Learn Angular in 2 days Step by Step

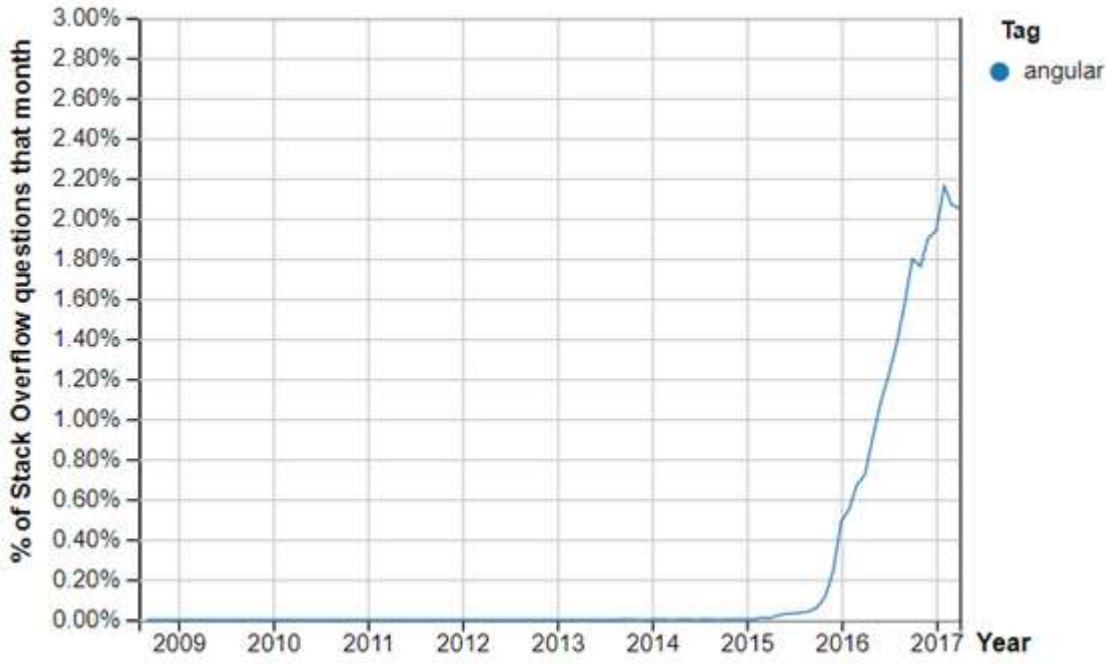
(Covering Angular 2 and Angular 4/6 Labwise)

For online Angular training or self-study materials visit: www.questpond.com
For Face to Face classroom/offline Angular training in Mumbai: www.stepbystepschools.net

Table of Contents

Introduction

Why should we learn Angular? , the below stack overflow graph says it all. Its popular, it is hot with lot of job openings. This book teaches you Angular step by step via 20 great Labs. So if you are here to learn Angular then you are at the right place and with the right book.



AngularJS vs Angular. There are two versions of Angular the old is named as AngularJS and the new one is named as just Angular. So when someone says AngularJS it means Angular 1.X and when someone says JUST Angular its Angular 2/4.

Who wrote this book?

My name is Shivprasad Koirala, a 43 years old selfish developer and author 😊. He steals times from his kids and family to write books 😊. You can catch his Learn Angular in 8 hours step by step video series from https://www.youtube.com/watch?v=oMgvi-AV-eY

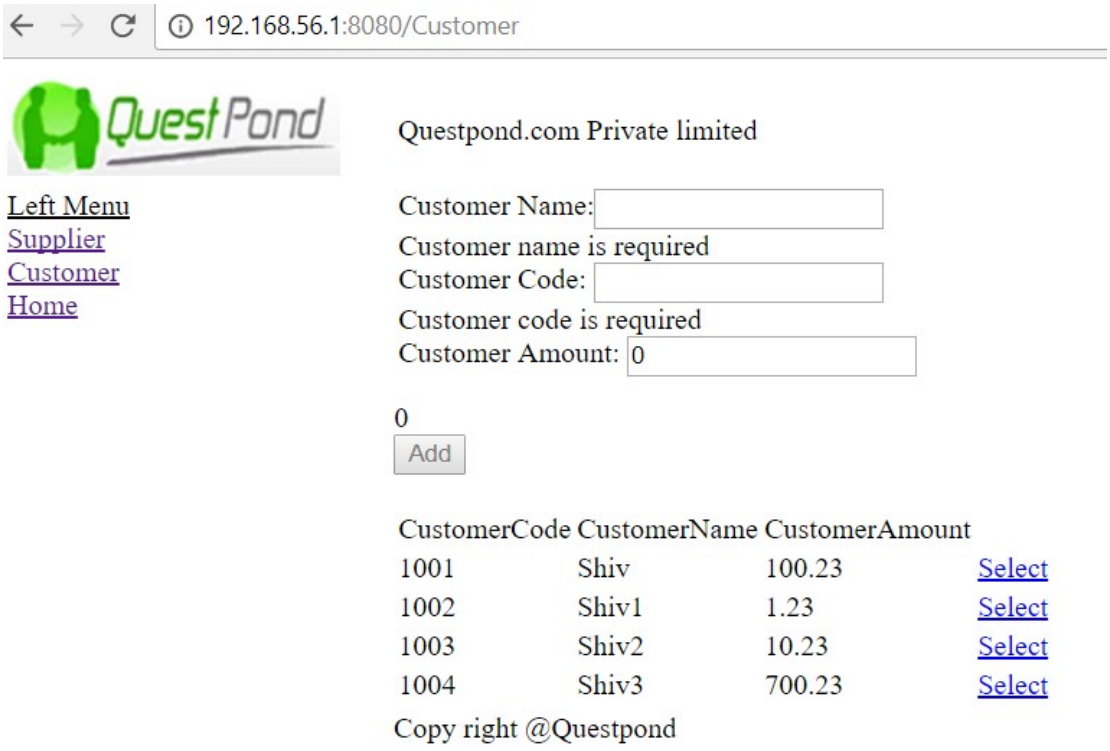
He feeds himself by recording training videos on www.questpond.com and also has an institute in Andheri, Mumbai where he takes training on Angular in weekly basis. For any technical/non-technical issues with this book please feel to contact him on questpond@questpond.com
Please do boost my EGO 😊 by tagging me on my Facebook profile at https://www.facebook.com/shivprasad.koirala , it just motivates us to write more.

How does this book teach you Angular?

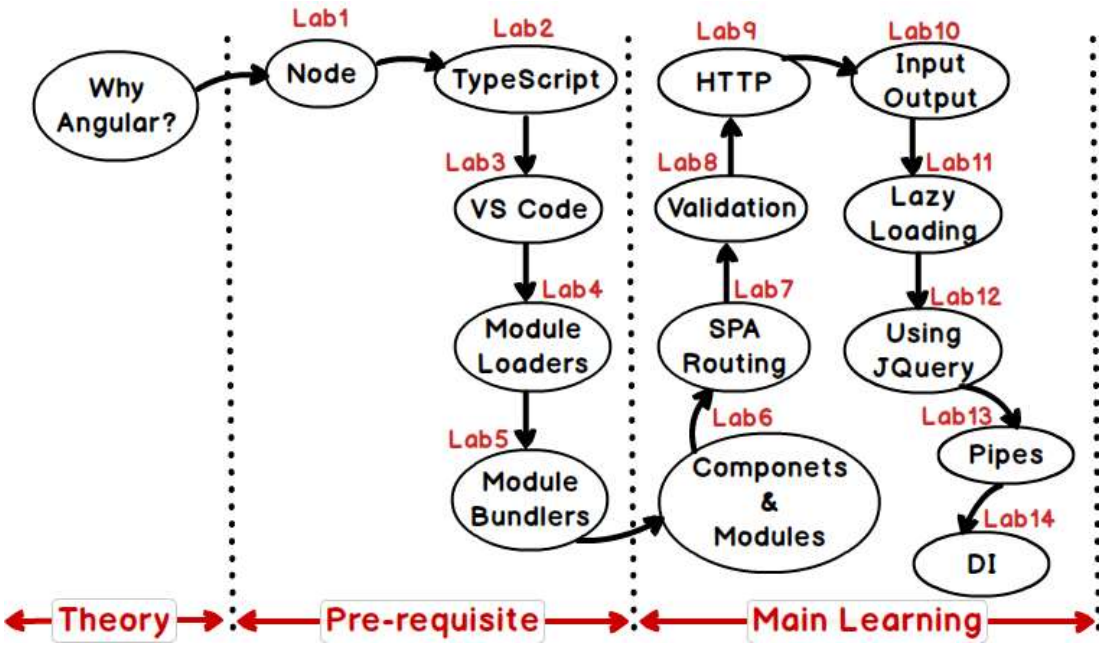
The best way to learn Angular or any new technology is by creating a project. So in this step by step series we will be creating a simple Customer data entry screen project.

This project will have the following features:-

- Application should have capability of accepting three fields Customer name, Customer and Customer Amount values.
- Customer name and Customer codes are compulsory fields and it should be validated.
- Application will have an “Add” button, which help us to post the current customer data to a Server. Once the data is added to the server it should displayed on the grid.
- Application will have a navigation structure wherein we will have logo and company name at the top, navigational link at the left and copy right details at the bottom of the screen.



So below is the road map of this book. It has three phases covering 14 labs step by step: -



Theory Phase: - In this phase we will understand what Angular is? And why do we need it.

Pre-requisite phase: -In this phase we will four important things Node, TypeScript, VS Code, Module loaders (SystemJS) and module bundlers (Webpack).

Main learning phase: - This is where actual angular starts. In this we will be having 10 labs and while covering those labs we will create the customer data entry screen project as discussed previously.

So do not wait any more start LAB by LAB and STEP by STEP.

Should I start from Angular 1, 2 or 4. Angular 1.X and 2.X are very much different. So even if you have done Angular 1.X you have to restart fresh from Angular 2.X. Angular 2.X and Angular 4.X are backward compatible so if you are learning Angular 2 you are learning Angular 4 and ahead. So people who are new to Angular just start from Angular 4 and **this book teaches Angular 4**.

Why do we need Angular?

*"Angular is an open source JavaScript framework which simplifies **binding code** between JavaScript objects and HTML UI elements."*

Let us try to understand the above definition with simple sample code.

Below is a simple "Customer" function with "CustomerName" property. We have also created an object called as "Cust" which is of "Customer" class type.

```
function Customer()
{
      this.CustomerName = "AngularInterview";
}
var Cust = new Customer();
```

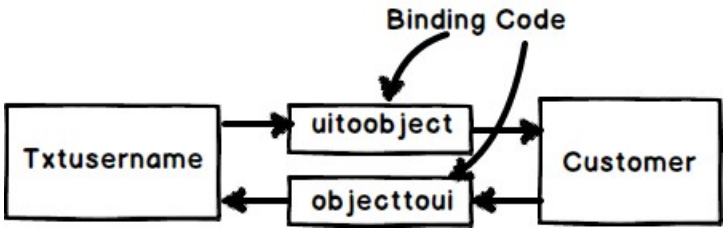
Now let us say in the above customer object we want to bind to a HTML text box called as "TxtCustomerName". In other words when we change something in the HTML text box the customer object should get updated and when something is changed internally in the customer object the UI should get updated.

```
<input type=text id="TxtCustomerName" onchange="UitoObject()"/>
```

So in order to achieve this communication between UI to object developers end up writing functions as shown below. "UitoObject" function takes data from UI and sets it to the object while the other function "ObjecttoUi" takes data from the object and sets it to UI.

```
function UitoObject()
{
      Cust.CustomerName = $("#TxtCustomerName").val();
}
function ObjecttoUi()
{
$("#TxtCustomerName").val(Cust.CustomerName);
}
```

So if we analyze the above code visually it looks something as shown below. Your both functions are nothing but binding code logic which transfers data from UI to object and vice versa.

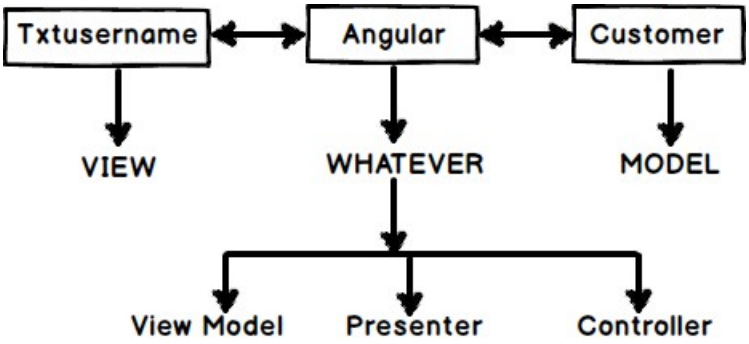


Binding Code

Now the same above code can be written in Angular as shown below. So now whatever you type in the textbox updates the "Customer" object and when the "Customer" object gets updated it also updates the UI.

```
<input type=text [(ngModel)]="Customer.CustomerName"/>
```

In short if you now analyze the above code visually you end up with something as shown in the below figure. You have the VIEW which is in HTML, your MODEL objects which are JavaScript functions and the binding code in Angular.



Now that binding code have different vocabularies.

- Some developers called it "ViewModel" because it connects the "Model" and the "View".
- Some call it "Presenter" because this logic is nothing but presentation logic.
- Some term it has "Controller" because it controls how the view and the model will communicate.

To avoid this vocabulary confusion Angular team has termed this code as "Whatever". It's that "Whatever" code which binds the UI and the Model. That's why you will hear lot of developers saying Angular implements "MVW" architecture. So concluding the whole goal of Angular is Binding, Binding and Binding.

Lab 1:- Practicing NodeJS

So the first JavaScript open source which you should know before learning Angular is NodeJS. In this lab whatever I am adding is also demonstrated in this YouTube practical Angular video as well, here is the link https://www.youtube.com/watch?v=-LF_43_Mqnw, so feel free to see demonstrative lab.

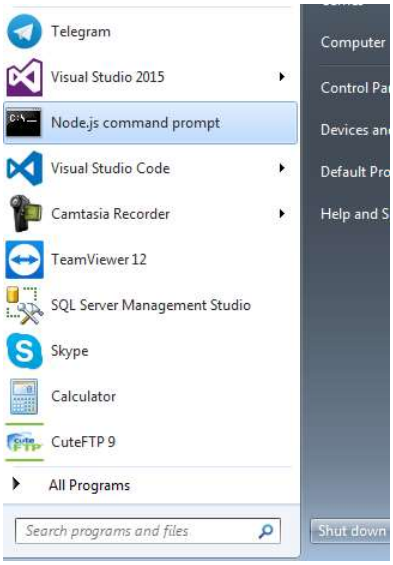
Theory: - What is NodeJS?

NodeJS is an open source JavaScript framework which does two things: -

- It helps you to run JavaScript outside the browser. NodeJS uses the chrome JavaScript engine to execute JavaScript outside the browser so that we can create desktop and server based application using JavaScript.
- It also acts a central repository from where we can get any JavaScript framework using NPM (Node package manager).

Learn but do not over learn. NodeJS is a big topic by itself. For Angular we just need to know how to use NPM commands. So we will be limiting ourselves only around how to use NPM. We will not be doing full-fledged node programming. Remember JavaScript is vast do not do unnecessary learning or else your focus will drift.

Step 1:- Installing NodeJS



In order to install NodeJS, go to https://nodejs.org/ & download the latest version and install it.

Once you install node you should see NodeJS command prompt in your program files as shown in the figure.

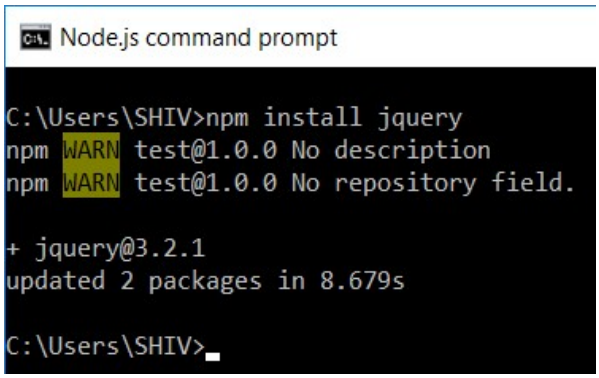
We can then open the NodeJS command prompt and fire NPM commands inside this command prompt.

In case you are completely new to NodeJS please see this NodeJS Video which explains NodeJS in more details.

Step 2: - Practicing NPM Install command

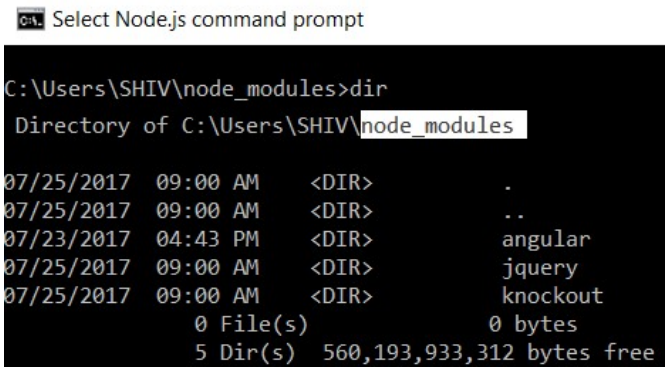
So let's practice the first command in NPM "npm install". "npm install" command helps you get the latest version of any JavaScript open source framework.

For example if you want to install jQuery you will open node command prompt and type "npm install jQuery" and once you press enter you should see "jQuery" has been installed in your computer.



Are you wondering where jQuery has been installed. It has been installed in the same folder where you ran the NPM command.

In that folder he has created a "node_modules" folder and in that he has created "jquery" folder where all jQuery had been loaded by "npm".



Step 3:- Understanding package.json file

When you work with large projects you would need lot of JavaScript frameworks. So in one project you would probably need jQuery, angular, lodash and so on. Doing "npm install" again and again is definitely wasting precious time of your life.

So to load all JavaScript framework references in one go "npm" team has given a package.json. In this package.json file you can make an entry to all JavaScript references and load those in one go.

To create package.json file open the node command prompt and type "npm init". After "npm init" command it would ask for package name, version number, project description and so on. Once you fill the entire question it will create a package.json file in your current folder. Below is how "npm init" command looks like.

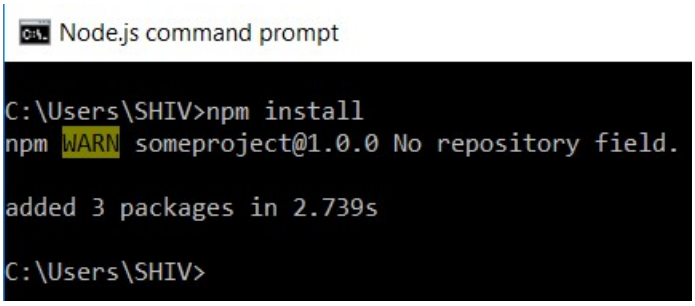
```
C:\Users\SHIV>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.
package name: (test) someproject
version: (1.0.0) 1.0.0
description: This is a simple project
git repository:
keywords:
author:
```

Once npm init command has been successfully executed it creates a “package.json” file in the current folder. If you open “package.json” file it has the following below structure.

Do not overload yourself with the entire information just zoom on the “dependencies” node. This node has all JavaScript dependencies listed out with version number. So in our package.json file we have all the dependencies listed down.

```
{
  "name": "test",
  "version": "1.0.0",
  "description": "",
  "main": "MyClass.js",
  "dependencies": {
    "angular": "^1.6.5",
    "jquery": "^3.2.1",
    "knockout": "^3.4.2",
    "lodash": "~4.17.4"
  },
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Wherever “package.json” file is created you just need to type “npm install” command as shown in the figure.



If you remember in package.json file we had 3 JavaScript framework dependencies listed those will be installed one after another. You can see in the image it is stating “added 3 packages”.

Learn but do not over learn. Package.json has lot of configuration do not spend your stamina in learning all those. As we do the labs ahead I will be walking through important ones. So keep moving ahead with the chapters.

Understanding versioning system in package.json

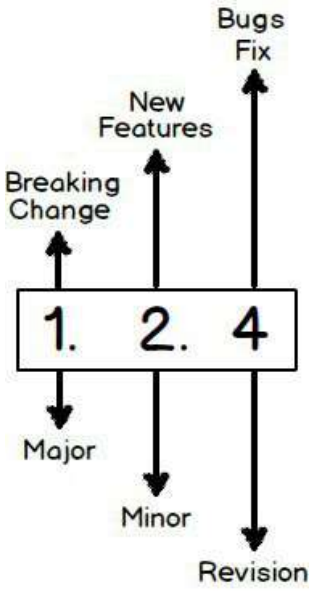
Most software versions follow semantic versioning. In semantic versioning versions are divided in to three numbers as shown in the image below.

The first number is termed as “major version”, second “minor version” and third “revision”.

Major version: - Any increment in major version is an indication that there are breaking changes in the software functionality. It’s very much possible that the old code will not work with these changes and have to be tested properly.

Minor version: - This version is incremented when we add new features but the old code still works.

Revision: - This version is incremented when we are just doing bug fixes. So there are no new functionalities added, no breaking changes and back ward compatible with old code.



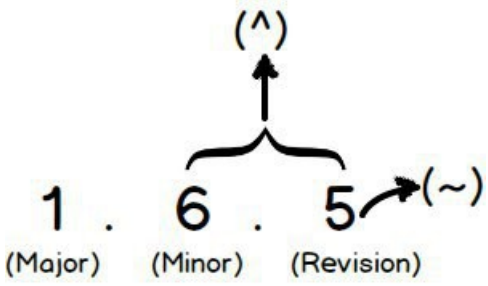
NPM follows semantic versioning but it also has some more special characters like “^”, “~”, “>” and so on. They dictate how NPM get latest should behave for Major and Minor versions.

```
"angular": "^1.6.5",
"angular": "1.6.5",
"angular": "~1.6.5",
```

For these formats 3 formats are very primary let’s understand each them.

Exact (1.6.5), Major/Minor (^1.6.5) or Minor(~1.6.5).

Exact (1.6.5): - This will do a get latest of exact version 1.6.5 not more or not less. If that version is not available it will throw up an exception.



Major/Minor(^1.6.5): - The carrot sign will get minimum 1.6.5 and if there are any higher MINOR / REVISION versions it will get that. It WILL NEVER GET HIGHER MAJOR VERSIONS. So if 1.6.5 has 1.6.7 it will get that, if it has 1.7.7 it will that, but if it as 2.0 it will NOT get that.

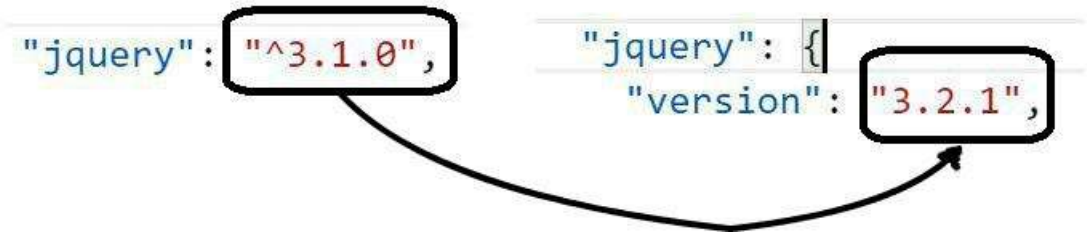
Minimum or lower (~1.6.5): - The tilde sign will get HIGHER REVISIONS. For if 1.6.5 has 1.6.7 it will get that, but if it has 1.7.5 it will not be installed, if it has 2.0 it will not be installed.

What is package-lock.json file?

As discussed in the previous sections package.json has “^” and “~” versioning mechanism. Now suppose in your package.json you have mentioned "jQuery": "^3.1.0"and jQuery has a new version “3.2.1”. So in actual it will install or in other words LOCK DOWN to “3.2.1”.

So in package.json you will have “^3.1.0” but actually you will be using “3.2.1”. This entry of actual version is present in “package-lock.json”. So package lock files have the EXACT versions which are used in your code.

Below is the image snapshot of both the files.



Important NPM commands

NPM has huge command list. Down below are some very important command which you will need now and then. As said previously LEARN but don't do over LEARNING.

Command	Explanation
npm install -g typescript	The -g command will install the package globally. In other words in your computer when you use this package it will take the global installation. This should be used only for command lines packages like grunt, TypeScript, npm and so on.
npm install -save jQuery	This will install the package and also make an entry in to the package.json file. Some time we are use a package in our project and forget to update package.json this comes very handy.
npm view -version jQuery npm view -versions jQuery	This first command will show you latest jQuery version on GitHub and the second one will show all versions in an ascending manner.
npm install -g npm	This command updates npm himself. "-g" as discussed in previously helps to install npm globally.

Lab 2:- Practicing TypeScript

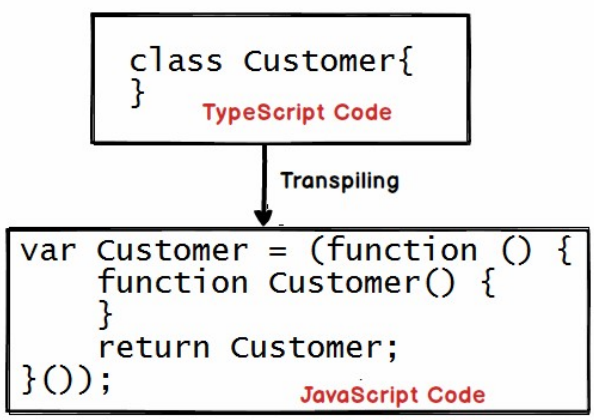
Angular is created using TypeScript language. So if you are doing development with Angular TypeScript is the way to go ahead.

Introduction: - Why do we need TypeScript?

Now JavaScript is a great and WEIRD language. So in JavaScript if you want to do inheritance you need to use prototype, it's not a strongly typed language, there is no polymorphism and so on. So when developers who come from C# and Java background it's very difficult for them to get

acquainted with this weird language. People who come from C# and Java background use OOP features a lot. So to fill this GAP answer is “TypeScript”.

“TypeScript is a sugar-coated Object-oriented programming language over JavaScript.”



So in typescript we can write code using keywords like “class”, “extends”, “interface” and so on.

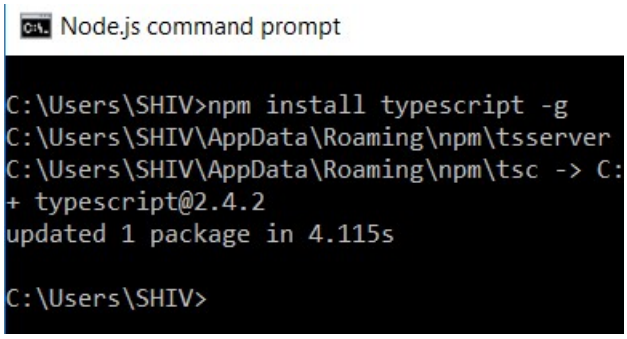
Internally TypeScript will compile (must be right word would be “transpile”) in to pure JavaScript code in terms of functions, closures and IIFE.

Please do watch 1 hour Training video on TypeScript which explains Typescript in more detail.

Step 1: - Installing TypeScript

So to install TypeScript we need to use “npm”. TypeScript is a JavaScript open source framework so the best way to get it installed is by using “npm”. So open node command prompt and type “npm install typescript -g”.

The “-g” command says that you can execute typescript command from any folder.

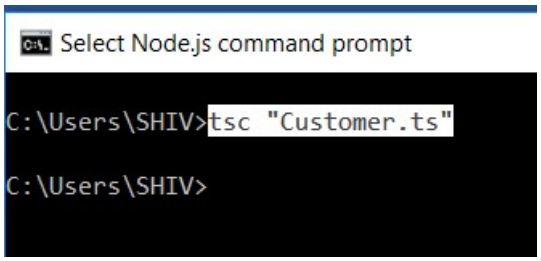


Step 2: - Compiling a simple TypeScript to JavaScript

Let’s try to understand how we can compile a TypeScript to JavaScript. For that let us create a simple “Customer.ts” file with the following code.

