Angular
Lab 6
Lifecycle Hooks

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 lifecycledemo
```

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 lifecycledemo
```

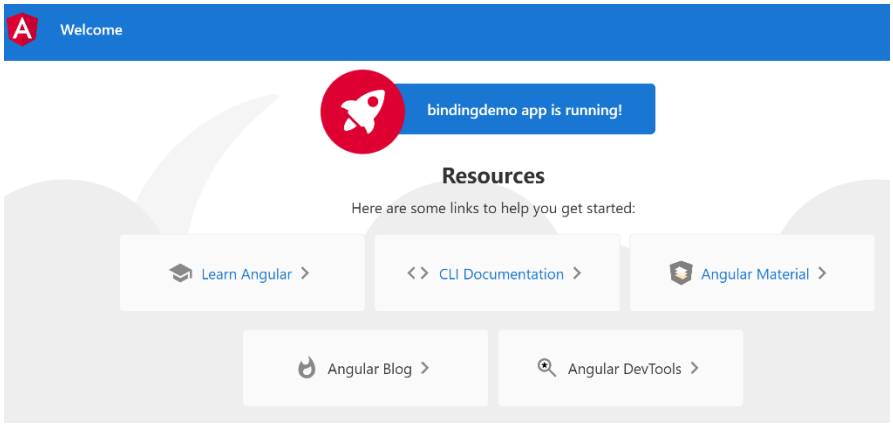
Build the app and serve it via Angular's live development server by typing:

```
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 Life cycle hooks for a single component

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

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

`app.component-v1.ts`

`app.component-v1.html`

A component has a lifecycle that starts when it is instantiated and its views (along with its child views) are rendered. This lifecycle continues with change detection, where updates are made to both the view and component as needed. The lifecycle ends with the destruction of the component and removing its rendered template from the DOM. Directives have a similar lifecycle as well.

Change detection is about updating (or re-rendering) the view (DOM) when the state of the application has changed.

The default change detection strategy is to check every single component for state change, traversing the component tree downwards starting from the root component. This change detection check is triggered by any of the following scenarios which can potentially change component state and which can occur in any component at any level in the tree:

1. Component initialization – for example, when bootstrapping an Angular application
2. Standard browser events such as keystroke, mouse clicks which are picked up by the DOM event listener
3. Getting a response from an XML HttpRequest (XHR) - which is used in the background by HttpClient
4. Timer functionality: setTimeout / setInterval may both have call back functions that change data
5. Promise objects: When asynchronous APIs return a Promise object (such as fetch), the then() callback function can also update the data.

When a state change is detected, the changes are propagated to the corresponding template via the specific binding used and the view corresponding to that template is re-rendered to reflect these latest changes

Lifecycle hooks are methods that can be used to tap into each of the key events in the life cycle of a component in order to perform relevant actions at that point (for e.g. perform specific initialization operations when a component is instantiated, or perform cleanup when a component is destroyed). To incorporate a lifecycle hook, the component can implement one of the interfaces associated with that hook. However, this implementation is optional, as long as the hook method is present it will be executed at that particular key event: the use of the interface is to mandate the implementation of the corresponding hook during compile time checking. The name of the hook is the name of the interface prefixed with ng (for e.g interface OnInit requires implementation of the ngOnInit hook).

Angular calls hook methods in a specific sequence
https://angular.io/guide/lifecycle-hooks#lifecycle-event-sequence

https://www.tektutorialshub.com/angular/angular-component-life-cycle-hooks/#the-order-of-execution-of-life-cycle-hooks

Key points to note:

- The bootstrap process consists of two key phases: constructing the component (or view) tree and then performing change detection. The constructor is called during the construction of the tree, and ngOnInit (with all other lifecycle hooks) are called during the change detection phase (after the tree has been constructed). Thus, the constructor will always run before ngOnInit.
- The constructor is the ideal place to perform dependency injection and also to initialize members of the class.
- The ngOnInit hook is used for all other initialization operations not covered in the constructor - such as making HttpClient calls to fetch data to populate a template.
- doCheck, afterContentChecked and afterViewChecked are called twice at application bootstrap time because the change detection process runs twice at bootstrap due to the app being served by the development server. In production mode, it will only run once.
- The ngOnChanges hook is not called as this hook is only relevant for child components that received @Input property values via property binding in the template of their parents.
- The ngOnDestroy hook is not called because this component is forever in the DOM throughout the life time for the application and is never removed.
- Typing into the first text box does not cause any hooks to be fired because there is no event binding created for the first text field.
- For the second text box, the event binding causes change detection to run and the 3 hooks: doCheck, afterContentChecked and afterViewChecked are then called.

References:

https://angular.io/guide/lifecycle-hooks

https://www.tektutorialshub.com/angular/angular-component-life-cycle-hooks/#angular-lifecycle-hook-example

4 Life cycle hooks for a component and its child

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

```
cd lifecycledemo
```

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

```
ng generate component FirstChild
```

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

`app.component-v2.ts`

`app.component-v2.html`

`first-child.component-v2.ts`

`first-child.component-v2.html`

Key points to note:

a) The constructors of the parent component and child component are always called first before any of the lifecycle hooks.
b) When the application is being bootstrapped, the various hooks of the child component are called after afterContentChecked in the parent component. The final concluding hooks of afterViewInit and afterViewChecked are called in the parent component after all hooks in the children in the tree are completed.
c) ngOnChanges is fired in the child component because it has a @Input property from which it receives values from its parent via property binding in the parent template. It is fired whenever there is any change to this property: which will be the case when the app is bootstrapped for the first time, and subsequently when there are changes in any of the component properties involved in the binding.
d) doCheck, afterContentChecked and afterViewChecked is called twice at bootstrap for both the parent and child component as explained earlier.
e) When any event occurs in either the parent or child component, the change detection always runs from the top of the component tree. Therefore, the doCheck in the parent component will always run first at the start of the change detection cycle.
f) ngOnDestroy is called for the child component when it is destroyed (i.e. when it is removed from the DOM due to the ngIf directive condition becoming false). ngOnDestroy is never called for the parent component because it is never removed from the DOM throughout the life time of the application.
g) When we add the child component back in the DOM again (i.e. by making the ngIf directive condition true), then the complete sequence of calls for a new instance of a component is performed (i.e. constructor, ngOnChanges, ngOnInit, etc)

References:

https://angular.io/guide/lifecycle-hooks

https://www.tektutorialshub.com/angular/angular-component-life-cycle-hooks/#angular-lifecycle-hook-example