Angular
Lab 7
Dependency Injection (DI) and Services

1 Lab setup

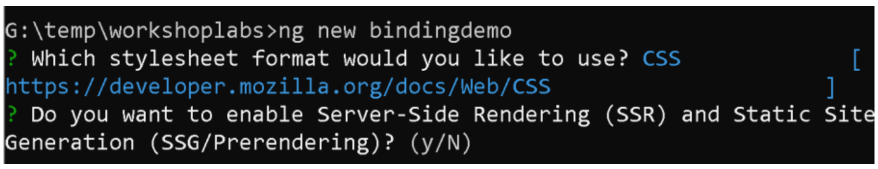
The setup here is identical to that in Lab 1

2 Generating a new Angular project

In either a separate command prompt (or shell terminal) or in an embedded terminal in Visual Studio Code, create a new Angular project with:

```
ng new servicesdidemo
```

Press enter to accept the default values for all the question prompts that follow regarding stylesheet format (CSS) and enabling Server side rendering (No)



```
G:\temp\workshoplabs>ng new bindingdemo
? Which stylesheet format would you like to use? CSS             [
https://developer.mozilla.org/docs/Web/CSS                       ]
? Do you want to enable Server-Side Rendering (SSR) and Static Site
Generation (SSG/Prerendering)? (y/N)
```

Navigate into the root project folder from the terminal with:

```
cd servicesdidemo
```

Build the app and serve it via Angular's live development server by typing:

```
ng serve
```

The final output line from this should indicate that the compilation process was successful and identify the port that the live development server is currently serving up the Angular app dynamically from (by default this will be port 4200)

Open a browser tab at:
http://localhost:4200/

You should be able to see the default landing page for all new autogenerated Angular projects.

To stop the development server, type Ctrl+C in the terminal window that it is running in. You can restart again anytime by typing `ng serve` in the project root folder.

3 Creating and using a basic service without DI

The source code for this lab can be found in the `changes` subfolder of `ServicesDI-Demo`

Modify / add the following files in `src/app` with the latest update from `changes`:

```
app.component-v1.html

app.component-v1.ts
```

Create the following new files (`product.ts` and `product.service.ts`)

```
product-v1.ts

product.service-v1.ts
```

Services are classes with basic methods that are intended to be reused at various points in the application. Their functionality is very focused and specific, in accordance to the Single Responsibility Principle (SRP) of software engineering.

https://www.freecodecamp.org/news/solid-principles-single-responsibility-principle-explained/

Components should only implement UI-specific logic (i.e. related to what users see in a view and how they interact with it). Any other functionality and logic should be delegated to a separate service class. Examples of functionality that might be implemented by a service class are:

a) Reusable core business logic (for e.g. calculating sales tax on a purchase)
b) Logging output to track key events in the lifecycle of a complex application
c) Retrieving / storing data from / to a backend server application via HTTP calls

This basic implementation of a service has a few major issues:

The ProductService instance is tightly coupled to the component that it is used in. If we wish to use a different ProductService class with different functionality (for e.g. one that reads data from a backend server), we would need to change the code in the component. We make also wish to provide a mock version of ProductService class for testing purposes, which would also require changing code in the component.

Dependency injection (DI) provides a mechanism whereby the required services (DI) are provided correctly at run time by Angular without requiring tight coupling to the components that they are used in.

4 Configuring and using a service via DI

DI framework terminology:

- A dependency is anything a class needs in order to perform correctly. Typically, this will be a service, but can also be a simple function or even a basic value.
- The consumer is the class that requires a dependency. Typically this will be a component, but can also be a directive or even another service
- A dependency provider (or simply provider) is used to configure an injector with a DI token. This token uniquely identifies the provider. The provider also provides information on how to create a dependency value / instance.
- Injectors are responsible for resolving and instantiating the dependencies and injecting them into consumers. It uses the DI token it has being configured with from the provider to locate provider which in turn provides it the information on how instantiate the dependency.
- Angular provides an instance of an Injector and a provider to all potential consumers in the application via Hierarchical dependency injection

4.1 Providing the service at the component level

Modify the following files in `src/app` with the latest update from `changes`:

`app.component-v2.ts`

The `providers` array metadata can be provided in the @Component, @NgModule and @Directive decorators to declare providers. The simplest and most common way to register a provider is to simply use the class name of the dependency / service associated with the provider directly in the `providers` array (this is known as the class provider syntax). In this situation, we can simply say we are registering the dependency.

The consumers (components, directives and other services) declare the dependencies that they need in their constructor, typically alongside an access modifier (usually public or private). The injector for that consumer then reads these dependencies and attempts to resolve them from the provider specified in the providers array metadata using the DI token. It then instantiates the dependency based on the instructions provided by the provider and then injects this instance into the consumers.

4.2 Providing the service in the application config file

Modify the following files in `src/app` with the latest update from `changes`:

`app.component-v2-2.ts`

`app.config-v2-2.ts`

Here, we have removed the registration of the service from the component itself and instead shifted it to the providers metadata array of the main application config, which is referenced in bootstrapping the application together with AppComponent in `main.ts`

4.3 Providing service at the application root level using providedIn

Modify the following files in `src/app` with the latest update from `changes`:

`app.config-v2-3.ts`

`product.service-v2-3.ts`

Here, we are registering the service at the application root as well, but via the @Injectable decorator in the service class itself. This is the most common way to provide a service in an application, and any newly generated services (to be covered later) will use this approach.

4.4 Alternative way to inject a service

Modify the following files in `src/app` with the latest update from `changes`:

`app.component-v2-4.ts`

Angular 17 provides an alternative syntax to constructor injection to introduce a service dependency into a component.

References:

https://www.tektutorialshub.com/angular/angular-dependency-injection/

https://angular.io/guide/architecture-services

5 Services that use other services

Open a new command prompt and navigate into the root project folder from the terminal with:

```
cd servicesdidemo
```

You can use the Angular CLI to generate a new service with:

```
ng generate service Logger
```

A short form for the above uses only the first letter of the commands, i.e.

```
ng g s service-name
```

This creates a new `logger.service.ts` file in `src/app`. Notice that the service class uses the @Injectable decorator to register itself at the application level. This is the default for all newly generated services.

Modify the following files in `src/app` with the latest update from `changes`:

`logger.service-v3.ts`

`product.service-v3.ts`

When the app is reloaded, you should be able to see the various log messages appearing in the Console tab of Chrome Developer Tools.

6 Services are singleton by default

In the command prompt in the root project folder `servicesdidemo`, generate 2 new components using the Angular CLI:

```
ng generate component firstChild
```

```
ng generate component secondChild
```

We generate a new service with:

```
ng generate service Basic
```

Modify the following files in `src/app` with the latest update from `changes`:

`app.component-v4.html`

`app.component-v4.ts`

`basic.service-v4.ts`

`first-child.component-v4.ts`

`first-child.component-v4.html`

`second-child.component-v4.ts`

`second-child.component-v4.html`

The service BasicService is accessed in both the FirstChild and SecondChild component via constructor injection. Notice that when you set a new value for the single number property in BasicService in either component, calling getNum on the service in other component returns the newly set value. In other words, there is only a single instance of BasicService that is created

The dependencies / services are singletons within the scope of an injector. When the injector gets a request for a particular service for the first time, it creates a new instance of the service. For all the subsequent requests, it will return the already created instance.

References:

https://www.tektutorialshub.com/angular/angular-singleton-service/