```
class Customer{
}
```

Now open NodeJS command prompt and type command 'tsc "Customer.ts"'. Once you press enter it will create "Customer.js" in the same folder.



If you remember "tsc" was registered globally during "npm install" command. So this "tsc" command can be executed in any folder.

Below is the JavaScript output from typescript command line utility.

```
var Customer = (function () {
    function Customer() {
    }
    return Customer;
}());
```

Many people term this conversion from typescript to JavaScript as "compiling". Personally, I feel we should call this process as "transpiling".

Compiling converts from a higher level languages like C# , Java , C++ to machine language or some intermediate language which cannot be read by humans. While transpiling converts from one higher level language to another higher-level language. In this both TypeScript and JavaScript are higher level language. So let's term this process and **transpiling** and let us call typescript as a "**transpiler**" rather than a compiler.

Step 3:- Using tsconfig.json file

The transpiling process of typescript has lot of advance settings. Below are some options you can pass to tsc command line while compiling: -

Options	**Description**
tsc Customer.ts –removecomments	While transpiling the comments will be removed
tsc Customer.ts --target ES5	This will compile using ES5 specifications.
tsc Customer.ts --outdir "c:\users\shiv"	This will compile to a specific output directory
tsc foo.ts bar.ts –outFile "Single.js"	This will compile multiple TS files to single JS file.
tsc Customer.ts -w	This command will run typescript continuously in the background and keep looking for changed files. If the file it will compile that file.

But now let's think practically, if I want transpile with ES5 specification, to a specific directory without comments the command line would become something as shown below.

```
tsc Customer.ts --outdir "c:\users\shiv" --target ES5 --removecomments
```

That's where tsconfig.json file comes to rescue. You can put all these configurations in "tsconfig.json" file and then just execute "tsc".

```
{
  "compilerOptions": {
    "target": "es5",
    "removeComments": false,
    "outDir": "/Shiv"
  }
}
```

Learn but do not over learn. Tsconfig.json has 1000's of properties do not spend your stamina in understanding all of them now. Move ahead with the labs when any new typescript config comes up we will look in to it.

Lab 3:- Practicing VS Code

What is VS Code?

Theoretically you can do Angular with a simple notepad. But then that would be going back to back ages of Adam and Eve and reinventing the wheel. So we will need some kind of tools by which will help us to type HTML easily, compile typescript and so on.

That's where VS code is needed. VS code is a free editor provided by Microsoft which will help us with all automation for HTML, JavaScript, and Typescript and so on.

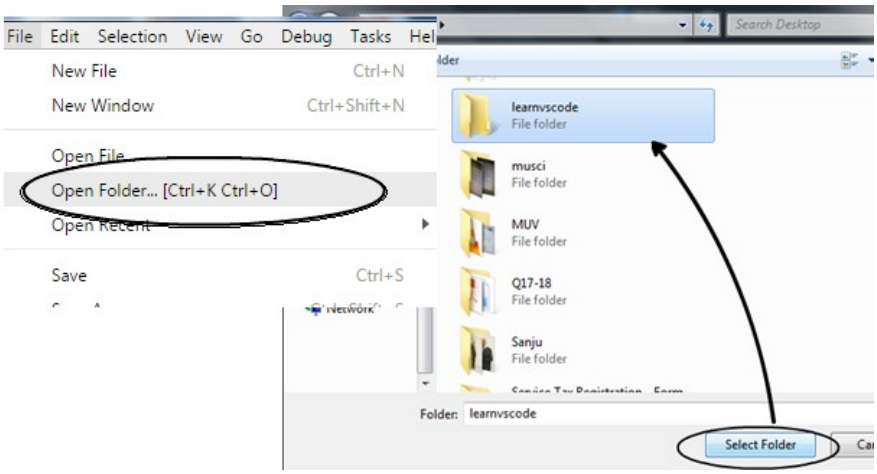
So go to https://code.visualstudio.com/download and depending on your operating system install the appropriate one. For instance I am having windows OS so I will be installing the windows version.

Once you download the setup it's a simple setup EXE run it and just hit next, next and finish.

You can also watch this VS code tutorial which will help you to understand
https://www.youtube.com/watch?v=gQ9CiRlRPKs

Point number 1: - All actions happens in a folder

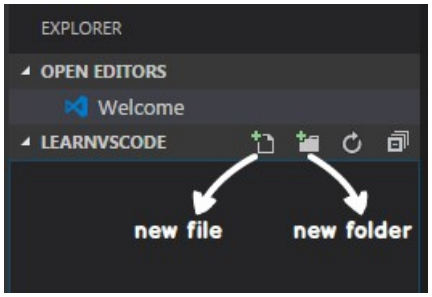
In VS code all source code you put inside a folder. So the first step is to create a folder and point VS code to that folder by clicking on File → Open and select folder shown in the below figure.



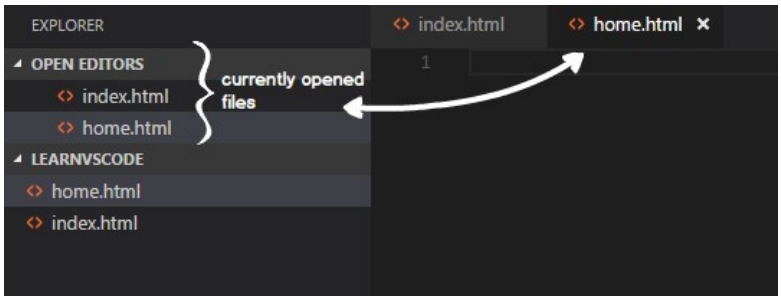
Point number 2: - Creating files and folders

If you want to create a file or sub folder you can click on the icons as shown in the figure.

The first icon creates a file and the second icon creates a folder.

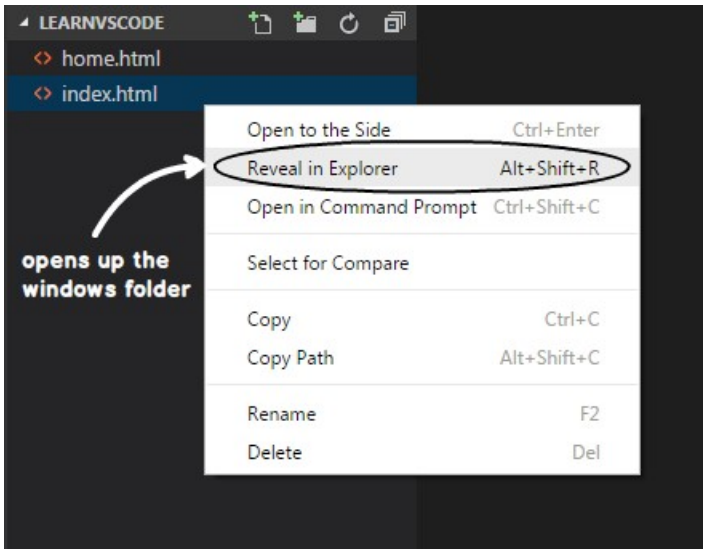


Point number 3: - Explorer and Open Editors



The explorer part of VS code has two section one which shows open editors and the other which shows your folder. You can see the image where open editors are shown. You can click on those cross signs to close the open files.

Point number 4:- Reveal in explorer

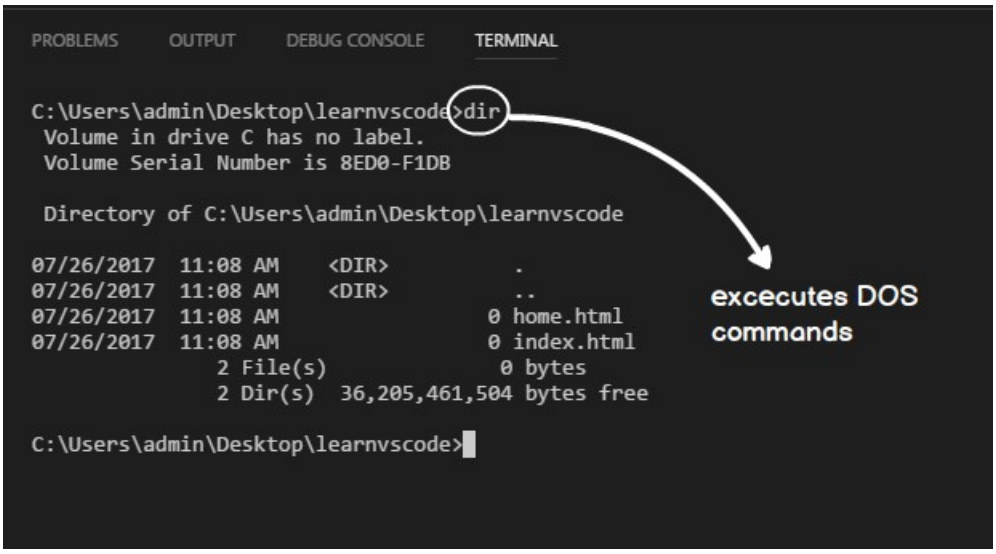


If you want to browse to the current folder, do can right click on the folder and click on “Reveal in Explorer”.

Point number 5: - Integrated Terminal

TypeScript, Node these frameworks mostly run through command prompts. So it would be great if we can have integrated command line inside VS code. VS code has something called as Integrated Terminal, you can open the integrated terminal by clicking on view → Integrated Terminal.

Once you are inside the integrated terminal you can fire “npm install”, “tsc” and so on. Below is how the integrated terminal looks like.



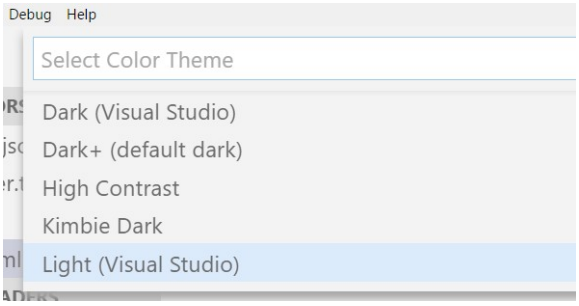
Point number 6: - Running multiple terminals

One of the things we always need is running multiple commands and for that we need a facility to load multiple terminals.

In VS code we can load multiple terminals by clicking on the plus sign as shown below. So in one terminal you can run the webserver and in the other terminal you can a code review tool.

Point number 7: - Changing to soothing color themes

By default, VS code shows black theme which is very good for health of your eyes. But sometimes to just have more code clarity you would like to change to some brighter theme. You can do that by clicking on file → preferences → color theme, you would get themes as shown in the below figure.



Point number 8: - VS code settings

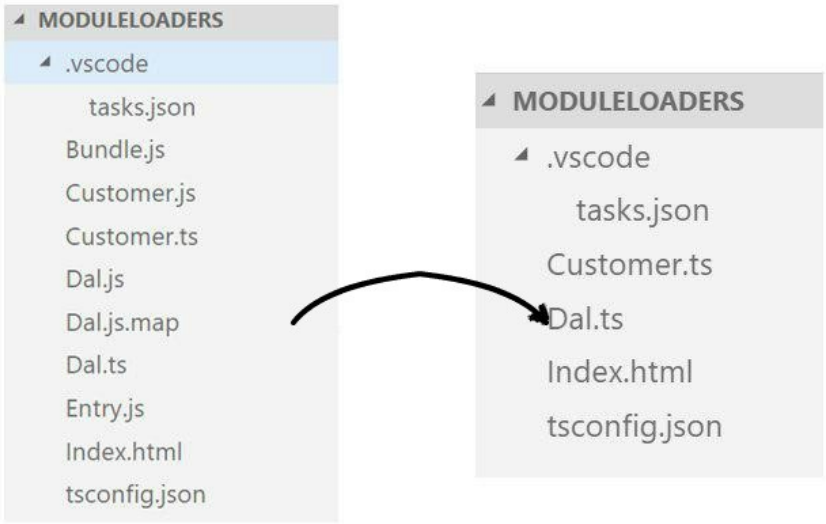
VS code has lot of settings like you can hide unwanted files and only focus on the files you want, change icons settings, change font size and so on. To apply a setting, you need to goto File → Preferences → Settings and you would be shown a figure as shown below.

In this there are two sections one which has the preference SAMPLE CODE SNIPPETS and the second section at what level you want to apply these snippets. You can apply snippets at two levels one at a project workspace level and other at user / computer level. If you apply at the workspace level it is only for that project and if you apply at the user level it will be applied for all projects in that computer.

For example in the user settings we have pasted the file exclude settings. In this we have specified we do not want to see the ".JS" and ".Map" files in the workspace.

```
{
    "workbench.sideBar.location": "left",
    "window.zoomLevel": 2,
    "window.menuBarVisibility": "default",
    "files.exclude": {
    "**/*.js": true,
    "**/*.js.map": true,
    "**/.hg": true,
    "**/CVS": true,
    "**/.DS_Store": true
  },
  "workbench.colorTheme": "Visual Studio Light"
}
```

Once that setting is applied you can see the "JS" and "Map" files are not seen.



Lab 4: - Understanding Module loaders using SystemJS

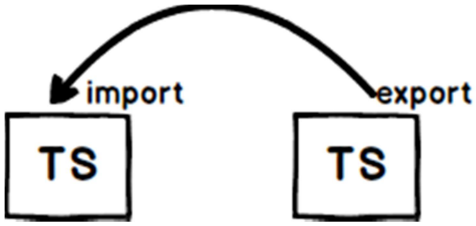
You can watch the below videos which demonstrates concept of Module Loaders and SystemJS practically.

Topic name	YouTube URL source
SystemJS	https://www.youtube.com/watch?v=nQGhdoIMKaM
CommonJS concept	https://www.youtube.com/watch?v=jN4IM5tp1SE

Step 1: - TypeScript Modules and Import/Export keywords

Modular development is one of the important pillars of development. A good software will always have self-contained modules. So you would like to create separate physical typescript or JavaScript files which will have self-contained code. Then you would like have to some kind of reference mechanism by which modules can be referred between those physical files.

In TypeScript, we do this by using "import" and "export" keywords.



So the modules which need to be exposed should have "export" keyword while modules which want to import the exported modules should have "import" keyword.

For instance, let's say we have two typescript files "Customer.ts" and "Dal.ts". Let's assume "Customer.ts" is using "Dal.ts".

So "Dal.ts" will use export to expose his modules while "Customer.ts" will use to import to get the exported module.

So in "Dal.ts" the classes which you want to export should be marked as "exported" as shown in the below code. If you do not mark it exported it cannot be imported.

```
export class Dal{
    Add(){
        alert("Dal add called");
    }}
```

Now in the “Customer.ts” we use “import” to call the exported class from the “Dal.ts”.

```
import {Dal} from "./Dal "
export class Customer{
    Add(){
       var dal = new Dal();
       dal.Add();
    }
}
```

So in short you use export and import to do modular development in typescript. But now how does this “TRANSPILE” to JavaScript code that we will see in the next section. At the end of the day all these modules are transpiled to JavaScript so let’s understand how that works under the hoods.

Step 2: - Module formats in JavaScript CommonJS, AMD, ES6

Let’s first define this word “Module” formats. We talked about modules in the previous section. Module formats define the JavaScript syntaxes of how the module should be exported and imported. In JavaScript world there are two ways of defining module formats: - Unofficial way and Official way. So prior to ES6 there was no official way so some of the unofficial viral ways of defining module formats are CommonJS , AMD , UMD and so on. Official way is only and only one ES6.

For instance, below is the format of CommonJS. In CommonJS the module which is exported is defined in “exports” variables and to import we need to use “require” keyword. You can see below is the JS output where the dal is exported using “exports” variable.

```
Object.defineProperty(exports, "__esModule", { value: true });
var Dal = (function () {
    function Dal() {
    }
    Dal.prototype.Add = function () {
        alert("Dal add called");
    };
    return Dal; }());
exports.Dal = Dal;
```

Below is the code for “Customer.js” which uses “require” to load “Dal.js”.

```
Object.defineProperty(exports, "__esModule", { value: true });
var Dal_js_1 = require("./Dal.js");
var Customer = (function () {
    function Customer() {
    }
    Customer.prototype.Add = function () {
        var dal = new Dal_js_1.Dal();
        dal.Add();
    };
    return Customer;
}());
exports.Customer = Customer;
```

So now this is a CommonJS format in the same way we have other formats as well. For example below is “amd” module format.

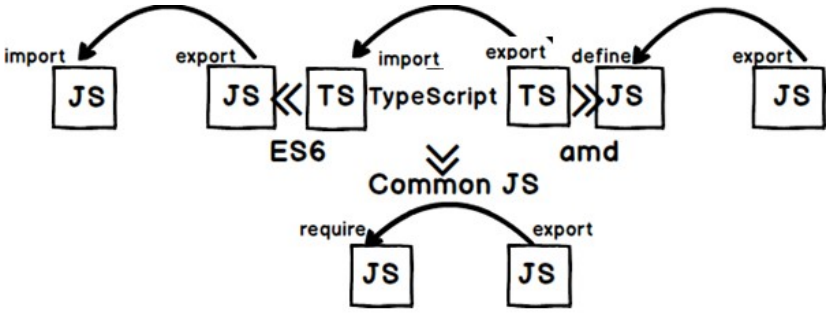
In this we export the classes in the “export” variable and use “define” to import. Below is the code of “define”. We are not pasting of “export” as it’s same like CommonJS.

```
define(["require", "exports", "./Dal.js", "./Validation.js"], function
(require, exports, Dal_js_1, Validation_js_1) {
    "use strict";
    Object.defineProperty(exports, "__esModule", { value: true });
    var Customer = (function () {
        function Customer() {
        }
        Customer.prototype.Add = function () {
            var val = new Validation_js_1.Validation();
            var dal = new Dal_js_1.Dal();
            dal.Add();
        };
```

```
        return Customer;
    }());
    exports.Customer = Customer;
});
```

In “ES6” module format to expose the class we need to “export” keywords and to consume we need to use “import”.

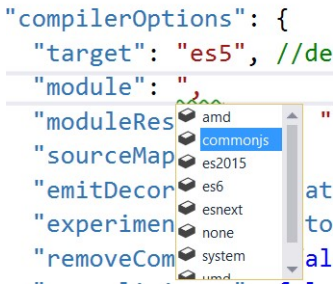
```
import { Dal } from "./Dal.js";
import { Validation } from "./Validation.js";
var Customer = (function () {
    function Customer() {
    }
    Customer.prototype.Add = function () {
        var val = new Validation();
        var dal = new Dal();
        dal.Add();
    };
    return Customer;
}());
export { Customer };
var Dal = (function () {
    function Dal() {
    }
    Dal.prototype.Add = function () {
        alert("Dal add called");
    };
    return Dal;
}());
export { Dal };
```



So in simple words “amd”,”CommonJS” and “ES6” define how modules will communicate with each other. Concluding ES6 uses “import / export”, amd uses “define/export” and CommonJS uses “require/export”.

All these module formats can be generated with simple an option change in TypeScript config file.

So in “tsconfig.json” we can set in “module” which module format we want.



Step 3: - Calling JavaScript module loaders in HTML UI

Now when we try to load JavaScript functions which are using module formats like AMD, CommonJS or ES6 it’s not so easy. For example in below code in HTML UI we have loaded “Dal.js” and “Customer.js”. This example has been demonstrated in the previous lab and is having “CommonJS” enabled.

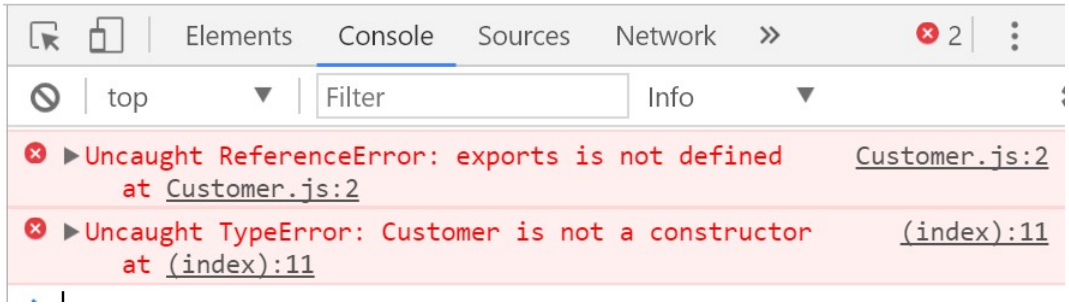
Also we have put the sequence properly, first we have added reference of “Dal.js” and then “Customer.js” because “Customer.js” is dependent on “Dal.js”.

But when we try to create “Customer” object and try to call “Add” it does not work.

