Lab 1 - Introduction to Angular

In this lab we will build a Hello World style Angular component. The key focus is to learn how to install all the required code and use them from the browser. We wont get too deep into Angular at this point.

Part 1 - Create a Project

Angular needs a lot of boiler plate code even for the simplest of projects. To save time a starter project is given to you. Throughout these labs we will create our projects from this starter project. This "project-seed" is based on the Angular Quickstart project you can download from the Angular website.

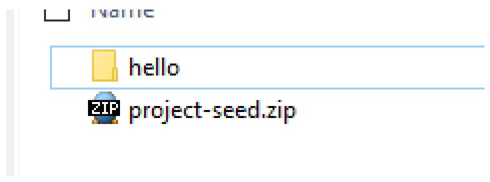
__1. Create a folder '**C:\LabWork**' on your machine. This will be used for various labs.

__2. Copy **C:\LabFiles\project-seed.zip** to the '**C:\LabWork**' folder.

__3. Extract the ZIP file within the folder. This should create a 'project-seed' subfolder. Make sure the zip software doesn't create an extra layer of subfolder as the zip will do this already.

__4. Rename the extracted folder to **hello**.

__5. The file system should look like this at this point.



We are not quite ready to run this project. Before we do that let's get to know more about the boiler plate code given to you.

Note: The 'project-seed' was created from the Angular Quickstart project

https://github.com/angular/quickstart

Part 2 - The Anatomy of a Simple Angular Project

Angular framework is released as several separate modules. For example, common, core, http, router etc. In addition, Angular depends on third party projects such as rxjs and zone.js. Manually downloading these modules can be cumbersome and error prone. This is why we use Node Package Manager (NPM) to download them.

You can use your favorite editor to edit code during these labs. For your convenience we have included the Notepad++ editor located under **C:\Software\NotepadPlus**.

__1. Open **package.json** from the **hello** folder in an editor.

This file declares the packages we have dependency on. Specifically the **dependencies** property lists packages that we need at compile and runtime. The **devDependencies** section lists packages that we need for certain development tasks. You will see **typescript** listed there. We need this to be able to compile our TS code into JS.

To save time and to take into account that your lab environment may not have Internet access we have already installed all the packages.

__2. Open a command prompt window and go to the **C:\LabWork\hello** directory.

```
cd C:\LabWork\hello
```

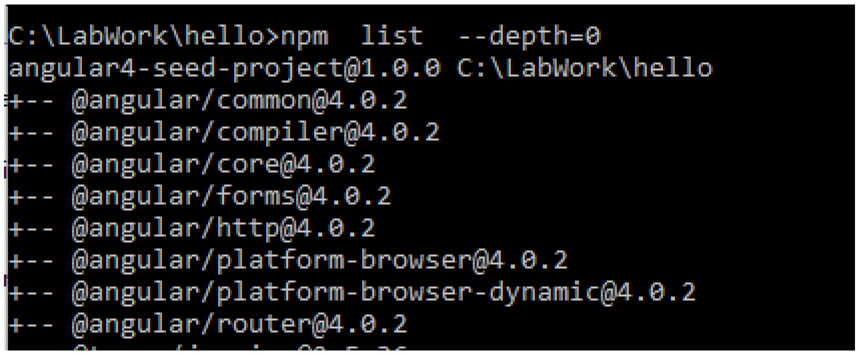
Although not required, you can run this command to install all the packages if you have access to the Internet.

```
npm install
```

Note: The starter project project-seed.zip was created in Windows. If you are running this lab in any other OS you must re-install all Node modules.

__3. Run this command to list what have been installed.

```
npm  list  --depth=0
```



```
C:\LabWork\hello>npm  list  --depth=0
angular4-seed-project@1.0.0 C:\LabWork\hello
+-- @angular/common@4.0.2
+-- @angular/compiler@4.0.2
+-- @angular/core@4.0.2
+-- @angular/forms@4.0.2
+-- @angular/http@4.0.2
+-- @angular/platform-browser@4.0.2
+-- @angular/platform-browser-dynamic@4.0.2
+-- @angular/router@4.0.2
```

__4. These packages are physically located inside the **node_modules** folder. Have a quick look at it.

Note: You may see some 'extraneous' packages but this is OK since the project-seed project also has some extra software needed for later labs. These are 'extraneous' because they are not listed as dependencies right now and do not need to be for basic Angular projects.

