Angular
Lab 7
Dependency Injection (DI) and Services

1 Lab setup

Make sure you have the following items installed

- Latest version of Node and NPM (note: labs are tested with Node v16.13 and NPM v8.1 but should work with later versions with no or minimal changes)
- Latest version of TypeScript (note: labs are tested with TypeScript v4.5.4 but should work with later versions with no or minimal changes)
- Latest version of Angular (note: labs are tested with Angular v13.1 but should work with later versions with no or minimal changes)
- Visual Studio Code (or a suitable alternative IDE for Angular/TypeScript)
- A suitable text editor (Notepad ++)
- A utility to extract zip files (7-zip)

In each of the main lab folders, there are two subfolders: `changes` and `final`.

- The `changes` subfolder holds the source code and other related files for the lab. The creation of the Angular app will proceed in a step-wise fashion in order to clearly demonstrate the various features of the framework. The various project source files in this folder are numbered in a way to reflect this gradual construction. For e.g. `xxx.v1.html`, `xxx.v2.html`, etc represents successive changes that are to be made to the `xxxx.html`

- The `final` subfolder holds the complete Angular project. If there was some issue in completing the step-wise construction of the app using the files from the `changes` subfolder, you can use this to run the final working version.

The project folder in `final` is however missing the crucial `node_modules` subfolder which contains all the dependencies (JavaScript libraries) that your Angular app needs in order to be built properly. These dependencies are specified in the `package.json` file in the root project folder. They are omitted in the project commit to GitHub because the number and size of the files are prohibitively large.

In order to run the app via the Angular development server (`ng serve`), you will need to create this subfolder containing the required dependencies. There are two ways to accomplish this:

a) Copy the project root folder to a suitable location on your local drive and run `npm install` in a shell prompt in the root project folder. NPM will then create the `node_modules` subfolder and download the required dependencies as specified in `package.json` to populate it. This process can take quite a while to complete (depending on your broadband connection speed)

b) Alternatively, you can reuse the dependencies of an existing complete project as most of the projects for our workshop use the same dependencies. In this case, copy the `node_modules` subfolder from any existing project, and you should be able to immediately start the app. If you encounter any errors, this probably means that there are some missing dependencies or Angular version mismatch, in which case you will then have to resort to `npm install`.

The lab instructions here assume you are using the Chrome browser. If you are using a different browser (Edge, Firefox), do a Google search to find the equivalent functionality.

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 routing and stylesheet.

Navigate into the root project folder from the terminal with:

```
cd servicesdidemo
```

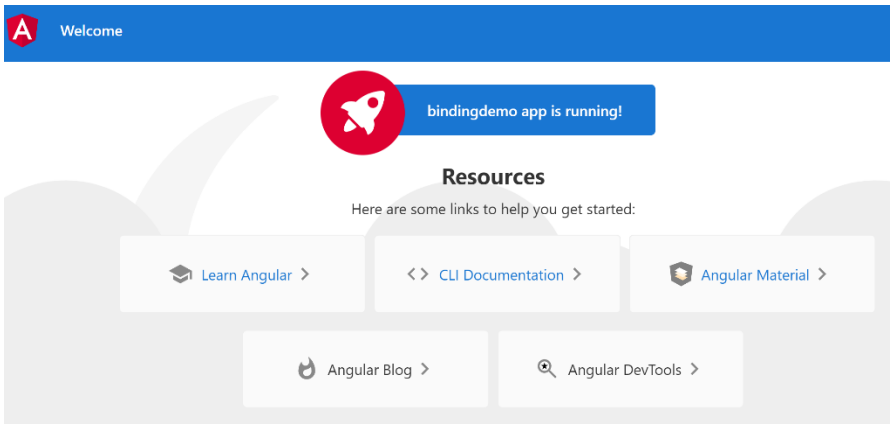
Build the app and serve it via Angular's live development server by typing:

```
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`

`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) Logging output to track key events in the lifecycle of a complex application
b) Reusable core business logic (for e.g. calculating sales tax on a purchase)
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

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.

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

`app.component-v2-2.ts`

`app.module-v2-2.ts`

Here, we are registering the service in the providers metadata array of the root module, making it available to all components, services and directives within the module itself.

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

`app.module-v2-3.ts`

`product.service-v2-3.ts`

Here, we are registering the service at the root module as well, but via the @Injectable decorator in the service class itself. This is the most common to provide a service in a module.

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
```

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 root module 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`

`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/