```
<script src="Dal.js"></script>
<script src="Customer.js"></script>
    <script>
       var cust = new Customer();
       cust.Add();
    </script>
```

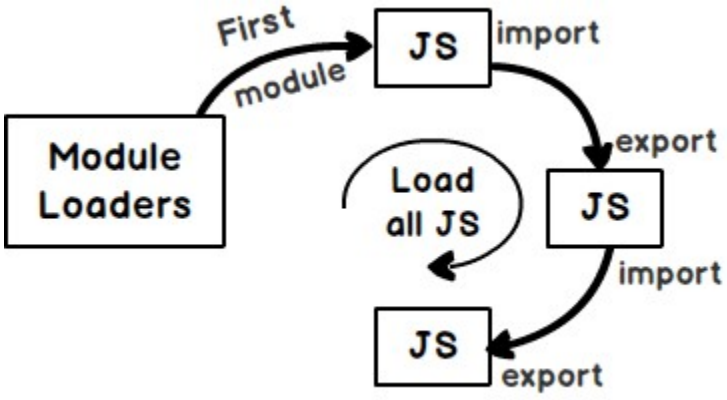
We end up with an error below stating that “exports” is not understood. That makes sense because browser does not know any keywords like “exports” and “require” as its not standard JavaScript.

The second problem is even if this code had worked i would still have ordering problems for large number of references. Let us say we have 15 modules which are referencing using module formats we would end with spending half-life arranging those sequences in HTML file. It would be great if we can just point to “Customer.js” and automatically using “exports” and “imports” the references is identified and “Address.js” is loaded.



That’s where we need JavaScript module loaders. Some example of module loaders are SystemJS , WebPack and so on.

So if we are using module loaders we just need to point to the first JS file and automatically using the “import/require/define” and “exports” it will get references of all the dependent JS files and load them accordingly.



Let’s demonstrate a module using “SystemJS”. So first go to “Node command prompt” and install “systemjs”.

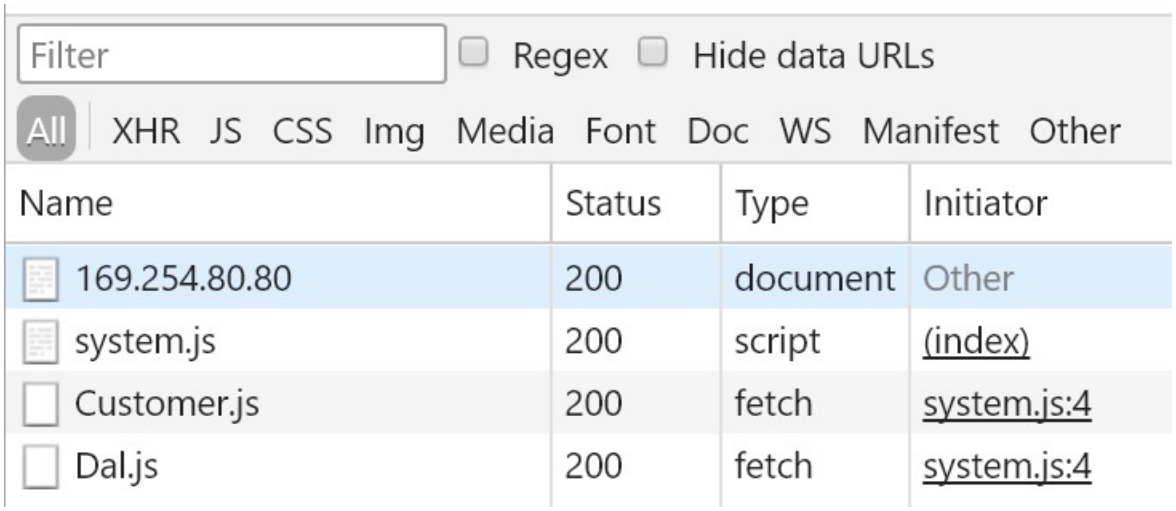
```
npm install systemjs
```

So in the HTML UI we need to tell “system.js” which is the first JS file to be loaded. You can see in the below code we are saying “SystemJS.import” load “Customer.js” as the first file.

```
<script src="system.js"></script>
    <script>

        SystemJS.import('./Customer.js')
        .then(function(module){
                var cust = new module.Customer();
                cust.Add();
        }).catch(function (err)
        { console.error(err); });;
    </script>
```

Once the file has been loaded in the then function we get the modules. We can then refer the module variable and create object of “Customer” function.

If you watch the network tab of chrome browser you can see first “system.js” loads “Customer.js” and then also loads its reference that is “Dal.js”.



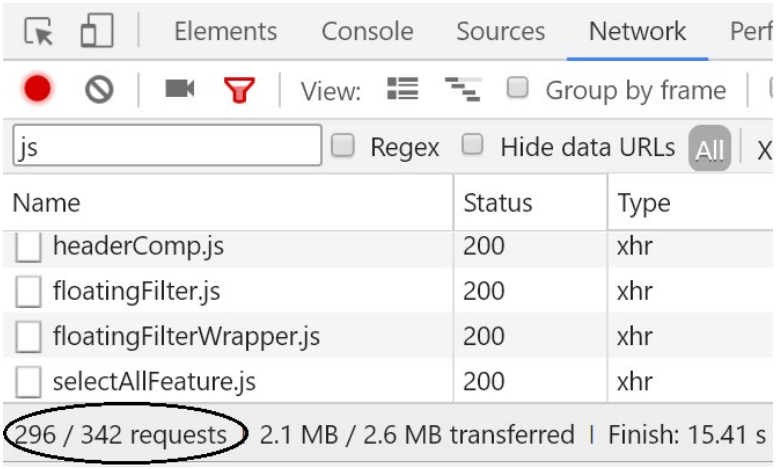
Lab 5:- Understanding Module Bundlers using WebPack

Introduction: - Runtime vs Compile time – Webpack Module bundler

In the previous Lab 4 systemJS was doing everything at the runtime. So in the browser it first loads “Customer.js” then “Address.js” and so on. If you have lot of dependencies you would end up with lot of requests.

At the side is a simple image from an enterprise project where we have 342 JavaScript file requests to load the site.

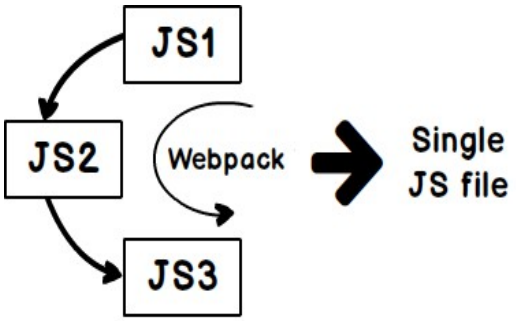
Now that’s a big number and with so many requests your application performance will come down a lot.



If we can create a SINGLE BUNDLE during the compile time itself, that would be a great performance booster.

That’s where WebPack (https://webpack.js.org/) comes to use. WebPack is a module bundler.

It takes the first JS file uses module definitions like CommonJS/AMD/UMD etc. and figures out the references and generates one Single JS file DURING COMPILE TIME. You can take this one single bundle JS and put in your web page.



So let's try to understand the basics of how WebPack works.

Step 1: - Install WebPack

So the first step is to install WebPack. So open the node command prompt and type the below NPM command. Please see we have use "-g" the global flag.

```
npm install -g webpack
```

Step 2: - Generate a Single Bundle

We take the same code which we have used in "Lab 4". In lab 4 if you see we have "Customer.js" calling "Address.js". So the first JavaScript file is "Customer.js". We just need to give the first JS file name in the WebPack command the final bundle file name. You can also see there is "—output-library" flag parameter. WebPack will expose "CustomerLibrary" component to us from where we can access the "Customer" class in the UI.

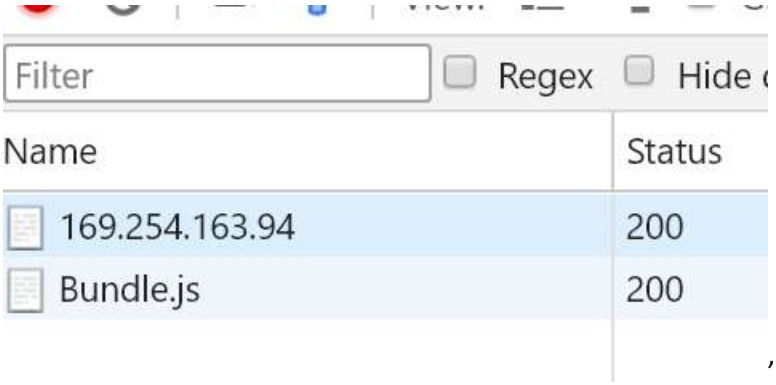
```
webpack "Customer.js" "Bundle.js" --output-library='CustomerLibrary'
```

Step 3: - Calling the javascript class in the webpage.

So now that we have a single bundle we can just load the JS file in the webpage and access the classes via the "CustomerLibrary". Remember this "CustomerLibrary" is coming from command line, please revisit step 2 again for details.

```
<script src="Bundle.js"></script>
    <script>
        var x = new CustomerLibrary.Customer();
        x.Add();
    </script>
```

If you now see the network tab you will see now only one single file "Bundle.js" as compared to multiple files.



SystemJS and WebPack. When we learn Angular we will first use SystemJS and then in one of the labs we will see how WebPack helps us.

So now that we have completed all the prerequisites it's time to start getting in to Angular. So from Lab 6 the actual Angular starts.

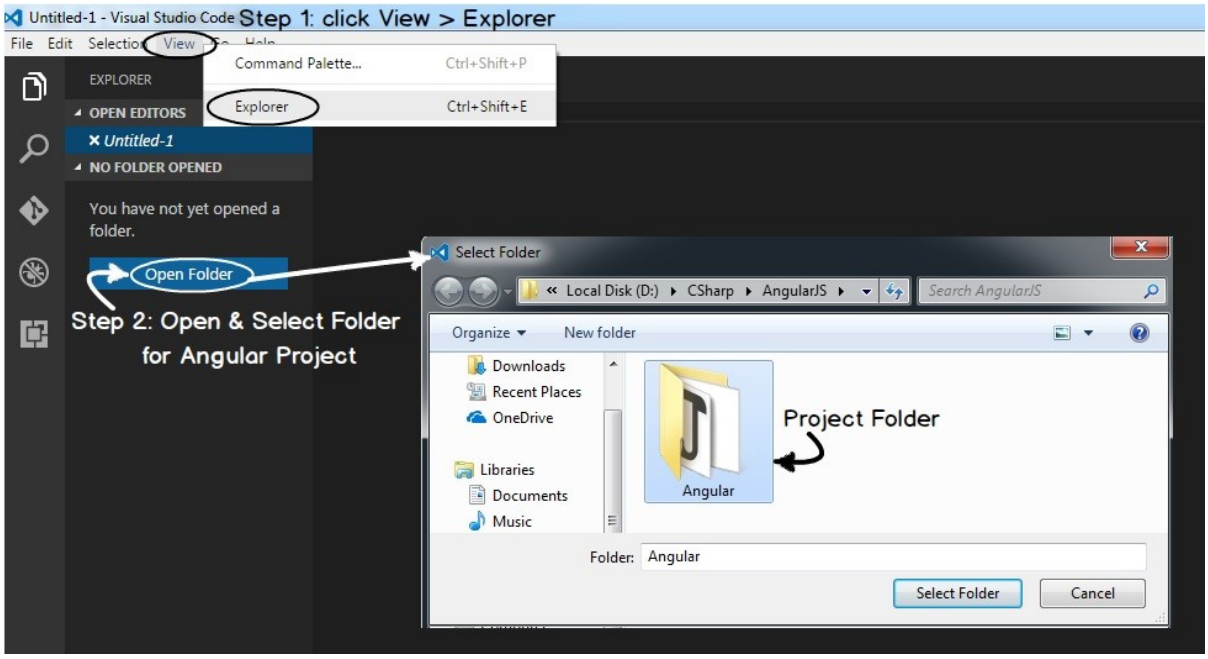
Lab 6: - Components and Modules (Running your first Angular Application)

Introduction

In this lab we will configure Angular environment and we will try to create the basic Customer screen and make it up and running. So let's start with the first step downloading angular framework and configuring typescript compiler.

Step 1: - NPM install to get Angular framework

So let's create a folder called as "Angular" and open the same using VS code. Please refer Lab 3 on how to use VS code. In case you want to Angular using visual studio you can see this video https://www.youtube.com/watch?v=oMgvi-AV-eY . If you are eclipse guy please mail me at questpond@questpond.com



So as we discussed in Lab 1 that “Node” has “npm” which helps us to get JavaScript open sources. So create a package.json file with the below details and do an “npm install” in the integrated command line of VS code.

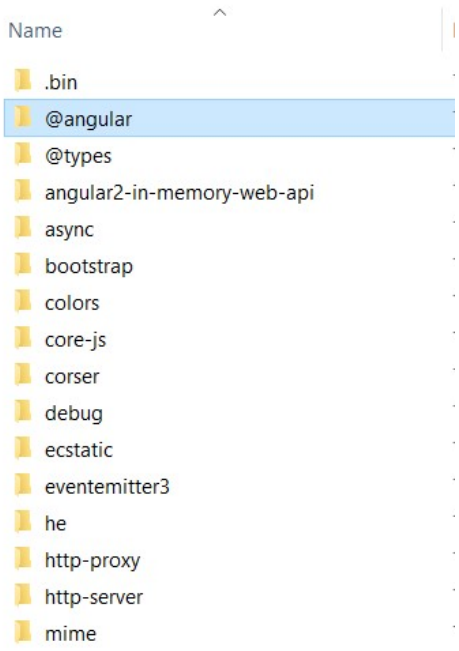
```
{
  "name": "angular-quickstart",
  "version": "1.0.0",
  "license": "ISC",
  "dependencies": {
    "@angular/common": "4.0.0",
    "@angular/compiler": "4.0.0",
    "@angular/core": "4.0.0",
    "@angular/forms": "4.0.0",
    "@angular/http": "4.0.0",
    "@angular/platform-browser": "4.0.0",
    "@angular/platform-browser-dynamic": "4.0.0",
    "@angular/router": "4.0.0",
```

```
    "@angular/upgrade": "4.0.0",
    "angular-in-memory-web-api": "0.3.2",
    "bootstrap": "^3.3.6",
    "core-js": "^2.4.1",
    "http-server": "^0.10.0",
    "reflect-metadata": "^0.1.3",
    "rxjs": "^5.4.1",
    "systemjs": "0.19.27",
    "zone.js": "^0.8.4"
  },
  "devDependencies": {
    "@types/core-js": "^0.9.42",
    "typescript": "2.3.4"
  }
}
```

If all the npm install runs successfully you should see a node_modules folder in your VS code project directory with the following folder structure.

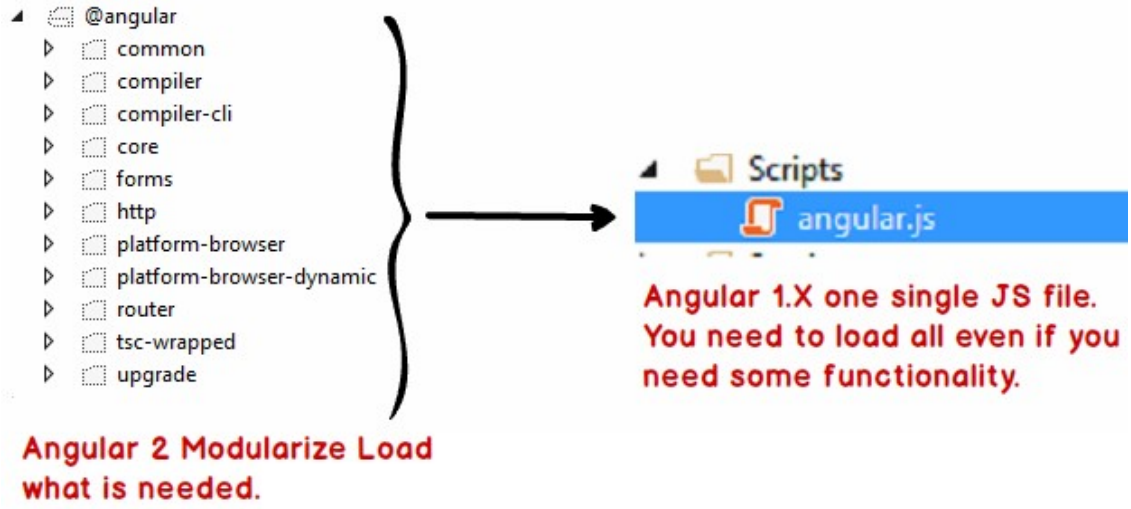
If you go inside the “Angular” folder you will see lot of folders like common, http, forms core and so on. Putting in simple words Angular is MODULAR.

They have developed isolated components for Forms, Http, Core and so on.



In AngularJS 1.X we just had one JS file which had the whole framework. So you drag and drop that single Angular 1.X JS file on HTML page and you are ready with the Angular 1.X environment. But the problem with having whole framework in one JS files was that you have to include all features even if you need it or not.

For instance, if you are not using HTTP still that feature will be loaded. In case of Angular we have separate discrete components which make Angular awesome.

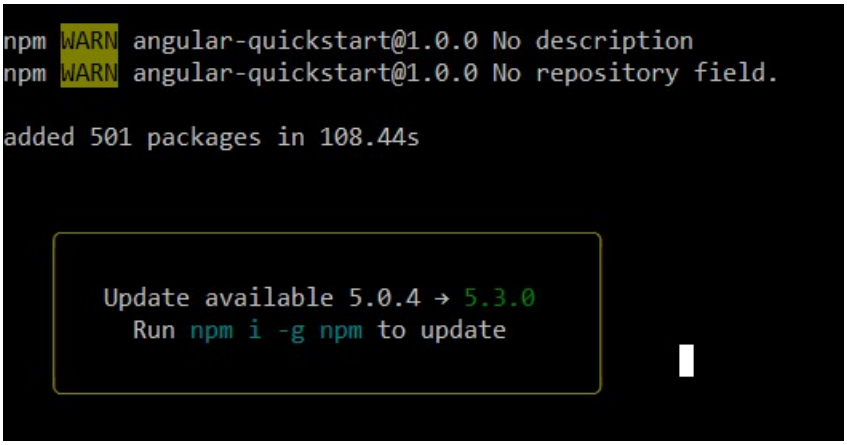


Common errors during NPM install

This is the most crucial part of Angular. You can see that in "package.json" file we have so many dependencies and these dependencies use each other. So if one version is incompatible the whole NPM command will fail. So below are the couple of things that you need to take care.

First keep NPM updated to the latest version or else you can end up with a warning as shown at the side.

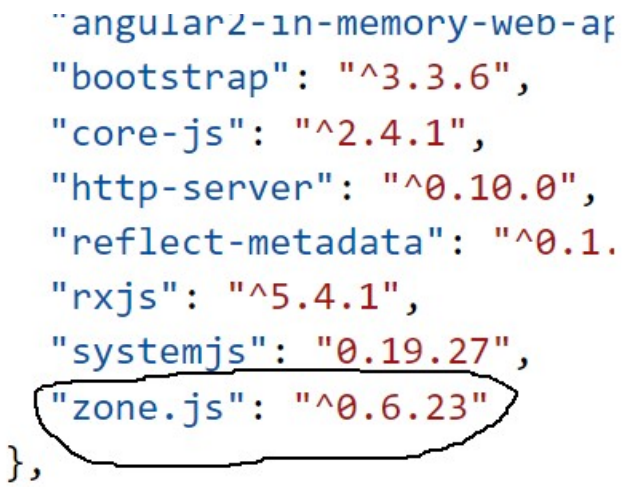
AND EVEN IF YOU DO NOT GET THIS WARNING STILL KEEP THE LATEST NPM VERSION.



To get the latest NPM version you need to use “npm install -g npm”. This command will update NPM itself. Also you can get the below error which shows incompatibility between modules. For example in the below figure it says “For Angular 4.0 you will need Zone.js which is greater than 0.8.4 version.

```
\SHIV\Downloads\Lab6CustomerAngularProject\Lab6CustomerAngularProject>npm install
@angular/core@4.0.0 requires a peer of zone.js@^0.8.4 but none was installed.
angular-quickstart@1.0.0 No description
angular-quickstart@1.0.0 No repository field.
```

So go to your project.json, fix the version number, DELETE node_modules folder and do a NPM again.



Errors of version mismatch can be huge as shown in the below figure. Sometimes fixing “package.json” can be more painful and iterative. When you do these iterations please ensure you delete “node_modules” folder so that you are not referring old versions.

```
added 43 packages in 63.082s

C:\Users\SHIV\Downloads\Lab6CustomerAngularProject\Lab6CustomerAngularProject>npm install
npm WARN angular2-in-memory-web-api@0.0.20 requires a peer of @angular/core@^2.0.0 but none was installed.
npm WARN angular2-in-memory-web-api@0.0.20 requires a peer of @angular/http@^2.0.0 but none was installed.
npm WARN angular2-in-memory-web-api@0.0.20 requires a peer of rxjs@5.0.0-beta.12 but none was installed.
npm WARN angular2-in-memory-web-api@0.0.20 requires a peer of zone.js@^0.6.23 but none was installed.
npm WARN angular-quickstart@1.0.0 No description
npm WARN angular-quickstart@1.0.0 No repository field.
```

You can also get error as shown below “ERR nottarget”. This error says that the particular version number is not found. Use “npm view packagename version” command to get the latest version and update your package accordingly.

```
C:\Users\SHIV\Downloads\Lab6CustomerAngularProject\Lab6CustomerAngularProject>npm install
npm ERR! code ETARGET
npm ERR! notarget No matching version found for zone.js@^0.8.23
npm ERR! notarget In most cases you or one of your dependencies are requesting
npm ERR! notarget a package version that doesn't exist.
npm ERR! notarget
npm ERR! notarget It was specified as a dependency of 'angular-quickstart'
npm ERR! notarget
```

Step 2:- TypeScript configuration tsconfig.json

Also in the same folder we need to create tsconfig.json which defines how typescript will compile. Do not worry on what settings are there in it we will explain as we go ahead. Remember do not OVER LEARN. For now just note that the below file defines how TSC will compile.

```
{
  "compilerOptions": {
    "target": "es5",//defines what sort of code ts generates, es5 because it's what most browsers currently UNDERSTANDS.
    "module": "commonjs",
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,//for angular to be able to use metadata we specify in our components.
    "experimentalDecorators": true,//angular needs decorators like @Component, @Injectable, etc.
    "removeComments": false,
    "noImplicitAny": false,
    "noStrictGenericChecks": true,
    "lib": ["es2016", "dom"]

  }
}
```

Step 3:- Configuring SystemJS module loader

SystemJS file defines configuration of how SystemJS will load the frameworks. So this file put it in the same folder with the name “systemjs.config.js”. The name should be EXACTLY “systemjs.config.js” because SystemJS looks with that name.

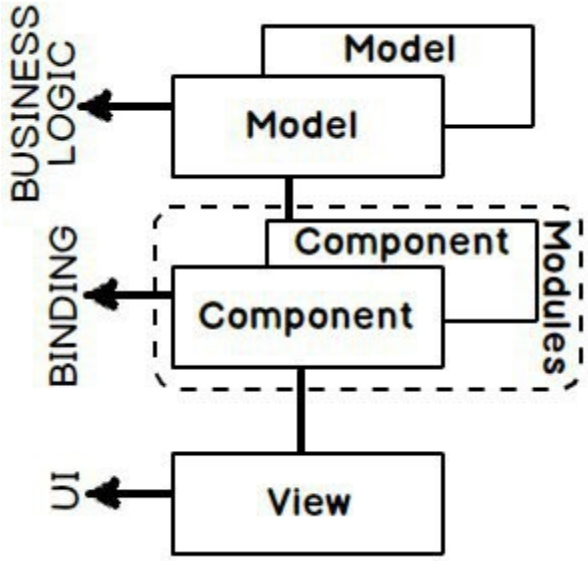
```
 (function (global) {
    System.config({
        paths: {
            // paths serve as alias
            'npm:': '../node_modules/'
        },
        // map tells the System loader where to look for things
        map: {
            // our app is within the app folder
            app: 'startup',
            // 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',
            '@angular/platform-browser': 'npm:@angular/platform-
browser/bundles/platform-browser.umd.js',
            '@angular/platform-browser-dynamic': 'npm:@angular/platform-
browser-dynamic/bundles/platform-browser-dynamic.umd.js',
            '@angular/http': 'npm:@angular/http/bundles/http.umd.js',
            '@angular/router': 'npm:@angular/router/bundles/router.umd.js',
            '@angular/forms': 'npm:@angular/forms/bundles/forms.umd.js',
            'lodash':'node_modules/lodash',
            // other libraries
            'rxjs': 'npm:rxjs',
            'angular-in-memory-web-api': 'npm:angular-in-memory-web-
api/bundles/in-memory-web-api.umd.js',
            'moment': 'npm:moment',
```

```
            // ag libraries
            'ag-grid-ng2': '../node_modules/ag-grid-ng2',
            'ag-grid': '../node_modules/ag-grid',
       'jquery': '../node_modules/jquery/dist/jquery.js',
            'ng2-auto-complete': '../node_modules/ng2-auto-complete/dist'
        },
        // packages tells the System loader how to load when no filename and/or
no extension
        packages: {
            app: {
                main: '../Startup/Startup.js',
                defaultExtension: 'js'
            },
            rxjs: {
                defaultExtension: 'js'
            },
            'angular-in-memory-web-api': {
                main: './index.js',
                defaultExtension: 'js'
            },
            'lodash': { main: './index.js', defaultExtension: 'js' },
            moment: {
                defaultExtension: 'js'
            },
            'ag-grid-ng2': {
                defaultExtension: "js"
            },
            'ag-grid': {
                defaultExtension: "js"
            },
            'ng2-auto-complete': {
                main: 'ng2-auto-complete.umd.js', defaultExtension: 'js'
```