Next, we will look at the simple source code of the template project. We have not covered much of TypeScript or Angular yet. So do not worry too much about the details. There will be plenty of more labs to learn the details.

__5. Open **hello/src/app/app.component.ts** in an editor.

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `<h1>Hello {{name}}</h1>`,
})
export class AppComponent  { name = 'Angular'; }
```

The **import** statement is loading the @angular/core module and bringing the "Component" name into the current namespace. Which means we can now start using the "Component" name for its intended purpose, which may be anything from a class name to a variable name. In this particular case "Component" is a decorator. By using the decorator with the AppComponent class we are saying that AppComponent is an Angular component.

We are also adding a few meta data elements to the decorator. The **selector** property declares the HTML tag for the component. The **template** property states the HTML content of the component.

__6. Next open **hello/src/app/app.module.ts**.

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent }  from './app.component';

@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

One of the main points of this file is to import the 'AppComponent' declared in the other file into the 'Module' declared by this file. This file also declares the class 'AppModule' which indicates what component declarations are part of the module for the application. Note the “bootstrap” setting that also points to the 'AppComponent'.

__7. Next open **hello/src/main.ts**.

```
import { platformBrowserDynamic } from '@angular/platform-browser-
dynamic';

import { AppModule } from './app/app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

The main goal of this file is to “bootstrap” the 'AppModule' class declared in the file you just saw. This is a function that binds a component to the DOM. Essentially, this will instantiate the component wherever the "my-app" tag appears in the DOM.

__8. Finally, open **hello/src/index.html**.

Notice that despite having dependency on so many modules we have only a handful of <script> tags. Not even Angular is showing up anywhere in the <script> tags. This is because the majority of the modules are loaded asynchronously and only as needed by a module loader. Within our TS code we use **import** to load a module. From HTML we need to use **System.import** API call. This is a standard API. However, none of the browsers today support it. SystemJS is an open source JavaScript module loader that provides a shim (drop in replacement) for it. We will use SystemJS in these labs to load modules.

Observe this code in index.html:

```
<script>
  System.import('main.js').catch(function(err){ console.error(err); });
</script>
```

This will load the "app" module as described in the 'main.js' file you just saw.

__9. Look at this HTML.

```
  <body>
    <my-app>Loading AppComponent content here ...</my-app>
  </body>
```

The <my-app> tag will be replaced by the HTML content of our component.

__10. Open **hello\src\systemjs.config.js** in an editor.

__11. Take a look at the 'map' information which tells SystemJS that our 'app' module files are in the 'app' source folder and where to find the Angular and other libraries that will be needed.

```
// map tells the System loader where to look for things
map: {
  // our app is within the app folder
  'app': 'app',

  // angular bundles
  '@angular/core': 'npm:@angular/core/bundles/core.umd.js',
  '@angular/common': 'npm:@angular/common/bundles/common.umd.js',
  '@angular/compiler': 'npm:@angular/compiler/bundles/compiler.umd.js',
```

Note: In prior versions of the Angular Quickstart project a more indirect way of importing the application was used. Previously the System.import would import 'app'

```
System.import('app')
```

But where was the "app" package defined? Here "app" is just a symbolic name for the hello/src/main.ts. This mapping was done in the systemjs.config.js file.

```
    packages: {
      app: {
        main: 'main.js',
        defaultExtension: 'js'
      },
```

This tells SystemJS that when we try to load the "app" package by calling System.import("app") it will need to make a HTTP request for "main.js".

The updated version of the Quickstart project still defines the 'app' package in the systemjs.config.js file but does not use the indirection to import in the HTML.

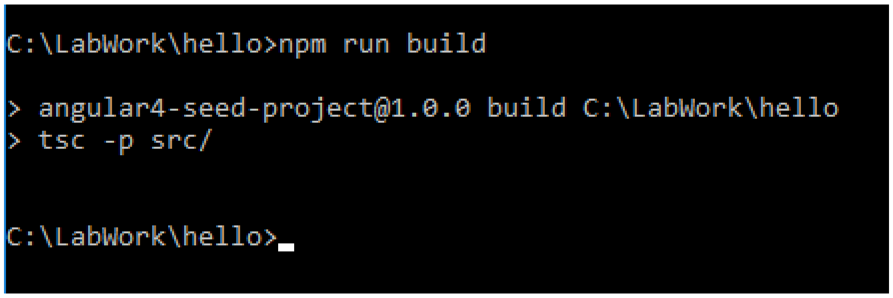
Part 3 - Compile TypeScript Code

TypeScript code needs to be compiled into JavaScript before we can run it in a browser.

