendReview(string lesson, string comments)
  {
    // Send an email with the Lesson and Comments
  }
}

static void Main(string[] args)
```

```
{
  EmailSender sender = new EmailSender();

  //Injecting Sender into Reviewer
  Reviewer reviewer = new Reviewer(Sender).
    SendReview("Dependecy Injection", "Super helpful!");
}
```

IoC Container

The *IoC Container* (Inversion of Control Container) is a framework that acts as the dependency injector. This allows the programmer to focus on using the service within the classes that depend on it, rather than managing the entire life cycle of the service. The IoC Container does all of the following:

1. Registers services with a concrete implementation (a class)
2. Instantiating, or resolving, the service class to be injected into the client class
3. Injecting the service
4. Disposing of the service instance based on the registered settings

Registering Services

Services are registered in the `ConfigureServices()` method of the `Startup` class in **Startup.cs**.

Once the services are registered, they are available for injection into client classes that use those services.

Services are registered using the `AddTransient()`, `AddScoped()`, and `AddSingleton()` methods. These methods dictate how the service's life cycle is managed.

- `AddTransient` - the service is created each time it's requested from the IoC Container. This means that when more than one class uses the service, those classes will be injected with a fresh new instance of that service, even if it's within the same request.
- `AddScoped` - the service is created for each client request. If multiple classes use and are injected with the service within the same request, only one instance of that service is created and used throughout the request for those classes.
- `AddSingleton` - the service is created once on the first time the service is requested and is instantiated for the lifetime of the application process.

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

  services.AddTransient<ITransientService, TransientService>
();
  services.AddScoped<IScopedService, ScopedService>();
  services.AddSingleton<ISingletonService, SingletonService>
();
}
```

AddDbContext

The `AddDbContext<T>()` method registers the framework-provided services that allow all page models to be injected with an instance of the `T` database context that will be used to access the application's database.

When calling the `AddDbContext<T>()` method to register the application's database context service, one must also pass in an instance of the `DbContextOptions` object that contains information such as the database provider type (`UseSqlServer()`, `UseSqlite()`, etc.), connection string (defined in **appsettings.json**), and other optional settings that defines the behavior of the context.

The `AddDbContext<T>()` method is called within `Startup.ConfigureServices()`.

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddDbContext<MyAppContext>(options =>
    options.UseSqlServer(
      Configuration.GetConnectionString("MyAppContext")));
}
```

Dependencies As Interfaces

In order to satisfy the Dependency Inversion Principle, injected services are typically referenced as *interfaces*. This allows the client class to use an implemented service whose behavior is well defined via its interface, and not have any knowledge or be concerned with the actual concrete class that implements that interface.
Any changes to the concrete class's methods or properties would not require any modifications to the client class since it only knows of the interface and its well defined abstract methods.

```
public class ReviewModel : PageModel
{
  // Notice IFormSender interface
  private readonly IFormSender _Sender;

  [BindProperty]
  public string Review {get;set;}

  [BindProperty]
  public int ProductID {get;set}

  // Notice IFormSender interface
  public ReviewModel(IFormSender sender)
  {
    _Sender = sender;
  }

  public async Task<IActionResult> OnPost()
  {
    await _Sender.SubmitReview(ProductID, Review);
    return RedirectToPage("/Index");
  }
}
```

Dependency

When one object (**Object A**) references another object (**Object B**), using its properties and methods, it means that the first object (**A**) *depends* on the second (**B**), making the second object (**B**) a *dependency*.

```
// Object A - the class that depends on Object B
class Reviewer
{
  private EmailSender Sender = new EmailSender();
  public SendReview(string lesson, string comments)
  {
    Sender.SendReview(lesson, comments);
  }
}

// Object B - the dependency
class EmailSender
{
  public SendReview(string lesson, string comments)
  {
    // Send an email with the Lesson and Comments
  }
}

static void Main(string[] args)
{
  Reviewer reviewer = new Reviewer().
    SendReview("Dependecy Injection", "Learned a ton!");
}
```

AddRazorPages()

Following the `services.Add{ServiceName}` naming convention, the `AddRazorPages()` method registers all the services required for the web app to function as a Razor Pages application.

`AddRazorPages()` is called within `Startup.ConfigureServices()`.

If you're curious and want to peek under the hood to see all the services that are registered within `AddRazorPages()`, you can find the code at dotnet/aspnetcore. Remember, it's all open source!

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddRazorPages();
}
```

DI and IOC Container

A built-in *IoC Container* (Inversion of Control Container) is provided with ASP.NET that implements all dependency injection functionality and allows the developer to implement structured code following the Dependency Inversion (DIP) and SOLID principles.

The IoC Container allows the developer to register services for injection in the `Startup.ConfigureServices()` method.

The IoC Container handles the lifecycles of services and lets the developer implement classes that use the registered services to perform work.

```
//Registering the Service
public class Startup
{
  public void ConfigureServices(IServiceCollection services)
  {
    services.AddRazorPages();
    services.AddScoped<ISendService, SendService>();
  }
}

//Injecting the service
public class ReviewModel : PageModel
{
  private readonly ISendService _sender;
  public ReviewModel(ISendService sender)
  {
    _sender = sender;
  }
}
```

Print Share