```
            }
        }
    });
})(this);
```

Understanding Angular Component and module architecture

As we said in the previous section that the whole goal of Angular is binding the model and the view. In Angular the binding code is officially termed as “Component”. So hence forth we will use the word “Component” for the binding code.



In enterprise projects you can have lot of components. With many components it can become very difficult to handle the project. So you can group components logically in to modules.

So hence forth I will be using two terms: -

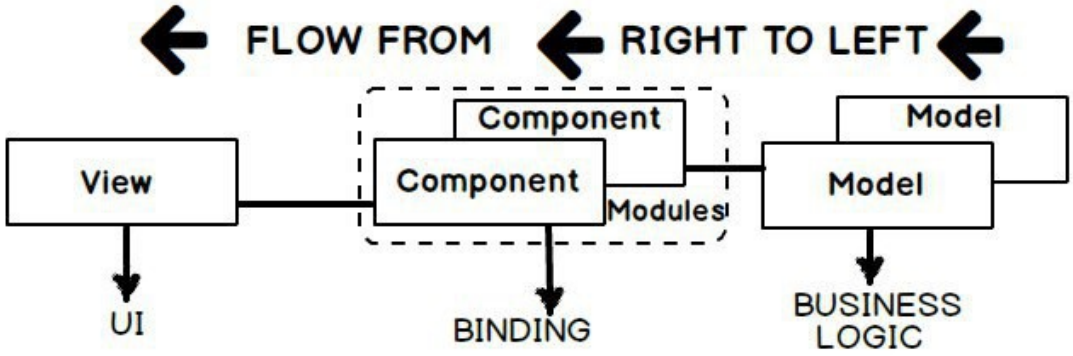
Components: - This will have the binding logic to bind the UI and the model.

Modules: - This will logically group components.

Step 4: - Following MVW Step by Step – Creating the folders

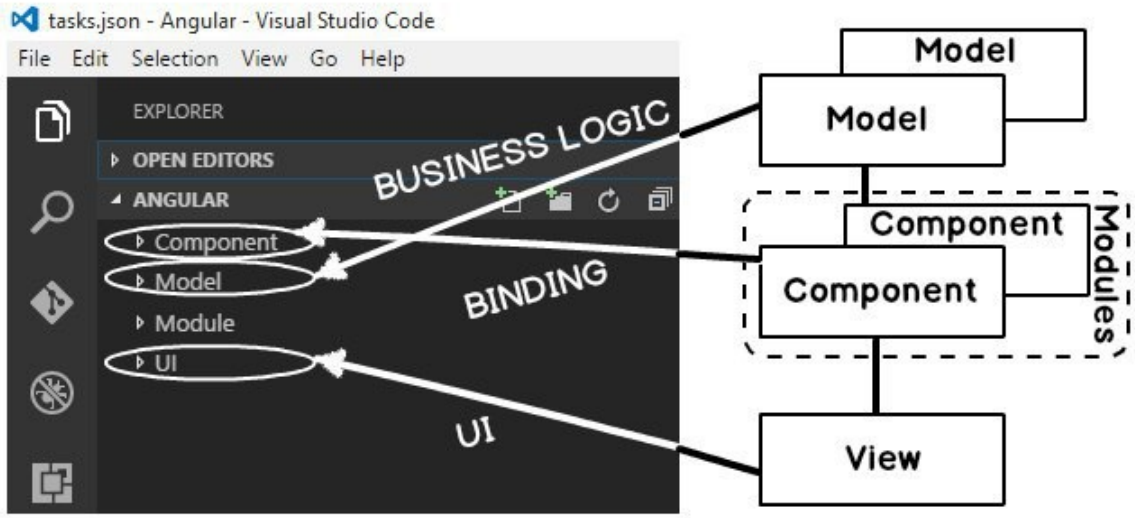
Before we start coding lets visualize the steps of coding. As we have said Angular is a binding framework. It follows MVW architecture. It binds HTML UI with the JavaScript code (model). So if we visualize it will looks something as shown in the image below. So let’s move from right to left. So let’s do the coding in the following sequence:-

1. Create the model.
2. Create the Component.
3. Create the module.
4. Create the HTML UI.

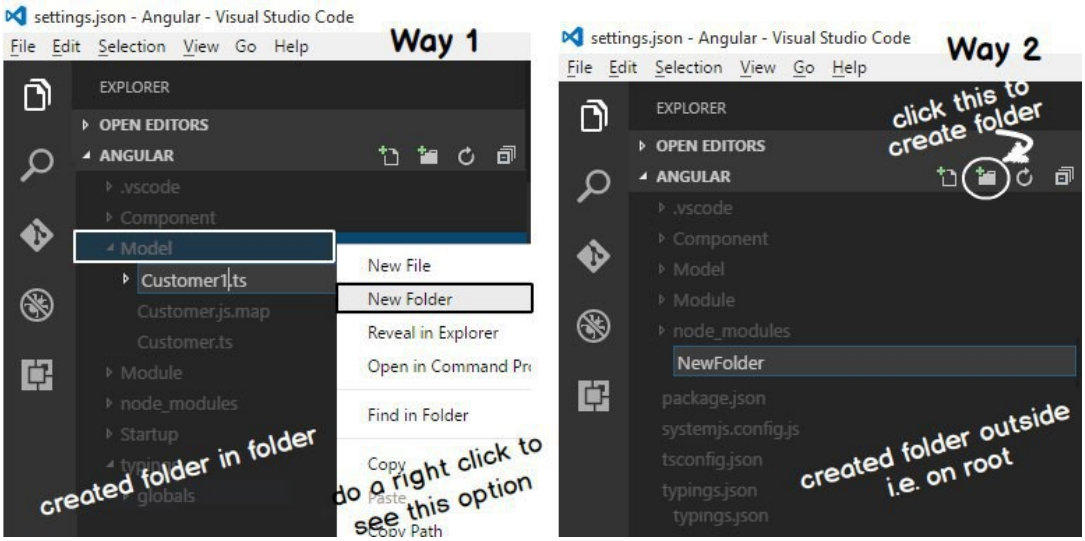


So let's first create four folders in our project:-

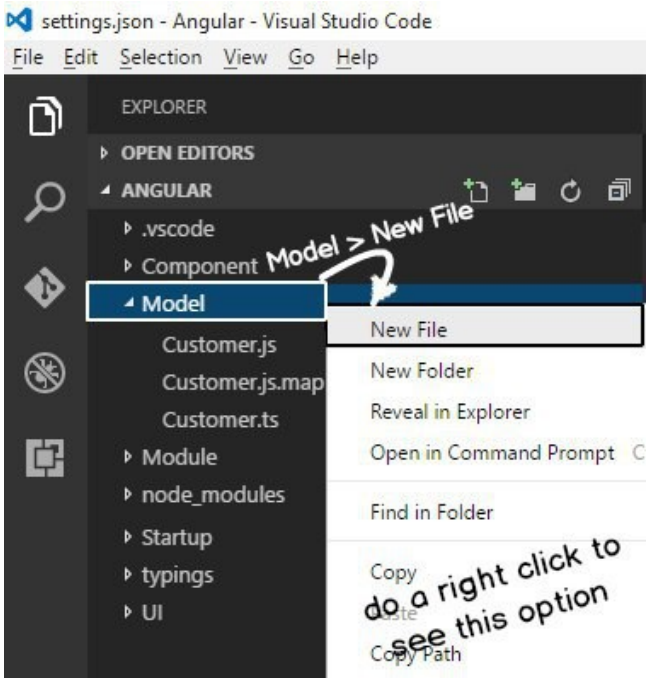
- **View folder: -** This folder will contain the HTML UI.
- **Model folder: -** This folder will have the main business typescript classes.
- **Component folder: -** This folder will have the binding code which binds the HTML UI and Model.
- **Module: -** This folder will have code which will logically group the components.



In order to create a folder in VS code you can use the "New folder" icon or you can right click and also create a folder.



Step 5:- Creating the model



A model is nothing but a class with properties and behavior. So let us first create the customer model with three properties “CustomerName”, “CustomerCode” and “CustomerAmount”.

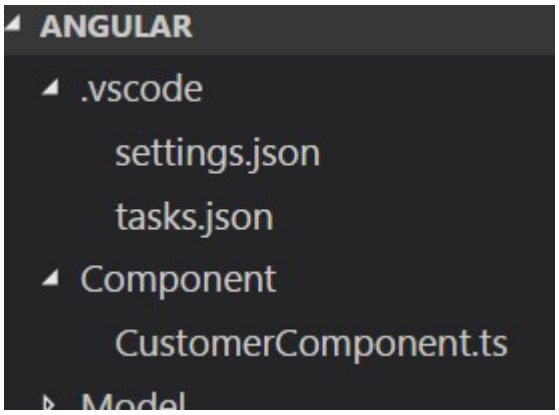
So right click on the “Model” folder and add a new file “Customer.ts”. Keep the extension of this file as “.ts” as it’s a typescript file.

While compiling the typescript command identifies only files with the extension “.ts”.

In the “Customer.ts” let’s create a “Customer” class with three properties. In this book we will not be going through the basics of TypeScript , please do go through this 1 hour training video of typescript which explains typescript in more detail.

```
export class Customer {
    CustomerName: string = "";
    CustomerCode: string = "";
    CustomerAmount: number = 0;
}
```

Step 6:- Creating the Component



The next thing we need to code is the binding code. Binding code in Angular is represented by something termed as “COMPONENTS”. Angular components have the logic which helps to bind the UI with the model.

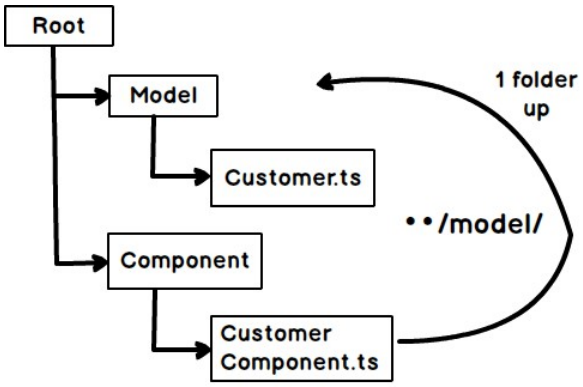
So right click on the component folder and add “CustomerComponent.ts” file as shown in the figure at the left.

In the component we need to import two things the Angular core and our Customer model. Please note “import” is a typescript syntax and not JavaScript. So in case you are not following the code , please see this Learn TypeScript in 1 hour video before moving ahead.

```
import {Customer} from '../Model/Customer'
import {Component} from "@angular/core"
```

The first line imports the “Customer” class in to the “CustomerComponent.ts”. This import is only possible because we have written “export” in “Customer.ts” file. The import and export generate code which follows CommonJS , AMD or UMD specifications. In case you are new to these specifications please see this CommonJS video which explains the protocol in more detail.

```
import {Customer} from '../Model/Customer'
```



Let's try to understand how "Customer" component is located. If you see it's using a relative path. In the import it says "../Model/Customer".

The "../" says go one folder up. So we are currently in the "Component" folder, so with "../" it travels to the root and from that point it makes entry in to the "Model" folder.

The next import command imports angular core components. In this we have not given any relative path using "../" etc. So how does typescript locate the angular core components?

```
import {Component} from "@angular/core"
```

If you remember we had used node to load Angular and node loads the JS files in the "node_modules" folder. So how does typescript compiler automatically knows that it has to load the Angular components from "node_modules" folder.
Typescript compiler uses the configuration from "tsconfig.json" file. In the configuration we have one property termed as "moduleResolution. It has two values:-

- **Classic: -** In this mode typescript relies on "./" and "../" to locate folders.
- **Node: -** In this mode typescript first tries to locate components in "node_modules" folder and if not found then follows the "../" convention to traverse to the folders.

In our tsconfig.json we have defined the mode as "node" this makes typescript hunt modules automatically in "node_modules" folder. That makes the "import" of Angular components work.

```
{
 {
  ....
  ....
    "moduleResolution": "node",
 ....
 ....
  }
}
```

So now that both the import statements are applied let us create the “CustomerComponent” and from that let’s expose “Customer” object to the UI with the object name “CurrentCustomer”.

```
export class CustomerComponent {
    CurrentCustomer:Customer = new Customer();
}
```

As we said previously that component connects / binds the model to the HTML UI. So there should be some code which tells that “CustomerComponent” is bounded with HTML UI. That’s done by something termed as “Component MetaData Attribute”. A component metadata attribute starts with “@Component” which has a “templateUrl” property which specifies the HTML UI with which the component class is tied up with.

```
@Component({
    selector: "customer-ui",
    templateUrl: "../UI/Customer.html"
})
```

This attribute is then decorated on the top of the component. Below goes the full code.

```
@Component({
    selector: "customer-ui",
    templateUrl: "../UI/Customer.html"
})
export class CustomerComponent {
    CurrentCustomer:Customer = new Customer();
}
```

Putting in simple words binding code is nothing but a simple typescript class which decorated by the “@Component” attribute which dictates that this typescript class is binded with which UI.

Below goes the full code of the Angular component.

```
// Import statements
import {Component} from "@angular/core"
import {Customer} from '../Model/Customer'

// Attribute metadata
@Component({
    selector: "customer-ui",
    templateUrl: "../UI/Customer.html"
})
// Customer component class exposing the customer model
export class CustomerComponent {
    CurrentCustomer:Customer = new Customer();
}
```

Step 7:- Creating the Customer HTML UI – Directives and Interpolation

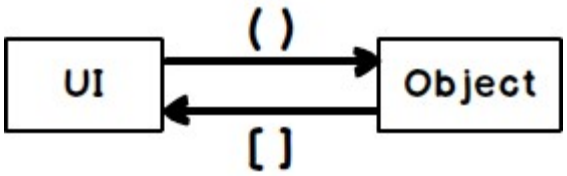
Now from the "CustomerComponent" , "Customer" is exposed via the "CurrentCustomer" object to UI. So in the HTML UI we need to refer this object while binding.

In the HTML UI the object is binded by using "Directives". Directives are tags which direct how to bind with the UI.

For instance if we want to bind "CustomerName" with HTML textbox code goes something as shown below:-

- "[(ngModel)]" is a directive which will help us send data from the object to UI and vice versa.
- Look at the way binding is applied to the object. It's referring the property as "CurrentCustomer.CustomerName" and not just "CustomerName". Why ??. Because if you remember the object exposed from the "CustomerComponent" is "CurrentCustomer" object. So you need to qualify "CurrentCustomer.CustomerCode".

```
<input type="text" [(ngModel)]="CurrentCustomer.CustomerName">
```



- Round brackets indicate data sent from UI to object.
- Square brackets indicate data is sent from object to UI.
- If both are present then it's a two way binding.

There would be times when we would like to display object data on the browser. By using "{{"braces we can display object data with HTML tags. In the below HTML we are displaying "CustomerName" mixed with HTML BR tag. These braces are termed as "INTERPOLATION". If you see the dictionary meaning of interpolation it means inserting something of different nature in to something else.

In the below code we are inserting object data within HTML.

```
{{CurrentCustomer.CustomerName}}<br />
```

Below goes the full HTML UI code with binding directives and interpolation.

```
<div>
Name:
<input type="text" [(ngModel)]="CurrentCustomer.CustomerName"><br /><br
/>
Code:
<input type="text" [(ngModel)]="CurrentCustomer.CustomerCode"><br /><br
/>
Amount:
<input type="text" [(ngModel)]="CurrentCustomer.CustomerAmount"><br
/><br />
</div>
{{CurrentCustomer.CustomerName}}<br /><br />
{{CurrentCustomer.CustomerCode}}<br /><br />
{{CurrentCustomer.CustomerAmount}}<br /><br />
```

Step 8:- Creating the Module

Module is a container or you can say it's a logical grouping of components and other services.

So the first import in this module is the “CustomerComponent” component.

```
import { CustomerComponent }   from '../Component/CustomerComponent';
```

We also need to import “BrowserModule” and “FormsModule” from core angular. “BrowserModule” has components by which we can write IF conditions and FOR loop. “FormsModule” provides directive functionality like “ngModel”, expressions and so on.

```
import { BrowserModule } from '@angular/platform-browser';
import {FormsModule} from "@angular/forms"
```

We also need to create a TypeScript class “MainModuleLibrary”. At this moment this class does not have any code but it can have code which will provide component level logic like caching, initialization code for those group of components and so on.

```
export class MainModuleLibrary { }
```

To create a module we need to use import “NgModule” from angular core. This helps us to define module directives.

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

“NgModule” has three properties:-

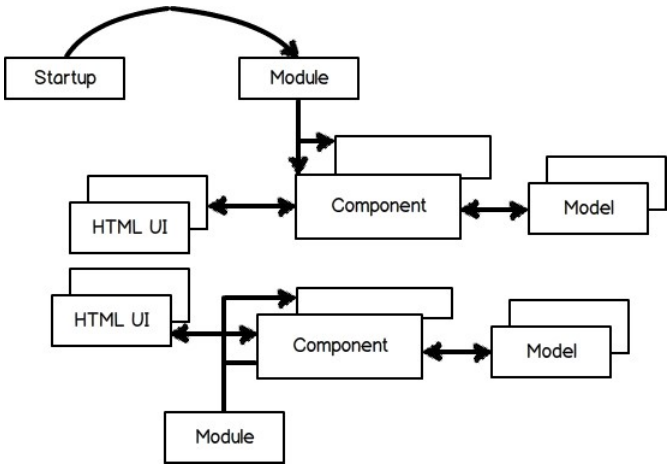
- **Imports: -** If this module is utilizing other modules we define the modules in this section.
- **Declarations: -** In this section we define the components of the modules. For now we only have one component ‘CustomerComponent”.
- **Bootstrap: -** This section defines the first component which will run. For example we can have “HomeComponent”, “CustomerComponent” and so on. But the first component which will run is the “HomeComponent” so that we need to define in this section.

```
@NgModule({
    imports: [BrowserModule,
              FormsModule],
    declarations: [CustomerComponent],
    bootstrap: [CustomerComponent]
})
```

Below goes the full code of Angular module which we discussed in this section.

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import {FormsModule} from "@angular/forms"
import { CustomerComponent }   from '../Component/CustomerComponent';
@NgModule({
    imports: [BrowserModule,
              FormsModule],
    declarations: [CustomerComponent],
    bootstrap: [CustomerComponent]
})
export class MainModuleLibrary { }
```

Step 9:- Creating the "Startup.ts" file



So we have created the UI, we have created the models, we have created components and these components are grouped in to modules. In an angular application you can have many modules.

So you need to define the startup module. So let's create a "Startup.ts" file which will define the startup module.

Below goes the "Startup.ts" file in which we have defined which module will be bootstrapped.

```
import { platformBrowserDynamic } from '@angular/platform-browser-
dynamic';
import { MainModuleLibrary } from '../Module/MainModuleLibrary';
const platform = platformBrowserDynamic();
platform.bootstrapModule(MainModuleLibrary);
```

Step 10: - Invoking "Startup.ts" file using main angular page

So let us create a startup HTML page which will invoke the "Startup.ts". Now in this page we will need to import four JavaScript framework files Shim, Zone, Meta-data and System JS as shown in the below code.

```
<script src="../../node_modules/core-js/client/shim.min.js"></script>
<script src="../../node_modules/zone.js/dist/zone.js"></script>
<script src="../../node_modules/reflect-metadata/Reflect.js"></script>
<script src="../../node_modules/systemjs/dist/system.src.js"></script>
```

Following are the uses of JS files:-

Shim.min.js	This framework ensures that ES 6 JavaScript can run in old browsers.
Zone.js	This framework ensures us to treat group of async activities as one zone.
Reflect.js	Helps us to apply meta-data on JavaScript classes. We are currently using @NgModule and @NgComponent as attributes.
System.js	This module will helps to load JS files using module protocols like CommonJS, AMD or UMD.

In this HTML page we will are calling the "systemjs.config.js" file. This file will tell system JS which files to be loaded in the browser.

```
<script src="../systemjs.config.js"></script>
<script>
    System.config({
        "defaultJSExtensions": true
    });

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

In the "import" we need to specify "startup" which will invoke "startup.js" file.

```
    System.import('startup').catch(function (err) { console.error(err);
});
```

Our customer screen is with the name “Customer.html”. So to load in to this screen we need to define a place holder. So in this place holder our Customer HTML page will load.

```
    <customer-ui></customer-ui>
```

If you remember when we created the component class we had said to load the HTML page in a selector. So that selector is nothing but a tag (placeholder) to load our Customer page.

```
@Component({
    selector: "customer-ui",
    templateUrl: "../UI/Customer.html"
})
```

Below goes the full HTML page with all scripts and the place holder tag.

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
      <meta charset="utf-8" />
</head>
<!-- 1. Load libraries -->
<!-- Polyfill(s) for older browsers -->
<script src="../../node_modules/core-js/client/shim.min.js"></script>
<script src="../../node_modules/zone.js/dist/zone.js"></script>
<script src="../../node_modules/reflect-metadata/Reflect.js"></script>
<script src="../../node_modules/systemjs/dist/system.src.js"></script>
<!-- 2. Configure SystemJS -->
<script src="../systemjs.config.js"></script>
<script>
    System.config({
        "defaultJSExtensions": true
    });
```

```
    System.import('startup').catch(function (err) { console.error(err);
});
</script>
<body>
    <customer-ui></customer-ui>
</body>
</html>
```

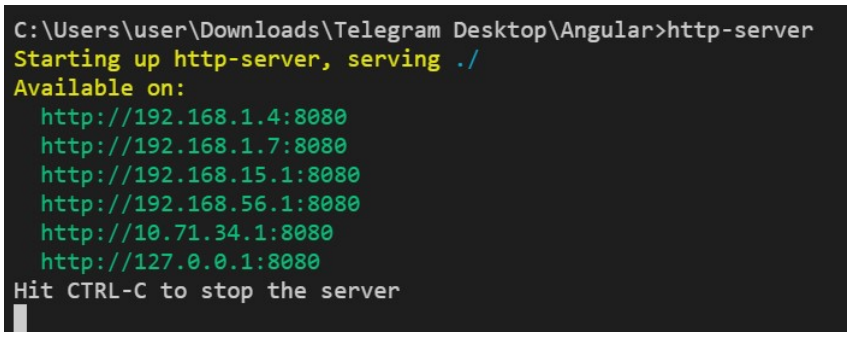
Step 11:- Setting up the typescript compiler

We also need to run typescript so that TS files gets compiled to JS files. So open the integrated terminal and type “tsc -w”. This will make typescript run continuously at the background. So as you type typescript will keep compiling when the file changes.

```
PS E:\Angular> tsc -w
11:05:08 PM - Compilation complete. Watching for file changes.


11:05:12 PM - File change detected. Starting incremental compila
tion...


11:05:13 PM - Compilation complete. Watching for file changes.
```

Step 12: - Running the http-server

So if everything is ok in Step 11 we now need to now run http server. To run server we need to type “http-server” in the VS code integrated terminal as shown in the figure.



In case your 80 port is blocked you can run this server on a specific port using command

```
"http-server -p 99"
```

This will run this server over 99 port.

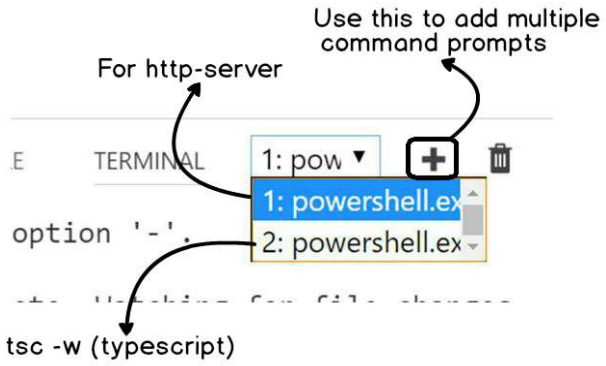
So once the web server is running you can now browse to the main angular HTML page.

Main angular page means the page in which we have put the scripts, put the place holder, SystemJS and so on.

Please note Customer.html is not the main page. This page will be loaded in the placeholder of main angular page.

Once the sites are running type in one of the textboxes and see the automation of binding output in expression.

Run in two terminals. You can open two terminals by clicking on the "+" in terminal window. So in one window you can run http-server and in other window you can run typescript in watch mode.



How to the run the source code?

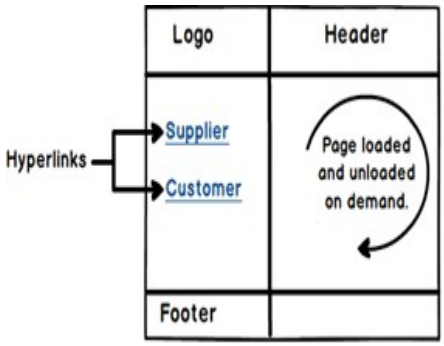
The source code that is attached in this book is without “node_modules” folder. So to run the code you need to open the folder using VS code and then do a NPM using the integrated terminal on the folder where you have “package.json” file.

Lab 7:- Implementing SPA using Angular routing

Fundamental of Single Page Application (SPA)

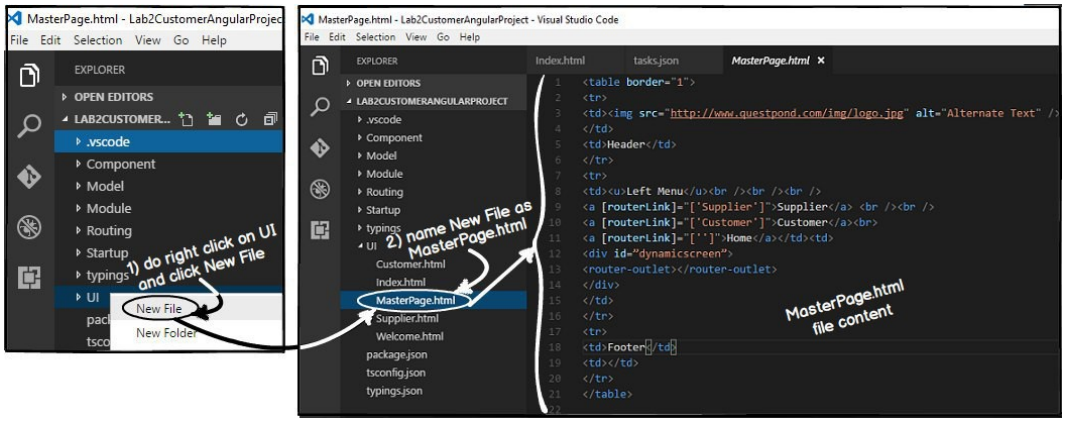
Now a day’s Single page application (SPA) has become the style of creating websites. In SPA we load only things which we need and nothing more than that.