__1. From the command prompt go to the root folder of our project **C:\LabWork\hello**.

__2. Run this command to compile TypeScript.

```
npm  run  build
```



```
C:\LabWork\hello>npm run build

> angular4-seed-project@1.0.0 build C:\LabWork\hello
> tsc -p src/


C:\LabWork\hello>_
```

Note. During the Labs you will use this command prompt window to compile the projects, leave it open all the time. If you close it simply open a new command prompt window and change to the project folder.

Our package.json has set up 'build' as a script task. This allowed us to run the compiler using the "npm run" command.

Alternatively, you can install typescript globally:

npm install typescript -g

After that you can simply enter this command in the project's root folder to compile TypeScript:

tsc -p src/

Part 4 - Test

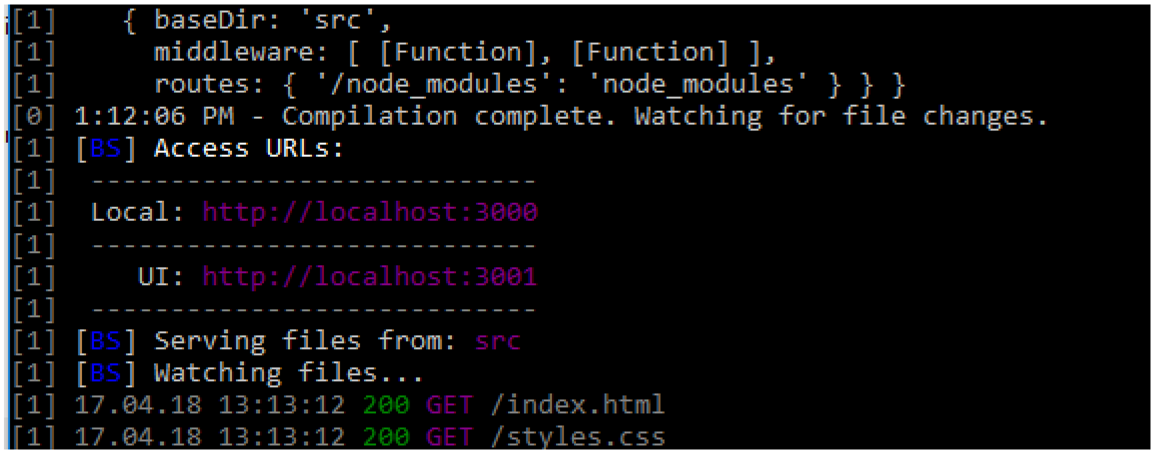
At this point the project has compiled JavaScript and would be ready to run and test. You could copy the project files to a web server and get them from there. The Angular quickstart project setup though has an embedded "lite-server" that can make it easier to run an Angular project to test. This section will use that feature.

__1. Make sure you are still in a command prompt in the root folder of our project **C:\LabWork\hello**. Open a new command prompt and go to that location if needed.

__2. Run the following command. The first time this command may take several seconds to run.

```
npm run start
```

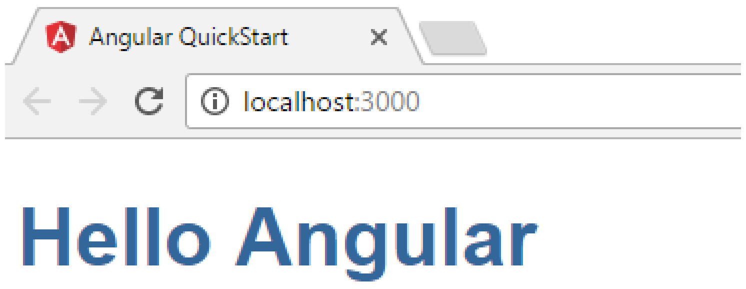
__3. After several seconds you should eventually see that the code has been compiled (we didn't really need to do it separately before) and a local server is running at the URL 'http://localhost:3000'.



```
[1]    { baseDir: 'src',
[1]      middleware: [ [Function], [Function] ],
[1]      routes: { '/node_modules': 'node_modules' } } }
[0] 1:12:06 PM - Compilation complete. Watching for file changes.
[1] [BS] Access URLs:
[1]  ----------------------------
[1]  Local: http://localhost:3000
[1]  ----------------------------
[1]     UI: http://localhost:3001
[1]  ----------------------------
[1] [BS] Serving files from: src
[1] [BS] Watching files...
[1] 17.04.18 13:13:12 200 GET /index.html
[1] 17.04.18 13:13:12 200 GET /styles.css
```