At the right-hand side is a simple website where we have Logo and header at the top, Left menu links and footer at the bottom. So the first time when user comes to the site all the sections of the master page will be loaded.
But when user clicks on Supplier link only Supplier page will load and not logo, header and footer again.
When user clicks on Customer link only Customer page will be loaded and not all other sections.
Angular routing helps us to achieve the same.

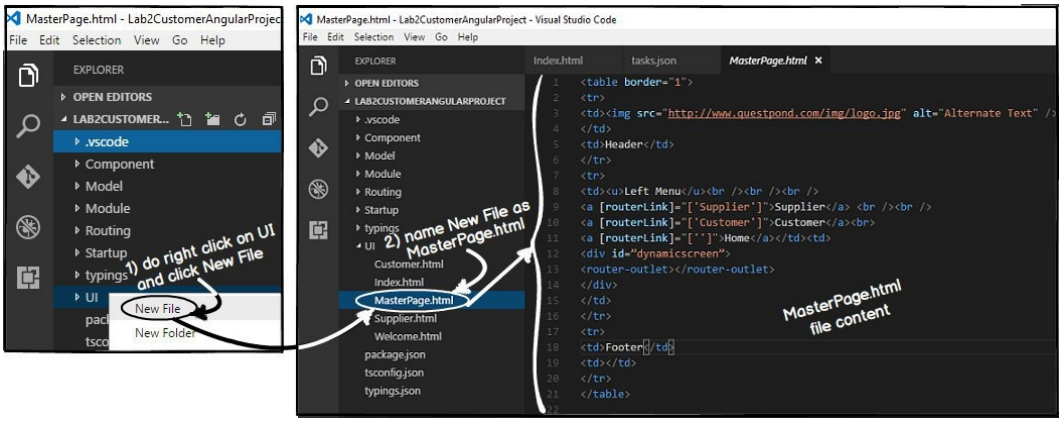


Step 1: - Creating the Master Page

As everything revolves around Master Page so first logical step is to create the “MasterPage”.



In this master page we will create placeholders for logo, header , menu , footer , copyright and so on. These sections will be loaded only once when the user browses the website first time. And in the later times only pages which are needed will be loaded on demand. Below is the sample HTML code which has all the placeholder sections. You can also see in this code we have kept a “DIV” tag section in which we would loading the pages on-demand.



Below is the overall main sections of “MasterPage”. Please note the “DIV” section with name “dynamicscreen” where we intend to load screens dynamically. We will fill these sections later.

```
<table border="1">
<tr>
<td>Logo</td>
<td>Header</td>
</tr>
<tr>
<td>Left Menu</td>
<td>
<div id=”dynamicscreen”>
  Dynamic screen will be loaded here
</div>
</td></tr>
<tr>
<td>Footer</td>
<td>Copyright</td></tr></table>
```

Step 2:- Creating the Supplier page and welcome page

Let's create two more HTML UI one "Supplier" page and "Welcome" page. In both these HTML pages we are not doing much, we have just greeting messages.

Below is the supplier pages text.

```
This is the Suppliers page
```

Below is welcome pages text.

```
Welcome to the website
```

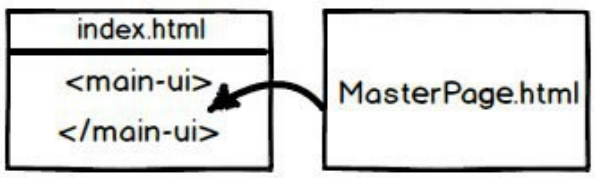
Step 3:- Renaming placeholder in Index.html

As explained in Part 1 "Index.html" is the startup page and it bootstraps all other pages using systemjs. In the previous lesson inside "Index.html", "Customer.html" page was loading. But now that we have master page so inside index page "MasterPage.html" will load.

So to make it more meaningful lets rename "customer-ui" tag to "main-ui". In this "main-ui" section we will load the master page and when end user clicks on the master page left menu links supplier, customer and welcome pages will load.

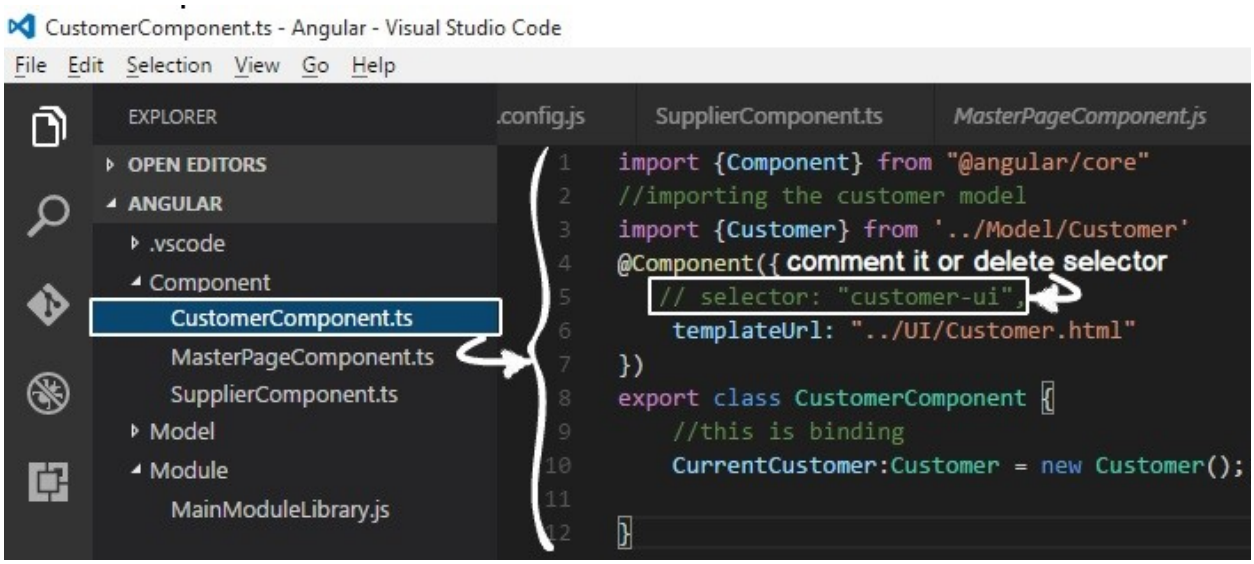
```
<body>
    <main-ui></main-ui>
</body>
```

So if look at the flow, first index.html will get loaded and then inside the "main-ui", "masterpage.html" gets loaded.



Step 4:- Removing selector from CustomerComponent

Now the first page to load in the index.html will be MasterPage and not Customer page. So we need to remove the selector from “CustomerComponent.ts”. This selector will be moved to MasterPage component in the later sections.

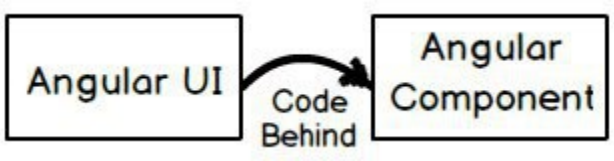


The final code of “CustomerComponent.ts” would look something as show below.

```
import {Component} from "@angular/core"
//importing the customer model
import {Customer} from '../Model/Customer'
@Component({
    templateUrl: "../UI/Customer.html"
})
export class CustomerComponent {
    //this is binding
    CurrentCustomer:Customer = new Customer();

}
```

Step 5:- Creating Components for Master, Supplier and Welcome page

Every UI which is Angular enabled should have component code file. We have created 3 user interfaces so we need three component code files for the same.



In the component folder, we will create three component TS files "MasterPageComponent.ts", "SupplierComponent.ts" and "WelcomeComponent.ts".

You can visualize component code files as code behind for Angular UI.

So first let us start with our "MasterPage.html" component which we have given the name it as "MasterPageComponent.ts". This master page will get loaded in "Index.html" page in the initial bootstrapping process. You can see in this component we have put the selector and this will be the only component which will have the selector.

```
import {Component} from "@angular/core"

@Component({
    selector: "main-ui",
    templateUrl: "../UI/MasterPage.html"
})
export class MasterPageComponent {
}
```

Below is the component code for "Supplier.html".

```
import {Component} from "@angular/core"
@Component({
    templateUrl: "../UI/Supplier.html"
})
export class SupplierComponent {
}
```

Below is the component code for “Welcome.html”. Both Supplier and Welcome component do not have the selector, only the master page component has it as it will be the startup UI which will get loaded in index page.

```
import {Component} from "@angular/core"


@Component({
    templateUrl: "../UI/Welcome.html"
})
export class WelcomeComponent {
}
```

Step 6: - Creating the routing constant collection

Once the master page is loaded in the index page, end user will click on the master page links to browse to supplier page, customer page and so on. Now in order that the user can browse properly we need to define the navigation paths. These paths will be specified in the “href” tags in the later steps. When these paths will be browsed, it will invoke the components and components will load the UI. Below is a simple table with three columns. The first column specifies the path pattern, second which component will invoke when these paths are browsed and the final column specifies the UI which will be loaded.

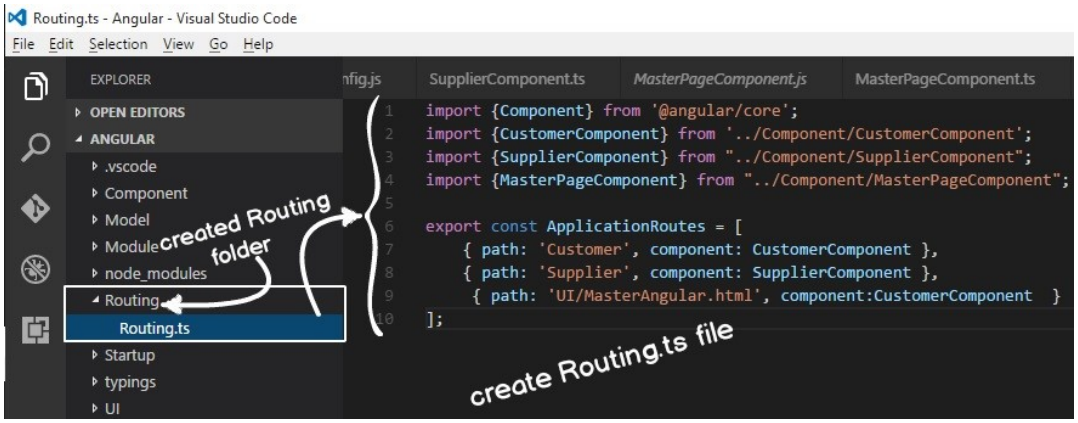
Path/URL	**Component**	**UI which will be loaded**
/	WelcomeComponent.ts	Welcome.html
/Customer	CustomerComponent.ts	Customer.html
/Supplier	SupplierComponent.ts	Supplier.html

The paths and component entries needs to be defined in a simple literal collection as shown in the below code. You can see the “ApplicationRoutes” is a simple collection where we have defined path and the component which will be invoked. These entries are made as per the table specified at the top.

```
import {Component} from '@angular/core';
import {CustomerComponent} from '../Component/CustomerComponent';
import {SupplierComponent} from "../Component/SupplierComponent";
import {WelcomeComponent} from "../Component/WelcomeComponent";
```

```
export const ApplicationRoutes = [
    { path: 'Customer', component: CustomerComponent },
    { path: 'Supplier', component: SupplierComponent },
     { path: '', component:WelcomeComponent  }
];
```

As a good practice above code we have it in a separate folder “routing” & in a separate file “routing.ts”.



Step 7: - Defining routerLink and router-outlet

The navigation (routes) is defined in “Step 6” in the collection needs to be referred when we try to navigate inside the website. For example, in the master page we have defined the left menu hyperlinks. So rather than using the “href” tag of HTML we need to use “[routerLink]”.

```
<a href=”Supplier.html”>Supplier</a>
```

We need to use “[routerLink]” and the value of “[routerLink]” will be the paths specified in the route collection define the previous step. For example in the “ApplicationRoutes” collection we have made one entry for Supplier path we need to specify the path in the anchor tag as shown in the below code.

```
<a import {Component} from '@angular/core';
import {CustomerComponent} from '../Component/CustomerComponent';
import {SupplierComponent} from "../Component/SupplierComponent";
```

```
import {WelcomeComponent} from "../Component/WelcomeComponent";

export const ApplicationRoutes = [
    { path: 'Customer', component: CustomerComponent },
    { path: 'Supplier', component: SupplierComponent },
     { path: '', component:WelcomeComponent  }
];
]">Supplier</a>
```

When the end user clicks on the left master page links the pages (supplier page, customer page and welcome page) will get loaded inside the “div” tag. For that we need to define “router-outlet” placeholder. Inside this placeholder pages will load and unload dynamically.

```
<div id=”dynamicscreen”>
<router-outlet></router-outlet>
</div>
```

So if we update the master page defined in “Step 1” with “router-link” and “router-outlet” we would end up with code something as shown below.

```
<table border="1">
<tr>
<td><img src="http://www.questpond.com/img/logo.jpg" alt="Alternate
Text" />
</td>
<td>Header</td>
</tr>
<tr>
<td><u>Left Menu</u><br /><br /><br />
<a [routerLink]="['Supplier']">Supplier</a> <br /><br />
<a [routerLink]="['Customer']">Customer</a></td><td>
<div id=”dynamicscreen”>
<router-outlet></router-outlet>
```

```
</div>
</td>
</tr>
<tr>
<td>Footer</td>
<td></td>
</tr>
</table>
```

Step 8:- Loading the routing in Main modules

In order to enable routing collection paths defined in “ApplicationRoutes” we need to load that in the “MainModuleLibrary” as shown in the below code. “RouterModule.forRoot” helps load the application routes at the module level.

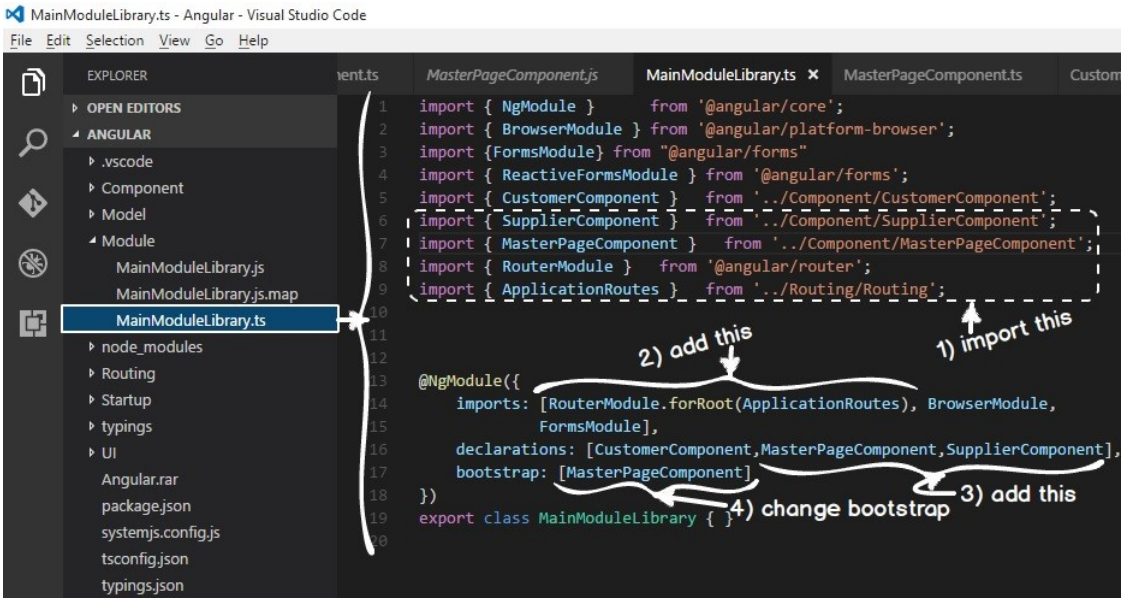
Once loaded at the module level it will be available to all the components for navigation purpose which is loaded inside this module.

```
@NgModule({
    imports: [RouterModule.forRoot(ApplicationRoutes),
              BrowserModule,
              FormsModule],
    declarations:
[CustomerComponent,MasterPageComponent,SupplierComponent],
    bootstrap: [MasterPageComponent]
})
export class MainModuleLibrary { }
```

The complete code with routes would look something as shown below.

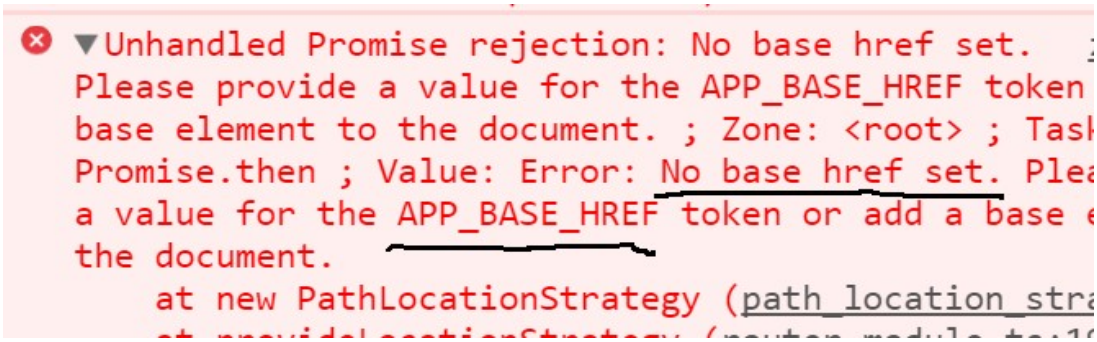
```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import {FormsModule} from "@angular/forms"
import { CustomerComponent }   from '../Component/CustomerComponent';
import { SupplierComponent }   from '../Component/SupplierComponent';
```

```
import { MasterPageComponent }   from
'../Component/MasterPageComponent';
import { RouterModule }   from '@angular/router';
import { ApplicationRoutes }   from '../Routing/Routing';

@NgModule({
    imports: [RouterModule.forRoot(ApplicationRoutes),
             BrowserModule,
             FormsModule],
    declarations:
[CustomerComponent,MasterPageComponent,SupplierComponent],
    bootstrap: [MasterPageComponent]
})
export class MainModuleLibrary { }
```



Step 9: - Define APP BASE HREF

Router module requires a root path. In other words “Supplier” route would become “companyname/Supplier”, “Customer” route would become “companyname/Customer” and so on. If you do not provide a route you would end up with error as shown below.

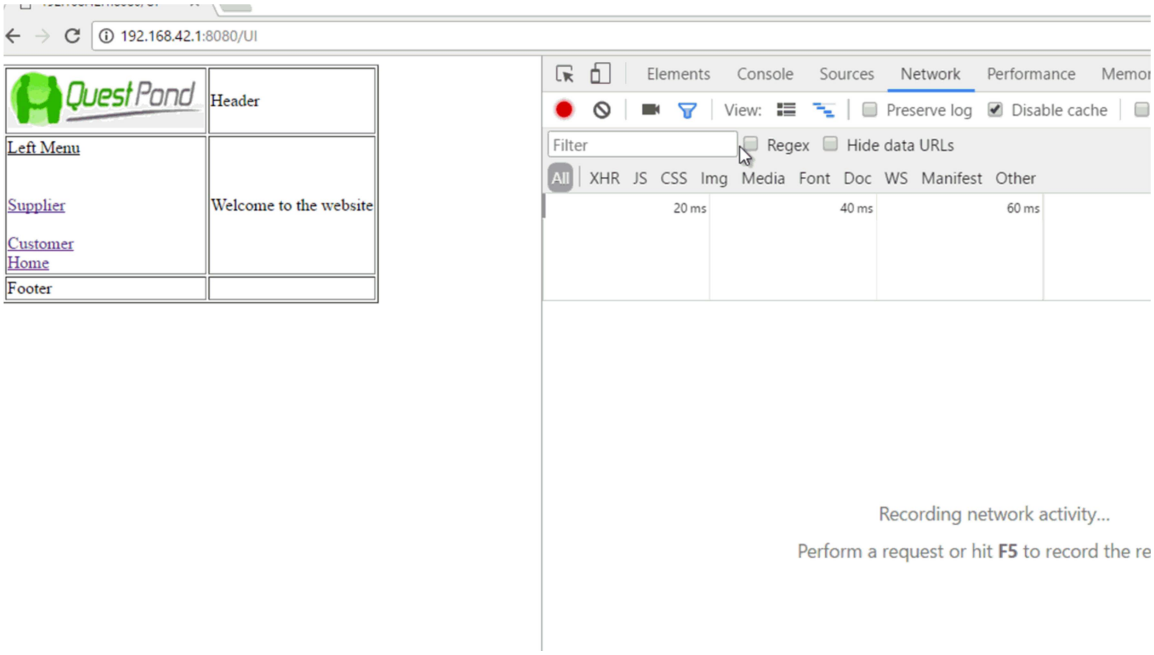


So in your “Index.html” we need add the HTML BASE HREF tag as show in the highlighted code below. At this moment we are not providing any root directory.

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
    <meta charset="utf-8" />
</head>
<base href="./">
<!—Other code has been removed for clarity -->
```

Step 10:- Seeing the output

Now run the website and try to browser to UI folder and you should see the below animated video output. You can see that the logo gets loaded only once and later when the user is click on the supplier links, customer links image is not loading again and again.



Step 11:- Fixing cannot match any routes error

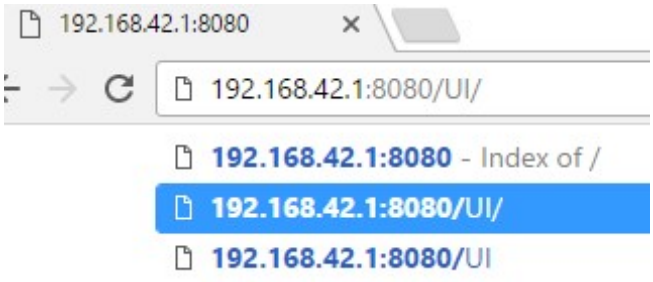
If you do a “F12” and check in the console part of the chrome browser, you would see the below error. Can you guess what the error is?

```
    at zone.js:502
    at ZoneDelegate.invokeTask (zone.js:265)
    at Object.onInvokeTask (core.umd.js:6233)
    at ZoneDelegate.invokeTask (zone.js:264)
▶Unhandled Promise rejection: Cannot match any routes: 'UI' ; Zone: angular
; Task: Promise.then ; Value: Error: Cannot match any routes: 'UI'
    at ApplyRedirects.noMatchError (router.umd.js:769)
    at CatchSubscriber.eval [as selector] (router.umd.js:747)
    at CatchSubscriber.error (catch.ts:58)
```

Your current angular application is route enabled. So every URL which is browsed is looked up in to routes collection.

So the first URL which you browse is "/UI" and it tries to lookup in to your routes collection and does not find one.

So that's why it throws up the above error.

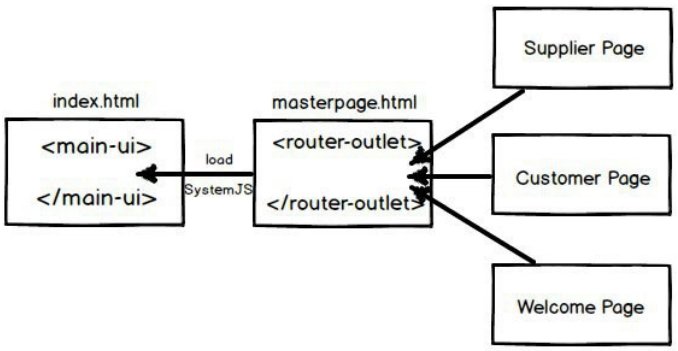


In order to fix the same, make one more entry for "UI" path & point it to "WelcomeComponent".

```
export const ApplicationRoutes = [
    { path: 'Customer', component: CustomerComponent },
    { path: 'Supplier', component: SupplierComponent },
     { path: '', component:WelcomeComponent  },
    { path: 'UI', component:WelcomeComponent  }

];
```

Understanding the flow

1. End user loads "index.html" page.
2. Index.html triggers SystemJS and loads masterpage.html.
3. When end users click on MasterPage links, other pages are loaded in "router-outlet" placeholders.



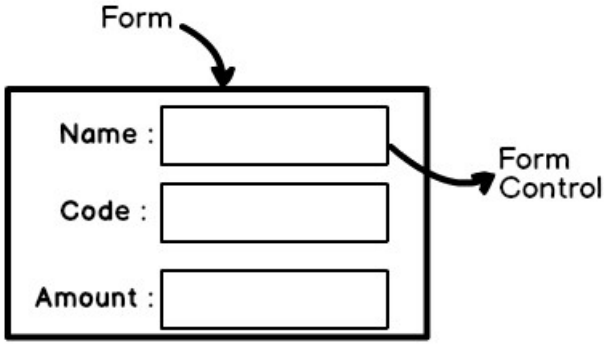
Lab 8:- Implementing validation using Angular form

In this lab we will try to understand how we can implement validations using Angular framework.

Requirement of a good validation structure

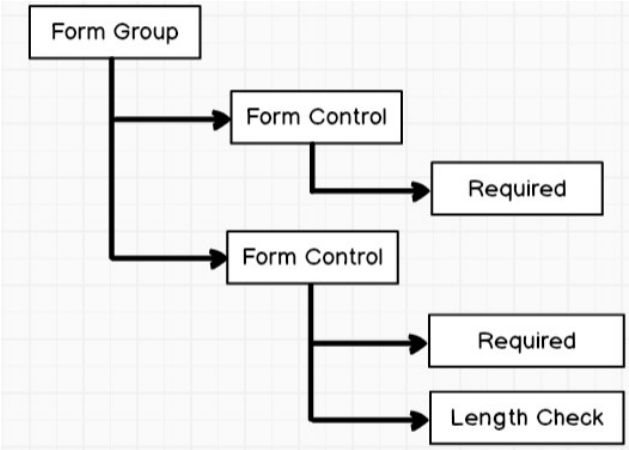
One of the important legs of software application is validations.

Validations are normally applied to user interfaces (Forms) and User interfaces have controls and on the controls, we apply validations.



In Angular form validation architecture structure is as follows.

At the top we have FormGroup, FormGroup has FormControls and FormControls has one or many validations.



There are 3 broader steps to implement Angular validation

1. Create FormGroup.
2. Create FormControl and with proper validations.
3. Map those validations to the HTML Form.

What kind of validation will we implement in this Lab?

In this lab we will implement the following validation on our Customer screen: -

- Customer Name should be compulsory.
- Customer code should be compulsory.
- Customer code should be in the format of A1001, B4004 and so on. In other words the first character should be a capital alphabet followed by 4 letter numeric.

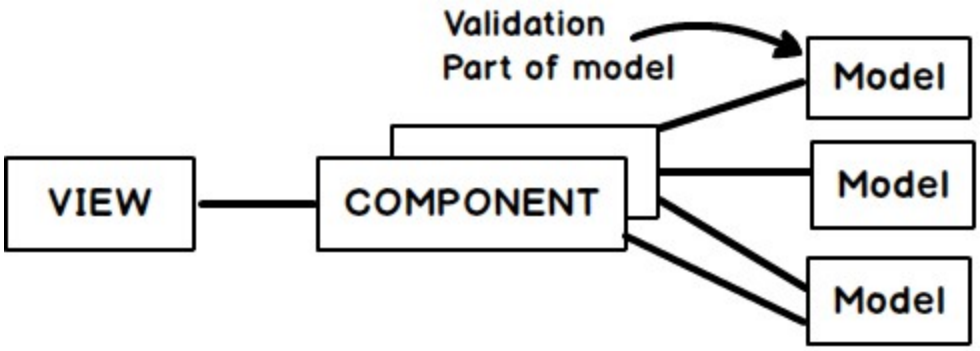
Note: - Customer code has composite validations.

Where to put Validations?

Even before we start with validations we need to decide right place to put validations. If you see in Angular we have three broader sections UI, Component and Model. So let's think over which is the right layer to put validations.

- **UI: -** UI is all about look and feel and positioning of controls. So putting validation in this layer is a bad idea. Yes on this layer we will apply validations but the validation code should be in some other layer.
- **Component: -** Component is the code behind (Binding code) which binds the UI and Model. In this layer more binding related activity should be put. One component can use many models so if we put the validation logic here we need to duplicate that in other components as well.
- **Model: -** A Model represents a real world entity like person, user, admin, product etc. Behavior a.k.a validations are part of the model. A model has Data and Behavior. So the right place where validations should be present is this Layer.

So let us put validation as a part of the model.



Step 1: - Import necessary components for Angular validators

So the first step is to import the necessary components for angular validators in the customer model. All angular validator components are present in "@angular/forms" folder. We need to import five components NgForm, FormGroup, FormControl and validators.

NgForm: - Angular tags for validation.
FormGroup: - Helps us to create collection of validation.
FormControl and Validators: - Helps us to create individual validation inside FormGroup.
FormBuilder: - Helps us to create the structure of Form group and Form controls. Please note one Form group can have many FormControls.

```
import {NgForm,
    FormGroup,
    FormControl,
    Validators,
    FormBuilder } from '@angular/forms'
```

Step 2: - Create FormGroup using FormBuilder

The first step is to create an object of FormGroup in which we will have collection of validation. The FormGroup object will be constructed using the "FormBuilder" class.

```
formGroup: FormGroup = null; // Create object of FormGroup
var _builder = new FormBuilder();
this.formGroup = _builder.group({}); // Use the builder to create
object
```

Step 3: - Adding a simple validation

Once the FormGroup object is created the next step is to add controls in the FormGroup collection. To add a control we need to use "addControl" method. The first parameter in "addControl" method is the name of the validation and second is the Angular validator type to be added.

Below is a simple code in which we are adding a “CustomerNameControl” using the “Validators.required” FormControl. Please note “CustomerNameControl” is not a reserved keyword. It can be any name like “CustControl”.

```
this.formGroup.addControl('CustomerNameControl', new
            FormControl('',Validators.required));
```

Step 4: - Adding a composite validation

If you want to create a composite validation then you need to create a collection and add it using “compose” method as shown in the below code.

```
var validationcollection = [];
validationcollection.push(Validators.required);
validationcollection.push(Validators.pattern("^[A-Z]{1,1}[0-
9]{4,4}$"));
this.formGroup.addControl('CustomerCodeControl', new FormControl('',
Validators.compose(validationcollection)));
```

Full Model code with validation

Below is the full Customer Model code with all three validations as discussed in the previous section. We have also commented the code so that you can follow it.

```
// import components from angular/form
import {NgForm,
    FormGroup,
    FormControl,
    Validators,
    FormBuilder } from '@angular/forms'
export class Customer {
    CustomerName: string = "";
    CustomerCode: string = "";
    CustomerAmount: number = 0;
    // create object of form group
```

```
    formGroup: FormGroup = null;

    constructor(){
        // use the builder to create the
        // the form object
        var _builder = new FormBuilder();
        this.formGroup = _builder.group({});

        // Adding a simple validation
        this.formGroup.addControl('CustomerNameControl', new
            FormControl('',Validators.required));

        // Adding a composite validation
        var validationcollection = [];
        validationcollection.push(Validators.required);
        validationcollection.push(Validators.pattern("^[A-Z]{1,1}[0-
9]{4,4}$"));
        this.formGroup.addControl('CustomerCodeControl', new
            FormControl('', Validators.compose(validationcollection)));
    }
}
```

Step 5: - Reference "ReactiveFormsModule" in MainModuleLibrary.

```
import {FormsModule , ReactiveFormsModule} from "@angular/forms"
@NgModule({
    imports: [RouterModule.forRoot(ApplicationRoutes),
              BrowserModule,
              FormsModule,
              ReactiveFormsModule],
    declarations: [CustomerComponent,
                   MasterPageComponent,
```

```
                    SupplierComponent,
                WelcomeComponent],
    bootstrap: [MasterPageComponent]
})
export class MainModuleLibrary { }
```

Step 6: - Apply formGroup to HTML form

The next thing is to apply 'formGroup' object to the HTML form. For that we need to use "[formGroup]" angular tag in that we need to specify the "formGroup" object exposed via the customer object.

```
<form [formGroup]="CurrentCustomer.formGroup">
</form>
```

Step 7: - Apply validations to HTML control

The next step is to apply formgroup validation to the HTML input controls. That's done by using "formControlName" angular attribute. In "formControlName" we need to provide the form control name which we have created while creating the validation.

```
<input type="text" formControlName="CustomerNameControl"
[(ngModel)]="CurrentCustomer.CustomerName"><br /><br />
```

Step 8: - Check if Validations are ok

When user starts filling data and fulfilling the validations we would like to check if all validations are fine and accordingly show error message or enable/disable UI controls.

In order to check if all validations are fine we need to use the "valid" property of "formGroup". Below is a simple example where the button will be disabled depending on whether validations are valid or not. "[disabled]" is an angular attribute which enables and disables HTML controls.

```
<input type="button"
```

```
value="Click" [disabled]="!(CurrentCustomer.formGroup.valid)"/>
```

Step 9:- Checking individual validations

"CurrentCustomer.formGroup.valid" checks all the validations of the "FormGroup" but what if we want to check individual validation of a control.

For that we need to use "hasError" function.
"CurrentCustomer.formGroup.controls['CustomerNameControl'].hasError('required')" checks that for "CustomerNameControl" has the "required" validation rule been fulfilled. Below is a simple code where we are displaying error message in a "div" tag which visible and not visible depending on whether the "hasError" function returns true or false.

Also note the "!" (NOT) before "hasError" which says that if "hasError" is true then hidden should be false and vice versa.

```
<div
[hidden]="!(CurrentCustomer.formGroup.controls['CustomerNameControl'].hasError('required'))">Customer name is required </div>
```

Step 10: - standalone elements

In our forms we have three textboxes "CustomerName" , "CustomerCode" and "CustomerAmount". In these three textboxes only "CustomerName" and "CustomerCode" has validations while "CustomerAmount" does not have validations.
Now this is bit funny but if we do not specify validations for a usercontrol which is inside a "form" tag which has "formGroup" specified you would end up with a long exception as shown below.

```
Error: Uncaught (in promise): Error: Error in ../UI/Customer.html:15:0
caused by:
      ngModel cannot be used to register form controls with a parent
formGroup directive.  Try using
      formGroup's partner directive "formControlName" instead.
Example:


    <div [formGroup]="myGroup">
```

```
      <input formControlName="firstName">
    </div>

    In your class:

    this.myGroup = new FormGroup({
       firstName: new FormControl()
    });

      Or, if you'd like to avoid registering this form control,
indicate that it's standalone in ngModelOptions:

      Example:


    <div [formGroup]="myGroup">
       <input formControlName="firstName">
       <input [(ngModel)]="showMoreControls"
[ngModelOptions]="{standalone: true}">
    </div>

Error:
      ngModel cannot be used to register form controls with a parent
formGroup directive.  Try using
      formGroup's partner directive "formControlName" instead.
Example:


    <div [formGroup]="myGroup">
      <input formControlName="firstName">
    </div>
```

```
    In your class:

    this.myGroup = new FormGroup({
       firstName: new FormControl()
    });

      Or, if you'd like to avoid registering this form control,
indicate that it's standalone in ngModelOptions:

      Example:
    <div [formGroup]="myGroup">
       <input formControlName="firstName">
       <input [(ngModel)]="showMoreControls"
[ngModelOptions]="{standalone: true}">
    </div>
```

The above error can be simplified in three simple points: -

1. It says that you have enclosed a HTML control inside a HTML FORM tag which has Angular form validations.
2. All controls specified inside HTML FORM tag which have angular validation applied SHOULD HAVE VALIDATIONS.
3. If a HTML control inside Angular form validation does not have validation you can do one of the below things to remove the exception: -
    - You need to specify that it's a standalone control.
    - Move the control outside the HTML FORM tag.

Below is the code how to specify "standalone" for Angular validations.

```
<input type="text" [ngModelOptions]="{standalone:true}"
[(ngModel)]="CurrentCustomer.CustomerAmount"><br /><br />
```

Also talk about we can remove from the form control and what happens

Complete code of Customer UI with validations applied

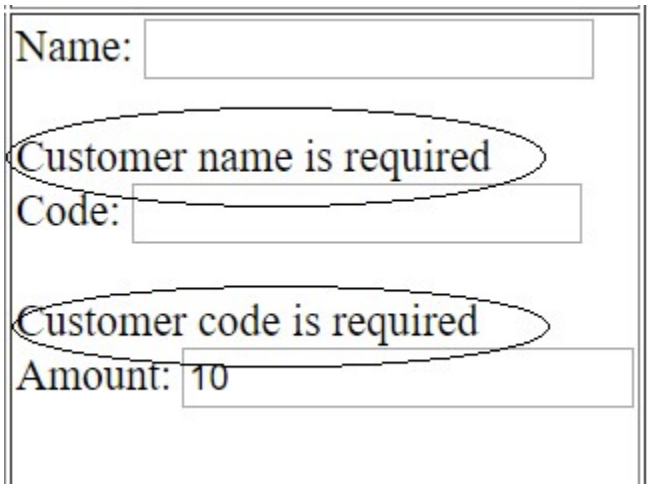
Below is the complete Customer UI with all three validations applied to “CustomerName” and “CustomerCode” controls.

```
<form [formGroup]="CurrentCustomer.formGroup">
<div>
Name:
<input type="text" formControlName="CustomerNameControl"
[(ngModel)]="CurrentCustomer.CustomerName"><br /><br />
<div
[hidden]="!(CurrentCustomer.formGroup.controls['CustomerNameControl'].h
asError('required'))">Customer name is required </div>
Code:
<input type="text" formControlName="CustomerCodeControl"
[(ngModel)]="CurrentCustomer.CustomerCode"><br /><br />
<div
[hidden]="!(CurrentCustomer.formGroup.controls['CustomerCodeControl'].h
asError('required'))">Customer code is required </div>
<div
[hidden]="!(CurrentCustomer.formGroup.controls['CustomerCodeControl'].h
asError('pattern'))">Pattern not proper </div>

Amount:
<input type="text"
[(ngModel)]="CurrentCustomer.CustomerAmount"
[ngModelOptions]="{standalone: true}"><br /><br />
</div>
{{CurrentCustomer.CustomerName}}<br /><br />
{{CurrentCustomer.CustomerCode}}<br /><br />
{{CurrentCustomer.CustomerAmount}}<br /><br />
<input type="button"
value="Click" [disabled]="!(CurrentCustomer.formGroup.valid)"/>
</form>
```

Write reactive forms

Run and see your validation in action

Once you are done you should be able to see the validation in action as shown in below figure.



Dirty, pristine, touched and untouched

In this lab we covered “valid” and “hasError” property and function. “formGroup” also has lot of other properties which you will need when you are working with validations. Below are some important ones.

Property	Explanation
dirty	This property signals if Value has been modified.
pristine	This property says if the field has changed or not.
touched	When the lost focus for that control occurs.
untouched	The field is not touched.

Lab 9: - Making HTTP calls

Importance of server side interaction

HTML user interfaces are dead if there is no server side interaction. Normally in a web application we would like to send the data entered by end user to the server. On server side we

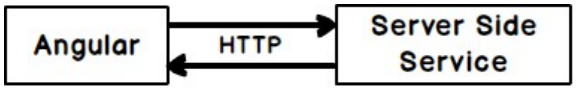
would have some kind of service which can be created in technologies like Java, C# and so on. The server side technology can save, edit or delete this data in a database.

In simple words we need to understand how to make HTTP calls from Angular code to a server side technology.

Yes, this is a pure Angular book

I intend to keep this book as a pure Angular book. So teaching server side technology like ASP.NET MVC, Java services, PHP is out of scope.

So the server side service would be FAKED (Mocked) by using "Angular inmemory Webapi". Please note this is not a service which you will use for production its only for test and development purpose.



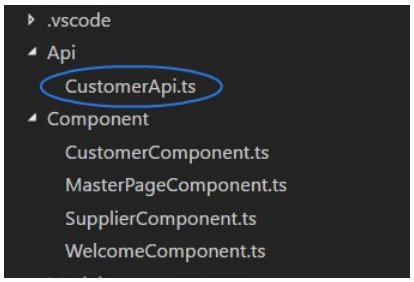
In case you want to learn ASP.NET MVC you can start from this video https://www.youtube.com/watch?v=Lp7nSImO5vk

Step 1: - Creating a fake server side service

Even though we have decided that we will not be creating a professional server side service using ASP.NET, Java etc. but we will still need one. So we will be using a fake service called as "angular-in-memory-web-api". Already the "angular-in-memory-web-api" has been installed when we did npm. You can check your "package.json" file for the entry of "angular-in-memory-web-api".

So let's create a folder called as "Api" and in that let us add "CustomerApi.ts" file and in this file we will write code which will create a fake customer service.

On this fake service we will make HTTP calls.

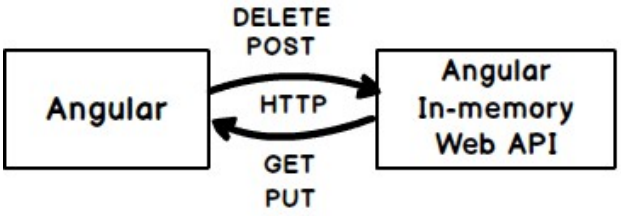


Below is a simple service created using "angular-in-memory" open source. In this service we have loaded a simple "customers" collection with some sample customer records.

```
import { InMemoryDbService } from 'angular-in-memory-web-api'
import {Customer} from "../Model/Customer"
export class CustomerApiService implements InMemoryDbService {
  createDb() {
    let customers =[
      { CustomerCode: '1001', CustomerName: 'Shiv' , CustomerAmount
:100.23 },
      { CustomerCode: '1002', CustomerName: 'Shiv1' , CustomerAmount
:1.23 },
      { CustomerCode: '1003', CustomerName: 'Shiv2' , CustomerAmount
:10.23 },
      { CustomerCode: '1004', CustomerName: 'Shiv3' , CustomerAmount
:700.23 }
    ]
    return {customers};
  }
}
```

So now when angular makes HTTP call it will hit this in-memory API.

So when you make a HTTP GET it will return the above four records. When you make a HTTP POST it will add the data to the in-memory collection.



In other words we do not need to create a professional server side service using ASP.NET or Java service.

Step 2: - Importing HTTP and WebAPI module in to main module

The next step is to import “HttpModule” from “angular/http” and in-memory API in to main module. Remember module is collection of components. So the “MainModule” has “CustomerComponent”, “SupplierComponent”, “MasterpageComponent” and “WelcomeComponent”.

```
import { HttpModule} from '@angular/http';
import {CustomerApiService} from "../Api/CustomerApi"
```

Also in the “NgModule” attribute in imports we need to specify “HttpModule” and “InMemoryWebApiModule”.

```
@NgModule({
    imports: [….
        HttpModule,
        InMemoryWebApiModule.forRoot(CustomerApiService),
            …..],
    declarations: […],
    bootstrap: [….]
})
export class MainModuleLibrary { }
```

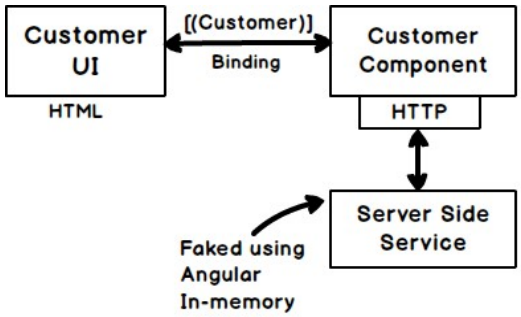
Shiv: - Talk about the sequence of the Httpmodule and Angular WebAPI

Where do we put the HTTP call?

The next question comes which is the right place to put HTTP calls?

So if you see the normal architecture of Angular it is as follows: -

- User interface is binded with the Angular component using bindings “[()]”.
- So once end user starts putting data in the UI the model object (in our case it’s the customer object) will be loaded and sent to the Customer component.
- Now customer component has the filled customer object. So the right place to make HTTP call is in the component.



Step 3: - Importing HTTP in the Customer component

So let's go ahead and import the Angular HTTP inside the Customer component.

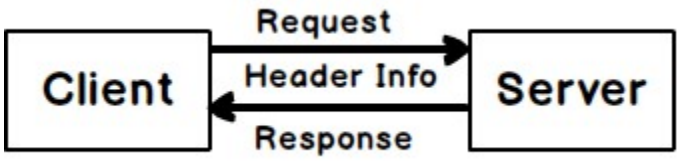
```
import { Http, Response, Headers, RequestOptions } from
'@angular/http';
```

We do not need to create object of HTTP using the new keyword, it will be dependency injected via the constructor. On the constructor component we have defined the object injection of HTTP.

```
constructor(public http:Http){
    }
```

Step 4: - Creating header and request information

When we send HTTP request from client to server or we get response, header information is passed with the request and response. In header information we have things like content types, status, user agent and so on.



So to create a request we need to first create a header and using that header create a request. One of the header information we need to provide is type of content we are sending to server is it XML, JSON etc.

Below is how to create simple request using basic header information.

```
let headers = new Headers({'Content-Type': 'application/json'});
let options = new RequestOptions({ headers: headers });
```

Step 5: - Making HTTP calls and observables

Angular HTTP uses something called as observables. So angular is an observer and it subscribes to observable like HTTP response. In other words, it is listening to data coming from the server.

So the below code says that we will be making a GET call to “api/customers” URL and when the data comes we will send the successful data to the “Success” function and when error occurs we will get it in “Error” function.

```
var observable = this.http.get("api/customers",  options);
        observable.subscribe(res => this.Success(res),
            res => this.Error(res));
```

Below is the code of Error and Success function. In “Success” function we are converting the response to JSON and assigning it to “Customers” collection which is in the component. If we have error we are displaying it in the browser console.

```
Error(err) {
        console.debug(err.json());
    }
    Success(res) {
        this.Customers = res.json().data;
    }
```

Step 6: - Creating a simple post and get call

With all that wisdom we have gathered from Step 4 and Step 5 lets write down two functions one which will display data and the other which will post data.

Below is a simple “Display” function which makes a HTTP GET call.

```
Display(){
        let headers = new Headers({
            'Content-Type': 'application/json'
        });
        let options = new RequestOptions({ headers: headers });
        var observable = this.http.get("api/customers",  options);
        observable.subscribe(res => this.Success(res),
            res => this.Error(res));}
```

As soon as the customer UI is loaded the customer component object will be created. So in the constructor we have called the “Display” function to load the customer’s collection.

```
export class CustomerComponent {
// other code removed for clarity
constructor(public http:Http){
            this.Display();
    }
// other codes removed for clarity
}
```

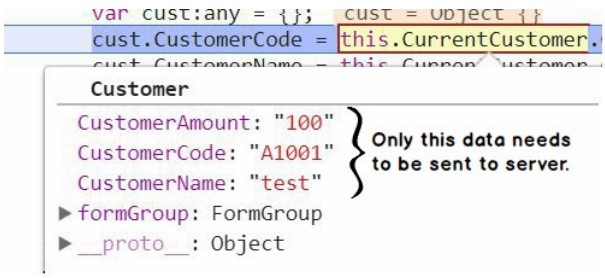
Below is simple “Add” function which makes a POST call to the server. In http POST call code below you can see customer object sent as the third parameter. After the “Add” call we have made call to “Display” so that we can see the new data added on the server.

```
Add(){
           let headers = new Headers({
            'Content-Type': 'application/json'
        });
        var cust:any = {};
        cust.CustomerCode = this.CurrentCustomer.CustomerCode;
        cust.CustomerName = this.CurrentCustomer.CustomerName;
        cust.CustomerAmount = this.CurrentCustomer.CustomerAmount;
        let options = new RequestOptions({ headers: headers });
        var observable = this.http.post("api/customers",cust,
options);
        observable.subscribe(res => this.Success1(res),
            res => this.Error(res));
        this.Display();
    }
```

In the above “Add” function you can see the below code which creates a fresh light weight customer object. So why do we need to create this fresh new object?

```
        var cust:any = {};
        cust.CustomerCode = this.CurrentCustomer.CustomerCode;
        cust.CustomerName = this.CurrentCustomer.CustomerName;
        cust.CustomerAmount = this.CurrentCustomer.CustomerAmount;
```

The current customer object has lot of other things like validations, prototype object etc. So posting this whole object to the server does not make sense. We just want to send three properties "CustomerCode", "CustomerAmount" and "CustomerName".



In other words we need to create a light weight DTO (Data transfer object) which just has those properties.

Step 7: - Connecting components to User interface

Now that our component is completed we need to attach the "Add" function to the button using the "click" event of Angular. You can see that the (click) is inside a round bracket, in other words we are sending something(event click) from UI to the Object.

```
<input type="button"
value="Click" (click)="Add()"
[disabled]="!(CurrentCustomer.formGroup.valid)"/>
```

Also we need to create a table in which we will use "ngFor" loop and display all customer's collection on the HTML UI. In the below code we have created a temporary object "cust" which loops through the "Customers" collection and inside <td> tag we are using the expressions ({{cust.CustomerName}}) to display data.