__4. The command should also open a web browser to this location automatically. If it does not for some reason, open a browser and go to:

```
http://localhost:3000
```



Hello Angular

__5. Leave the browser and the command prompt running for the next section.

Part 5 - Change the Component Code

We will make a small change. You will see how the 'start' command will automatically recompile the code.

__1. Open **hello/src/app/app.component.ts** in an editor.

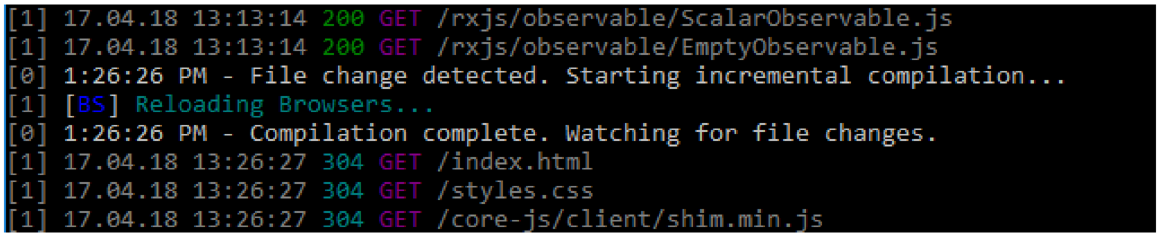
__2. Make the change shown in bold below.

```
@Component({
  selector: 'my-app',
  template: `<h1>Hello {{name}}</h1>`,
})
export class AppComponent  { name = 'partner!'; }
```

The component has some state in the form of a field called 'name'. It is rendering the value of this "name" variable in HTML using the {{name}} syntax. This is an example of one way data binding where data is taken from the component and shown in DOM.

__3. Save the file.

__4. Back in the command prompt you should see lots of messages about the code being recompiled and the browser being refreshed. There are a lot of messages about resources being requested again but nearly all of them will have a '304' status because they have not been modified.

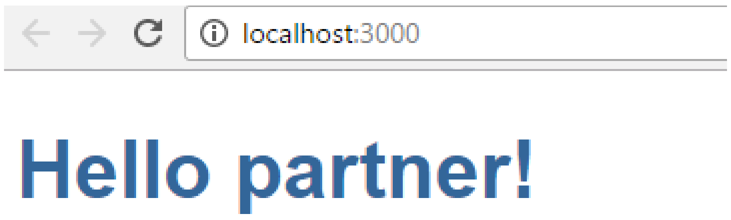


```
[1] 17.04.18 13:13:14 200 GET /rxjs/observable/ScalarObservable.js
[1] 17.04.18 13:13:14 200 GET /rxjs/observable/EmptyObservable.js
[0] 1:26:26 PM - File change detected. Starting incremental compilation...
[1] [BS] Reloading Browsers...
[0] 1:26:26 PM - Compilation complete. Watching for file changes.
[1] 17.04.18 13:26:27 304 GET /index.html
[1] 17.04.18 13:26:27 304 GET /styles.css
[1] 17.04.18 13:26:27 304 GET /core-js/client/shim.min.js
```

Note: The code automatically recompiled because the 'start' command runs a 'build:watch' command that will look for the source of the application to change and automatically recompile if it does. If you want you can run this separately as:

```
npm run build:watch
```

__5. Go to the browser window and verify the new message appears.



Hello partner!

__6. In the command prompt, hit '**<CTRL>-C**' and verify you want to terminate the batch job.

__7. Close all open files.

Part 6 - Review

Our goal in this lab was to take a look at a basic Angular application and go through the workflow of compiling TypeScript.

First we learned that our code is written in TypeScript which needs to be transpiled into JavaScript.

Next, we briefly looked at how module loading works with SystemJS. Within the code we load modules using the **import** statement. At runtime SystemJS loads these modules asynchronously using Ajax. There is no need to add these JavaScript files to the HTML using <script> tags.