```
<table>
    <tr>
        <td>Name</td>
         <td>code</td>
          <td>amount</td>
```

```
    </tr>
      <tr *ngFor="let cust of Customers">
        <td>{{cust.CustomerName}}</td>
         <td>{{cust.CustomerCode}}</td>
          <td>{{cust.CustomerAmount}}</td>
    </tr>
</table>
```

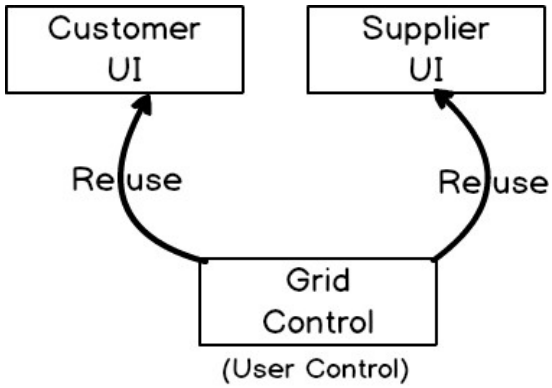
Go ahead and run the application. If you go and add “customer” object you should see the HTTP calls happening and the list getting loaded in the HTML table.

Lab 10: - Input, Output and emitters

Theory

Reusability is one of the most important aspects of development. As Angular is a UI technology we would like to create UI controls and reuse them in different UI.

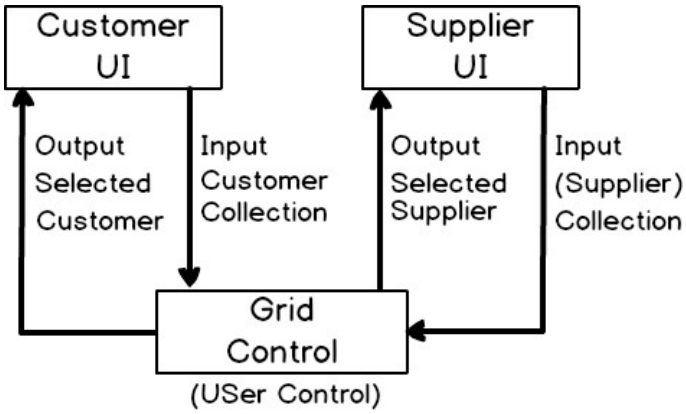
For example in the Customer UI we have the table grid display. If we want to create grid in other UI we need to again repeat the “<tr><td>” loop. It would be great if we can create a generic reusable grid control which can be plugged in to any UI.



If we want to create a GENERIC reusable grid control you need to think GENERICALLY, you need to think in terms of INPUTS and OUTPUTS.

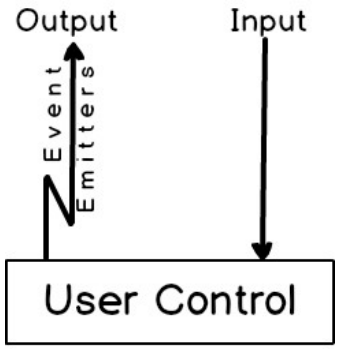
So the first visualization which you should have in your mind is that your GRID control is like a self-contained unit which gets a input of some data collection and when any one selects grid data the selected object is sent outside.

So if you are using this grid control with a Customer UI you get a Customer collection, if you are using a Supplier UI you will get a supplier collection.



In order to achieve this generic thought process Angular has provided three things Input, Output and Event emitters.

Input helps us define the input data to the user control. Output uses event emitters to send data to the UI in which the user control is located.



Planning how the component will look like

First let us plan how our grid component will look like. Grid component will be called in main HTML using “<grid-ui></grid-ui>” HTML element.

The grid component will have three attributes: -

- **grid-columns: -** This will be a input in which we will specify the columns names of the grid.

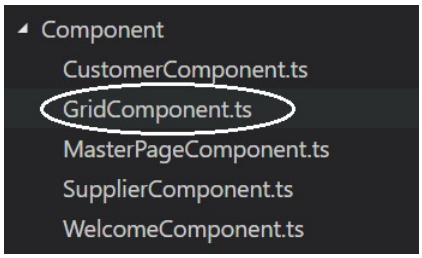
- **grid-data: -** This will be again a input in which we will provide the data to be displayed on the grid.
- **grid-selected: -** This will be a output from which the selected object will be sent via emitter events to the contained UI.

```
<grid-ui  [grid-columns]=”In this we will give column names"
    [grid-data]="In this we will give data for grid"
    (grid-selected)="The selected object will be sent in event">
</grid-ui>
```

Step 1: - Import input, output and Event Emitter

So first let us go ahead and add a separate file for grid component code and name it “GridComponent.ts”.

In this file, we will write all code that we need for Grid component.



The first step is to add necessary components which will bring in Input and Output capabilities. For that we need to import component, Input, Output and event emitter component from “angular/core”.

```
import {Component,
        Input,
        Output,
        EventEmitter} from "@angular/core"
```

Step 2: - Creating the reusable GridComponent class

As this is a component we need to decorate the class with “@Component” decorator. The UI for this component will coded in “Grid.html”. You can also see in the below code we defined the selector has “grid-ui”, can you guess why?

If you remember in the planning phase we had said that the grid can be called by using “<grid-ui>” tag.

```
@Component({
    selector: "grid-ui",
    templateUrl : "../UI/Grid.html"
})
export class GridComponent {
}
```

Step 3: - Defining inputs and output

As this is a grid component we need data for the grid and column names for the grid. So we have created two array collection one, “gridColumns” (which will have column names) and “gridData” (to data for the table).

```
export class GridComponent {
    // This will have columns
    gridColumns: Array<Object> = new Array<Object>();
    // This will have data
    gridData: Array<Object> = new Array<Object>();}
```

There are two methods “setGridColumns” and “setGridDataSet” which will help us to set column names and table data to the above defined two variables.

These methods will be decorated by using “@Input” decorators and in this we will put the names by which these inputs will be invoked while invoking this component.

```
// The top decorator and import code is removed
// for clarity purpose

export class GridComponent {
// code removed for clarity
    @Input("grid-columns")
    set setgridColumns(_gridColumns: Array<Object>) {
            this.gridColumns = _gridColumns;
        }
    @Input("grid-data")
    set setgridDataSet(_gridData: Array<Object>) {
```

```
            this.gridData = _gridData;
        }
}
```

The names defined in the input decorator will be used as shown below while making call to the component in the main UI.

```
<grid-ui  [grid-columns]="In this we will give column names"
    [grid-data]="In this we will give data for grid" >
</grid-ui>
```

Step 4: - Defining Event emitters

As discussed in this Labs theory section we will have inputs and outputs. Outputs are again defined by using "@Output" decorator and data is sent via event emitter object.

To define output we need to use "@Output" decorator as shown in the below code. This decorator is defined over "EventEmitter" object type. You can see that the type is "Object" and not "Customer" or "Supplier" because we want it to be of a generic type. So that we can attach this output with any component type.

```
    @Output("grid-selected")
    selected: EventEmitter<Object> = new EventEmitter<Object>();
```

Now when any end user selects an object from the grid we need to raise event by using the "EventEmitter" object by calling the "emit" method as shown below.

```
// Other codes have been removed for clarity purpose.
export class GridComponent {
    @Output("grid-selected")
    selected: EventEmitter<Object> = new EventEmitter<Object>();

    Select(_selected: Object) {
        this.selected.emit(_selected);
    }
}
```

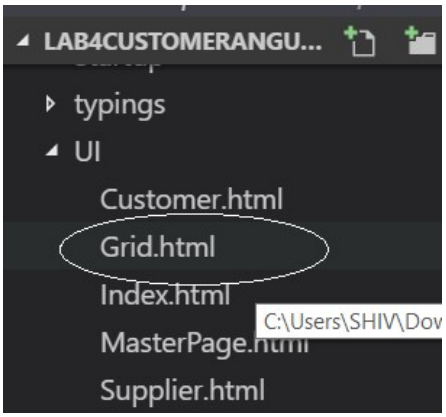
Below goes the full code of “GridComponent.ts” which we have discussed till now.

```
import {Component,
        Input,
        Output,
        EventEmitter} from "@angular/core"

@Component({
    selector: "grid-ui",
    templateUrl : "../UI/Grid.html"
})
export class GridComponent {
    gridColumns: Array<Object> = new Array<Object>();
    // inputs
    gridData: Array<Object> = new Array<Object>();

    @Output("grid-selected")
    selected: EventEmitter<Object> = new EventEmitter<Object>();
    @Input("grid-columns")
    set setgridColumns(_gridColumns: Array<Object>) {
            this.gridColumns = _gridColumns;
        }

    @Input("grid-data")
    set setgridDataSet(_gridData: Array<Object>) {
            this.gridData = _gridData;
        }

    Select(_selected: Object) {
        this.selected.emit(_selected);
    }
}
```

Step 5: - Creating UI for the reusable component

Also we need to create UI for the "GridComponent.ts". Remember if we have an Angular component we NEED A HTML UI for it.

So in the UI folder we will add "Grid.html" in which we will write the code of table display.



In the "GridComponent.ts" (refer Step 4 of this Lab) we have defined input "gridColumns" variable in which we will provide the columns for the grid. So for that we had made a loop using "*ngFor" which will create the columns "<td>" dynamically.

```
<table>
 <tr>
        <td *ngFor="let col of gridColumns">
            {{col.colName}}
        </td>
    </tr>
</table>
```

And to display data in the grid we need to loop through "gridData" variable.

```
    <tr *ngFor="let colObj of gridData">
        <td *ngFor="let col of gridColumns">
            {{colObj[col.colName]}}
        </td>
        <td><a [routerLink]="['Customer/Add']"
(click)="Select(colObj)">Select</a></td> </tr>
```

So the complete code of “Grid.html” looks as shown below.

```
<table>
    <tr>
        <td *ngFor="let col of gridColumns">
            {{col.colName}}
        </td>
    </tr>
    <tr *ngFor="let colObj of gridData">
        <td *ngFor="let col of gridColumns">
            {{colObj[col.colName]}}
        </td>
        <td><a [routerLink]="['Customer/Add']"
(click)="Select(colObj)">Select</a></td>
    </tr>
</table>
```

Step 6: - Consuming the component in the customer UI

So now that our reusable component and its UI is completed, let us call the component in the “Customer.html” UI.

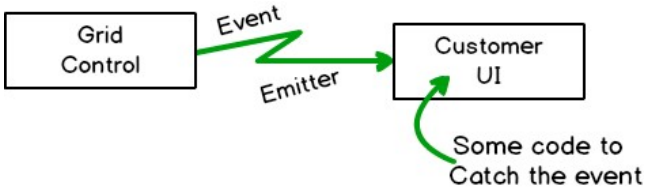
Below is the full proper code which has column names defined in “grid-columns” and in “grid-data” we have set “Customers” collection. This “Customers” collection object is exposed from the “CustomerComponent” class. This “Customers” collection is that collection which we had created during the “HTTP” call lab. This variable has collection of “Customer” data.

```
<grid-ui
      [grid-
columns]="[{'colName':'CustomerCode'},{'colName':'CustomerName'},{'colN
ame':'CustomerAmount'}]"
    [grid-data]="Customers"
    (grid-selected)="Select($event)"></grid-ui>
```

Also we need to ensure that the old “<table>” code is deleted and is replaced with the above “<grid-ui>” input /output tag.

Step 7: - Creating events in the main Customer component

If you remember in our “GridComponent” we are emitting event by calling “emit” method. Below is the code snippet for it. Now this event which is emitted needs to caught in the “CustomerComponent” and processed.




```
export class GridComponent {
    Select(_selected: Object) {
        this.selected.emit(_selected);
    }
}
```

So in order to catch that event in the main component we need to create a method in “CustomerComponent” file. So in the customer component typescript file we will create a “Select” function in which the selected customer will come from the GridComponent and that selected object will be set to “CurrentCustomer” object.

```
export class CustomerComponent {
Select(_selected:Customer) {
        this.CurrentCustomer.CustomerAmount =_selected.CustomerAmount;
        this.CurrentCustomer.CustomerCode = _selected.CustomerCode;
        this.CurrentCustomer.CustomerName = _selected.CustomerName;
    }
}
```

Step 8: - Defining the component in the mainmodule

Also we need to ensure that the “GridComponent” is loaded in the main module. So in the main module import the “GridComponent” and include the same in the declaration of “NgModule” decorator as shown in the below code.

```
import { GridComponent }   from '../Component/GridComponent';
@NgModule({
    imports: [RouterModule.forRoot(ApplicationRoutes),
        InMemoryWebApiModule.forRoot(CustomerApiService),
             BrowserModule,ReactiveFormsModule,
             FormsModule,HttpModule],
    declarations: [CustomerComponent,
                MasterPageComponent,
                SupplierComponent,
                WelcomeComponent,
                GridComponent],
    bootstrap: [MasterPageComponent]
})
export class MainModuleLibrary { }
```

And that’s it now hit “Control + B” on keyboard, run the server and see your reusable grid working.

Lab 11: - Lazy loading using dynamic routes

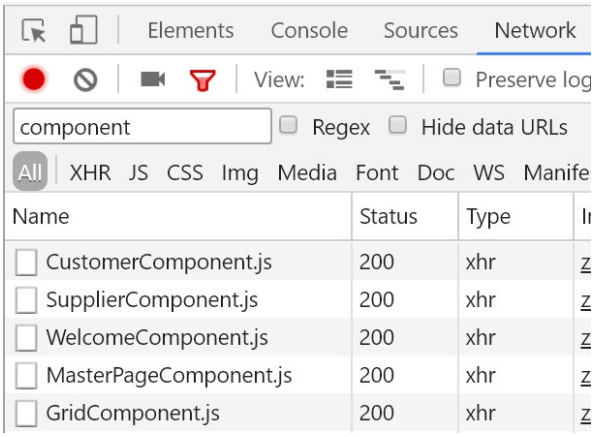
Theory

Big projects will have lot of components and modules, in other words we will end up with lot of JS files on the browser client side. Loading these JS files in ONE GO at the client browser side would really hit performance.

If you load the current application at this stage and see the developer tools you will see on the network tab all JS files are getting loaded at the start.

When the first time the user comes to the site we would like to just load the welcome component and master component JS only.

When the user clicks on supplier and customer respective JS files should be loaded at that moment.

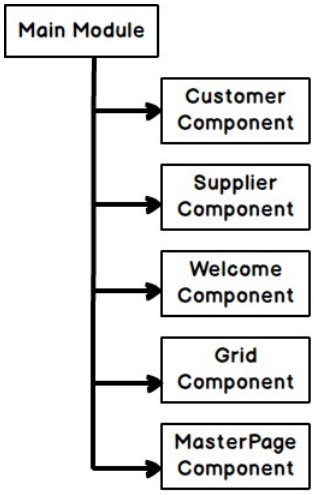


Let's investigate who is the culprit?

If you see the current architecture of our project we have one module (MainModule.ts) and all components current belong to this only ONE Module.

So when this module loads it loads all components inside it.

In simple words we need to BREAK MODULES in to separate physical module files.



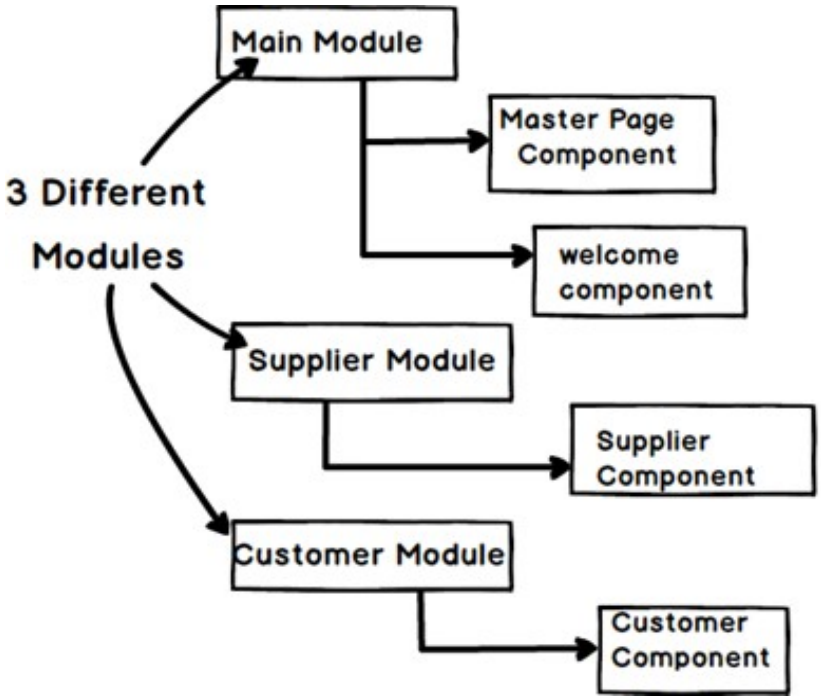
Step 1: - Creating three different physical modules

So as discussed in the previous part of the theory we need to first divide our project in to three different physical module files: -

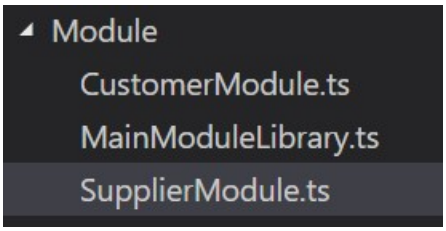
MainModule ,

SupplierModule and

CustomerModule.



So in the module folder let us create three different physical module files. We already have MainModule.ts we need to create two more.



MainModule.ts :- This module will load "MasterPageComponent.ts" and "WelcomeComponent.ts".

SupplierModule.ts :- This module will load "SupplierComponent.ts".

CustomerModule.ts :- This will load CustomerComponent and GridComponent. Remember grid is used only in Customer UI so this should load only when Customer functionality is loaded.

Step 2: - Removing Supplier and Customercomponent from MainModule

The first thing is we need to remove all references of CustomerComponent, SupplierComponent and GridComponent from the MainModule. Below is source code which is represented as strike out and now all it needs to be removed from the MainModule. In the main module, we only have reference to WelcomeComponent and MastePageComponent.

*Two modules are decoupled from each other **when import does not exist between those modules**. Even if you do not use the component and there is an import decoupling is not complete and the JS will be loaded.*

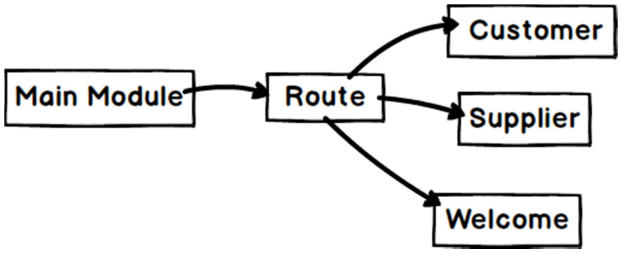
```
Lot of Code has been removed for clarity. Please download source code
for full code.
~~import { CustomerComponent }   from '../Component/CustomerComponent';~~
~~import { SupplierComponent }   from '../Component/SupplierComponent';~~
import { WelcomeComponent }   from '../Component/WelcomeComponent';
~~import { GridComponent }   from '../Component/GridComponent';~~
import { MasterPageComponent }   from
'../Component/MasterPageComponent';

@NgModule({
    imports: [RouterModule.forRoot(ApplicationRoutes),
        InMemoryWebApiModule.forRoot(CustomerApiService),
             BrowserModule,ReactiveFormsModule,
             FormsModule,HttpModule],
    declarations: [~~CustomerComponent~~,
                MasterPageComponent,
                ~~SupplierComponent~~,
                WelcomeComponent,
                ~~GridComponent~~],
    bootstrap: [MasterPageComponent]
})
export class MainModuleLibrary { }
```

Step 3: - Creating different Route files

As said previously "**A SIMPLE IMPORT REFERENCE**" will make two modules coupled. If the modules are coupled those JS files will be loaded.

If you remember "MainModule.ts" loads the Routes from "Routing.ts" and Routing.ts has import references to SupplierComponent and CustomerComponent.

So loading routing will load the other components as well and we will not be able to achieve lazy loading.



So let us remove all references of Customer and Supplier component from MainModule.ts, see the striked out code down below.

```
import {Component} from '@angular/core';
~~import {CustomerComponent} from '../Component/CustomerComponent';~~
~~import {SupplierComponent} from "../Component/SupplierComponent";~~
import {WelcomeComponent} from "../Component/WelcomeComponent";
export const ApplicationRoutes = [
~~    { path: 'Customer', component: CustomerComponent },~~
~~    { path: 'Supplier', component: SupplierComponent },~~
     { path: '', component:WelcomeComponent  },
    { path: 'UI/Index.html', component:WelcomeComponent  }
];
```

But still we need to still define routes for "Customer" and "Supplier" and the same time not use "import" statement as that makes the module coupled. If you look at the current syntax of defining route we need to have that component in the import or else we cannot define the route.

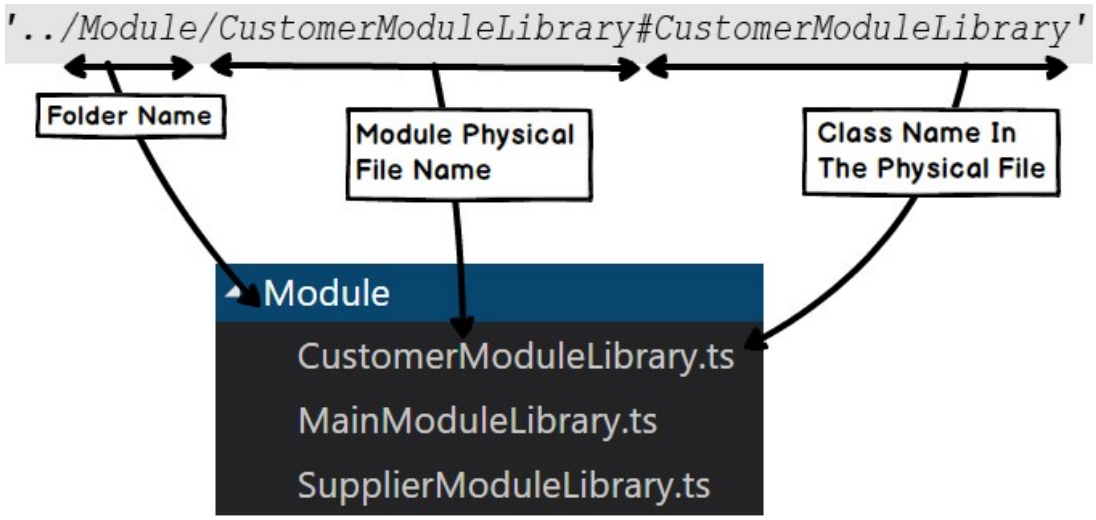
```
{ path: 'CustomerComponent', component:CustomerComponent  },
```

For that Angular has given a nice property called as “loadChildren”. In “loadChildren” we need to give the module in a single quote like a string. It thus means that this thing will be evaluated during run time and now compile time.

```
{
path: 'Customer',
loadChildren:'../Module/CustomerModuleLibrary#CustomerModuleLibrary'
}
```

The structure of “loadChildren” should follow this pattern: -

- The first element in the ‘loadChildren” is the folder name, in case module file is in a folder.
- The second element is the physical filename of the module. In our case we have “CustomerModuleLibrary.ts”, “SupplierModuleLibrary.ts” and so on.
- The third element after the “#” is the class name which should be loaded from the physical module file. It’s very much possible you can have many classes in one physical module file, but after the “#” we define which of those classes should be loaded.



The full code of the route will look something as shown below.

```
import {Component} from '@angular/core';
import {WelcomeComponent} from "../Component/WelcomeComponent";
export const ApplicationRoutes = [
    { path: 'Customer',
```

```
loadChildren:'../Module/CustomerModuleLibrary#CustomerModuleLibrary' },
    { path: 'Supplier',
loadChildren: '../Module/SupplierModuleLibrary#SupplierModuleLibrary'
},
     { path: '', component:WelcomeComponent  },
    { path: 'UI/Index.html', component:WelcomeComponent  },
    { path: 'UI', component:WelcomeComponent  }
];
```

We also need to create two more route files one for “Customer” and one for “Supplier” as shown below.

```
import {Component} from '@angular/core';
import {CustomerComponent} from "../Component/CustomerComponent";
export const CustomerRoutes = [
    { path: 'Add', component:CustomerComponent  }
];
```

```
import {Component} from '@angular/core';
import {SupplierComponent} from "../Component/SupplierComponent";
export const SupplierRoutes = [
    { path: 'Add', component:SupplierComponent  }
];
```

“SupplierRoutes” and “CustomerRoutes” are child routes while the “ApplicationRoutes” is the parent route.

Step 4: - Calling Child routes in Supplier and Customer modules

In supplier module and customer modules we need to load their respective routes defined in “Step 3”. To load child routes we need to use “RouterModule.forChild”.

```
import { NgModule }      from '@angular/core';
import { CommonModule } from '@angular/common';
import {FormsModule , ReactiveFormsModule} from "@angular/forms"
import { SupplierComponent }   from '../Component/SupplierComponent';
import { RouterModule }   from '@angular/router';
import { SupplierRoutes }   from '../Routing/SupplierRouting';
import {CustomerApiService} from "../Api/CustomerApi"
@NgModule({
    imports: [RouterModule.forChild(SupplierRoutes),
              CommonModule,ReactiveFormsModule,
              FormsModule],
    declarations: [SupplierComponent],
    bootstrap: [SupplierComponent]
})
export class SupplierModuleLibrary { }
```

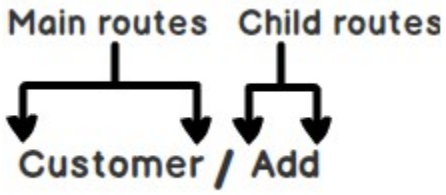
Same way we need to for Customer Module.

```
import { NgModule }      from '@angular/core';
import { CommonModule } from '@angular/common';
import {FormsModule , ReactiveFormsModule} from "@angular/forms"
import { CustomerComponent }   from '../Component/CustomerComponent';
import { GridComponent }   from '../Component/GridComponent';
import { RouterModule }   from '@angular/router';
import { CustomerRoutes }   from '../Routing/CustomerRouting';
import { InMemoryWebApiModule } from 'angular2-in-memory-web-api';
import {CustomerApiService} from "../Api/CustomerApi"
import { HttpModule } from '@angular/http';

@NgModule({
    imports: [RouterModule.forChild(CustomerRoutes),
        InMemoryWebApiModule.forRoot(CustomerApiService),
              CommonModule,ReactiveFormsModule,
```

```
              FormsModule,HttpModule],
    declarations: [CustomerComponent,
                GridComponent],
    bootstrap: [CustomerComponent]
})
export class CustomerModuleLibrary { }
```

Step 5: - Configuring routerlinks

In step 3 and 4 we have defined parent routes and child routes. Parent routes are defined in "Routing.ts" while child routes are defined in "CustomerRouting.ts" and "SupplierRouting.ts". So now the router link has to be changed to "Supplier/Add" and "Customer/Add" as shown in the below code.




```
<a [routerLink]="['Supplier/Add']">Supplier</a> <br />
<a [routerLink]="['Customer/Add']">Customer</a><br>
```

So now the full code look of master page looks as shown below.

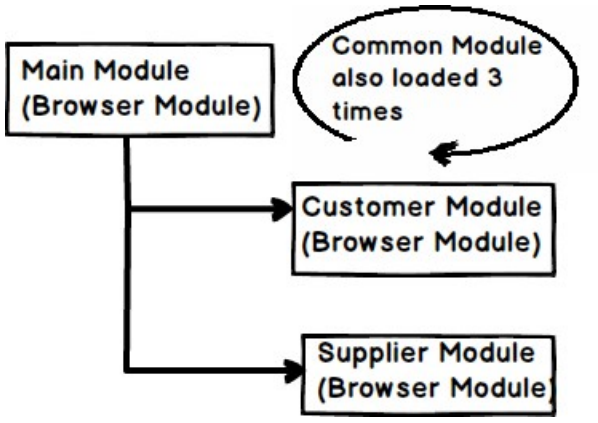
```
<table border="0" width="100%">
<tr>
<td width="20%"><img src="http://www.questpond.com/img/logo.jpg"
alt="Alternate Text" />
</td>
<td width="80%">Questpond.com Private limited</td>
</tr>
<tr>
<td valign=top><u>Left Menu</u><br />
<a [routerLink]="['Supplier/Add']">Supplier</a> <br />
<a [routerLink]="['Customer/Add']">Customer</a><br>
<a [routerLink]="['']">Home</a>
```

```
</td>
<td>
<div id="dynamicscreen">
<router-outlet></router-outlet>
</div></td></tr>
<tr>
<td></td>
<td>Copy right @Questpond</td></tr></table>
```

Step 6: - Replacing browser module with common module

"BrowserModule" and "CommonModule" are modules of angular. "BrowserModule" has code which starts up services and launches the application while "CommonModule" has directives like "NgIf" and "NgFor".
"BrowserModule" re-exports "CommonModule". Or if I put in simple words "BrowserModule" uses "CommonModule". So if you are loading "BrowserModule" you are loading "CommonModule" also.

So now if you are loading "BrowserModule" in all three modules then you will end uploading "CommonModule" 3 times. When you are doing Lazy Loading you really do not want to load things 3 times, it should be loaded only once.



So if you have "BrowserModule" in all three modules then you would end up getting such kind of error as shown in below figure.

This error says "BrowserModule" has already been loaded in the "MainModule" please use "CommonModule" in "CustomerModule" and "SupplierModule".

So in main module load the browser module and in rest of modules load “CommonModule”.

```
import { BrowserModule } from '@angular/platform-browser';
// Other imports have been removed for clarity
@NgModule({
    imports: [RouterModule.forRoot(ApplicationRoutes),
        InMemoryWebApiModule.forRoot(CustomerApiService),
             BrowserModule,ReactiveFormsModule,
             FormsModule,HttpModule],
    declarations: [
                MasterPageComponent,
                WelcomeComponent],
    bootstrap: [MasterPageComponent]
})
export class MainModuleLibrary { }
```

But in customer and supplier module just load common module.

```
import { CommonModule } from '@angular/common';
// other imports has been removed for clarity

@NgModule({
    imports: [RouterModule.forChild(SupplierRoutes),
```

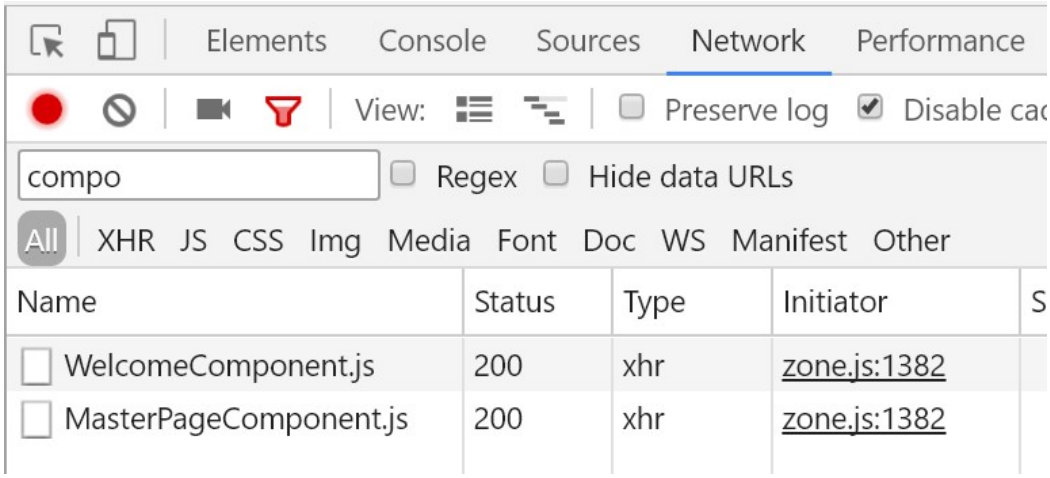
```
            CommonModule,ReactiveFormsModule,
            FormsModule],
    declarations: [SupplierComponent],
    bootstrap: [SupplierComponent]
})
export class SupplierModuleLibrary { }
```

```
import { CommonModule } from '@angular/common';
// Other import has  been removed for claroty
@NgModule({
    imports: [RouterModule.forChild(CustomerRoutes),
        InMemoryWebApiModule.forRoot(CustomerApiService),
            CommonModule,ReactiveFormsModule,
            FormsModule,HttpModule],
    declarations: [CustomerComponent,
                GridComponent],
    bootstrap: [CustomerComponent]
})
export class CustomerModuleLibrary { }
```

Step 7: - Check if Lazy loading is working

Now run your application, go to network tab and check if lazy loading is working. You can see when the application run at the start only “WelcomeComponent” and “MasterPageComponent” is loaded. Once you click on supplier and customer the respective components will loaded at that time.

Please put a proper filter so that you do not see all JS files in your network.

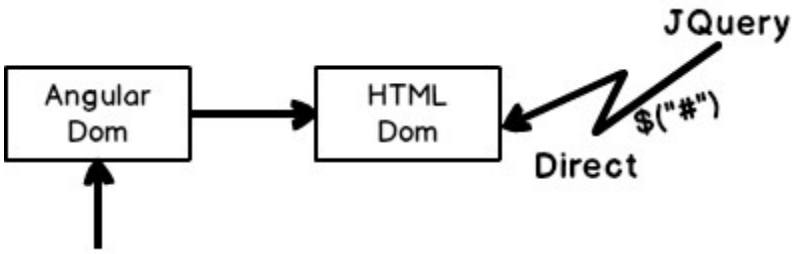


Lab 12: - Using jQuery with Angular

Introduction

If you look at jQuery it is a very old and trusted JavaScript framework. It has lot of UI components which are stable and trusted. As a practice, you should not use jQuery with Angular because both of them can overlap with DOM manipulation causing confusion.

jQuery manipulates HTML DOM directly using "$" syntaxes while Angular creates a sugar-coated DOM over HTML DOM and does the manipulation via this sugar-coated DOM.

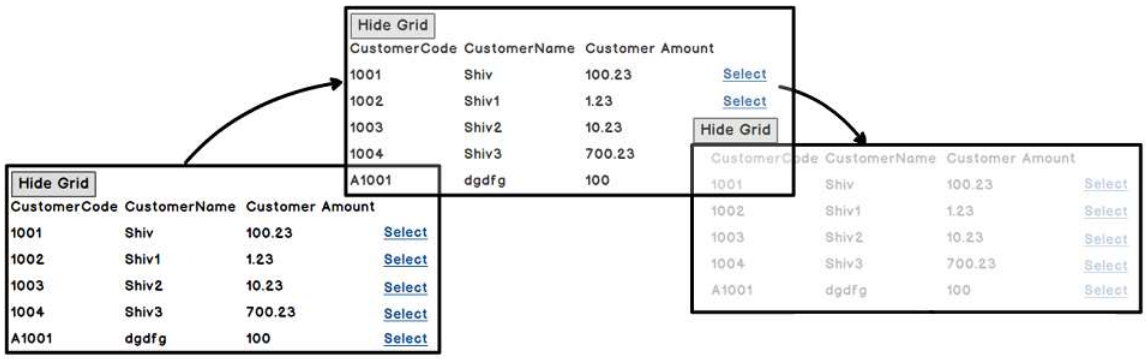


So jQuery can manipulate HTML DOM without Angular not having knowledge about this manipulation creating more confusion.

But then there are instances when we would like to use jQuery UI components like jQuery grids, jQuery calendar and so on which probably are not available in Angular.

> *If Angular is your prime framework, then first see that you get your solution inside Angular if not then use jQuery or any other framework.*

In this lab we will use jQuery to fade away and fade in our grid component. So let's create a button with name Hide grid. When the end user clicks on hide grid the grid should gradually become visible and invisible.



Step 1: - Install jQuery

So the first step is to get jQuery. Let's fire up the node command and also let's get jQuery as well as let us save the entry in to "package.json" file.

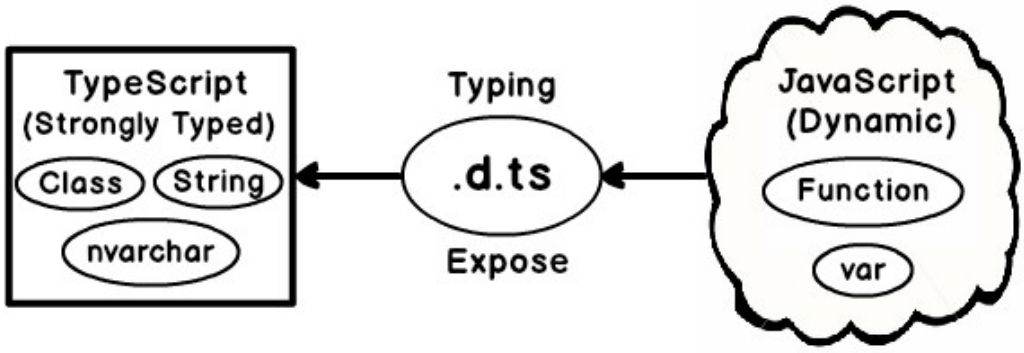
```
npm install jquery -save
```

Step 2: - Install jQuery Typings

JavaScript is divided in to two generations, one generation before TypeScript i.e. pure JavaScript and other generation after typescript. JavaScript is a dynamic and an untyped language while TypeScript is strongly typed. We can call methods which does not exist, assign variables which are not created as so on.

On other side TypeScript is strongly typed. Everything is done during design time/compile time, TypeScript has to know the methods, parameters everything upfront.

Now frameworks like jQuery are made in pure JavaScript, so if they need to be used in TypeScript we need to expose their types, parameters and so on. That's where we need to create typing files. Typing's are TypeScript files which expose the shape and structure of JavaScript objects so that TypeScript can understand JavaScript types during design time.



You must be wondering so do we need to create the typing's file manually? No you do not need to. jQuery already has typing's on the contrary almost all popular JavaScript frameworks have their typing's file.

So to get the jQuery typing's we need to do npm install by pointing at "@types/jquery". In fact you can load any types by using "@types" for example if you want to load lodash you can do "npm install" on "@types/lodash".

```
npm install @types/jquery -save
```

Step 3: - Providing ID's in the UI

If you see jQuery refers HTML UI elements using selectors. In selectors, we need to provide name or ids by which we can get reference of that HTML element. So let's wrap our "grid" component inside a DIV tag and lets assign some "id" value to it, like the one we have given in the below code "divgrid".

Also we have created a button which calls the "Fade" methods from the component.

```
<input (click)="Fade()" type="button" value="Hide Grid"/>
<div id="divgrid">
<grid-ui
      [grid-
columns]="[{'colName':'CustomerCode'},{'colName':'CustomerName'},{'colN
ame':'CustomerAmount'}]"
    [grid-data]="Customers"
    (grid-selected)="Select($event)"></grid-ui>
<div>
```

Step 4: - Importing and using jQuery in Component

Now the first thing is to import jQuery in to your component file. For that we need to use the below code. You can see the import statement is bit differently used. We are using "*" and "as" keyword. "*" means we want to refer full JQuery file and "$" is the alias by which we will be referring jQuery syntax in the code.

```
import * as $ from "jquery";
```

Once jQuery has been imported we can now use "$" to execute jQuery syntaxes. You can see we have created a method called as "Fade" in which we are referring the HTML DIV ID and calling the "fadeToggle" method to fade in and out.

```
import * as $ from "jquery";
export class CustomerComponent {
   // Code removed for clarity
   Fade(){
        $("#divgrid").fadeToggle(3000);
    }
   }
```

Step 5: - Make entry in to Systemjs.config.js

Our JS files are loaded using SystemJS module loader. So we need to make an entry in to "Systemjs.config.js" file stating where the jQuery file is located.

```
'jquery': '../node_modules/jquery/dist/jquery.js'
```

If you see we have specified the folder path as "dist". "dist" stands for distribution folder. The final compiled copy of jQuery is in this folder so we have specified the same.

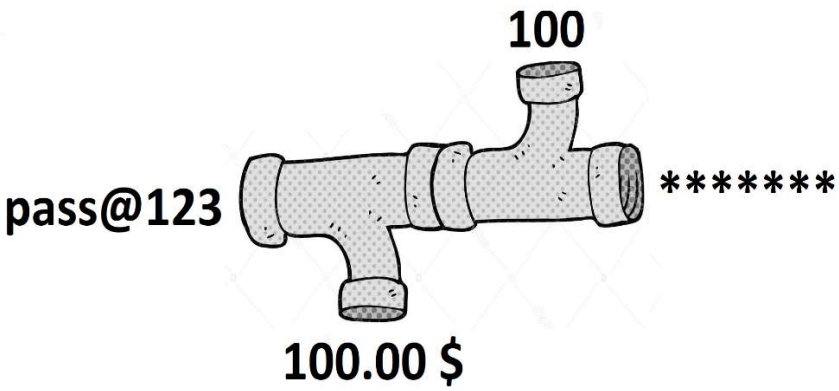
Now run the program and see the output. If you click on the "Hide" button you should see the grid fading out and in.

Lab 13: - Pipes in Angular

This lab is the smallest of all labs till now with some theory followed by three steps. So if you are tired and worn out, this lab will be relaxing for you.

Introduction

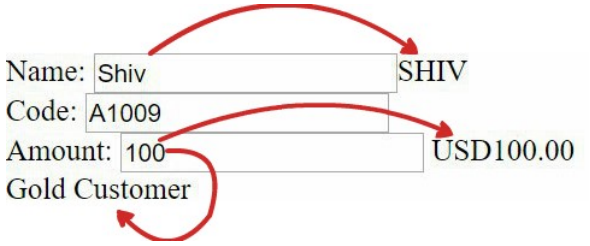
Pipes take in a data input and transforms data to a different output. For example, pipe can take "Learn Angular" as input and transform to "LEARN ANGULAR" in capital or you can give input as "100" which you want to display (transform) as "100.00 $" on UI.



So for our lab number 13 we will do the following pipe transforms: -

- Change Customer name in to capital letters.
- Change Customer amount to proper formatted amount with $ currency sign.
- Depending on the Amount we would like to display “Gold” or “Silver” customer status. So if the amount exceeds 100 it will display “***Gold Customer***” and if less than 100 it will display “***Simple Customer***”.

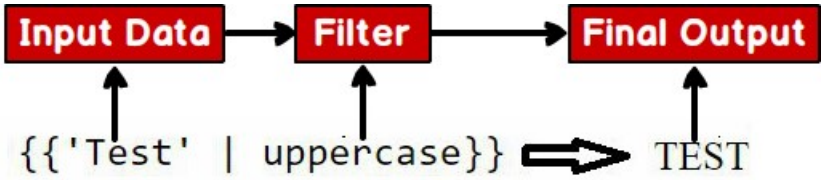
See down the image for more detail of how the output produced on the browser would really look like.



Understanding syntax of pipes

Before we move ahead with labs on pipes let us understand how the syntax of pipes looks like. You need to write pipe syntax inside an expression “{{ }}”. You need to first put data that needs to be formatted and then followed by “|” symbol as shown in the below figure. Once you run it, it displays final formatted output on the HTML.

Pipe syntaxes are written inside HTML.



Step 1: - Applying readymade pipes

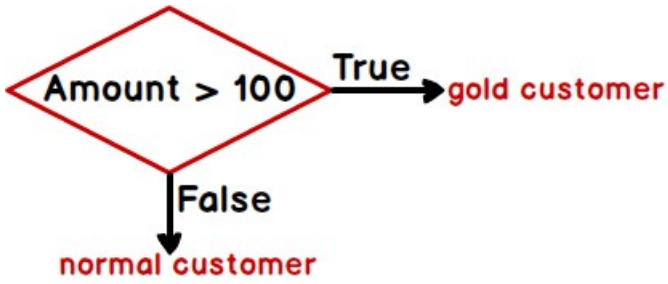
There are two broader categories of pipes:- Readymade pipes and Custom pipes. You can get list of readymade pipes from https://angular.io/api?type=pipe.

So to display “CustomerName” in capital and “CustomerAmount” with currency sign we can use two readymade filters “uppercase” and “currency” respectively. Below are the code syntaxes for the same.

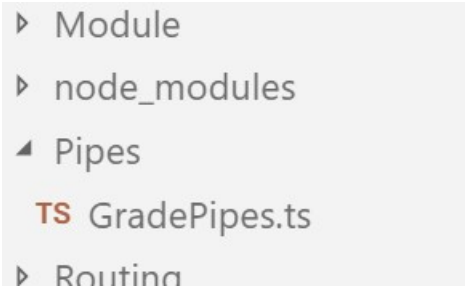
```
{{CurrentCustomer.CustomerName | uppercase}}
{{CurrentCustomer.CustomerAmount | currency}}
```

Step 2: - Creating custom pipes for customer grading

But the third requirement is a bit complicated. In this is the entered amount is greater than 100 we need to display “Gold Customer” and if it’s less than 100 we need to display “Normal Customer”.



So let’s create a separate folder called as “Pipes” and in that lets create “GradePipes.ts” as shown in the below figure.



In “GradePipes.ts” we will write the code which will have the logic. So first thing we need to import “Pipe” and “PipeTransform” from core as these classes are the building blocks for pipes.

```
import { Pipe, PipeTransform } from '@angular/core';
```

The next thing is we need to create a “GradePipes” class which implements the “PipeTransform” interface and has “@Pipe” decorated on the top of it. Every pipe class should have a “transform” method as it is small “t” “r”… “transform” which has a input parameter i.e. customer amount which will be provided in the UI.

```
@Pipe({name: 'GradePipes'})
export class GradePipes implements PipeTransform {
  transform(value:number): string {
        if(value < 100){
            return "Simple Customer";
        }
        else{
            return "Gold Customer";
        }
  }
}
```

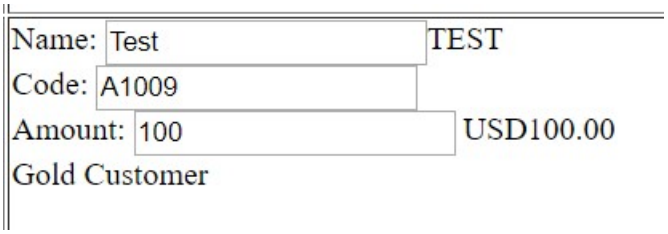
In "@Pipe" decorator we have also specified a "name". This name will be used in the UI to make call to the pipe.

Step 3: - Applying the grading pipe on the UI

And then finally we need to call the pipe in the HTML UI. We need can now call "GradePipes" as shown in the below code. "CurrentCustomer.CustomerAmount" is the input and the pipe is "GradePipes".

```
{{ CurrentCustomer.CustomerAmount|GradePipes}}
```

Once done, enjoy your produced output.



Lab 14: - Providers, Services and Dependency Injection

The Problem

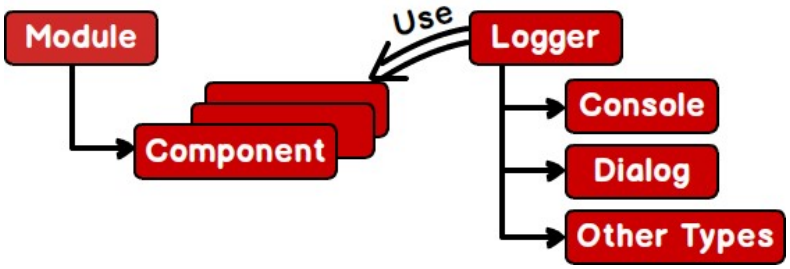
Courtesy: - https://medium.com/nestle-usa/4-tips-for-starting-your-baby-on-fruits-veggies-aafd6b2c0cde

A sign of really good software architecture is: -

"When you make change at one place you do not need to change at many places".

Now take the below situation. In your project let's say you have lot of modules and 100's (just put a figure) components. Now in all your 100 components you want to implement logging utility.

In logging utility also you have different varieties. Some logger just logs to the console of the browser while some display using dialog boxes.



So assume that these loggers are used in those 100's of components to log error and instrumentation messages. So in all your components you will do two things: -

- Import the logger component using "import" syntax.
- And then create object of the logger.

Below is the code snippet for it.

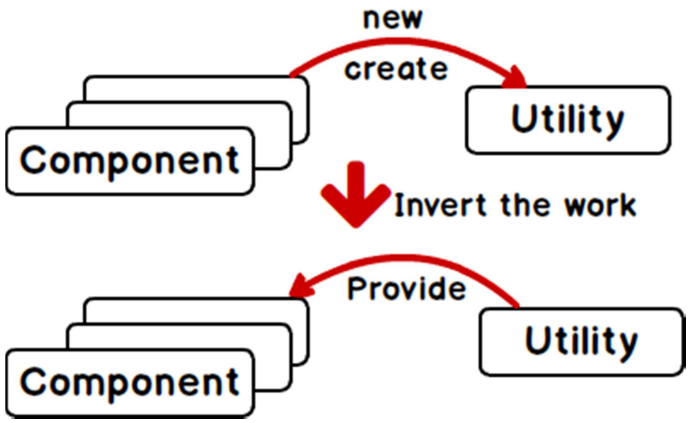
```
import { ConsoleLogger} from "../Utility/Utility"
// code deleted for clarity
export class CustomerComponent {
    logger: ConsoleLogger = new ConsoleLogger ();
    // code removed for clarity
}
```

Now let's say somewhere down the line you think you want to log messages using dialog boxes , think about the amount of changes you need to make in your 100 of components.

Solution DI and IOC

Let us conclude what is the cause which leads to change in all components. To solve this problem below is the solution, read the below sentence slowly with full senses alive.

"The problem is because Component is creating the object of Utility, if we can INVERT this, means if we can inject /provide "Utility" to the components rather than creating, then this will make our architecture better".



In other words, we need to implement “Inversion of control”. Means invert the object creation to someone else and component just refers a generic utility reference.

So rather than components creating the object of the utility component will ASK for providing/ injecting the “logger” object via constructor.

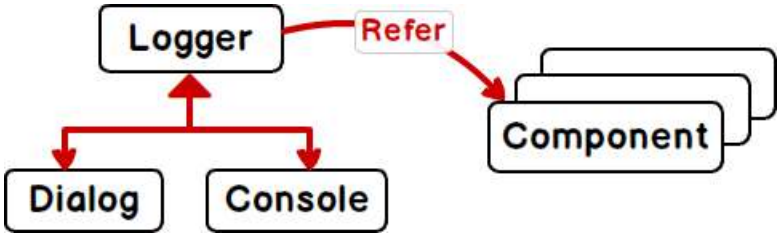
```
export class CustomerComponent {

        constructor(logger:Logger){
        }
}
```

Inject/Provide object

IOC is a concept and to implement IOC we need to use DI. I would encourage you to see this YouTube video which explains DI and IOC in detail
https://www.youtube.com/watch?v=FuAhnqSDe-o

So now that we know DI and IOC concepts, let’s do a demonstration for the same.

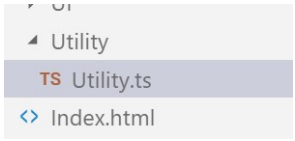
Demo of DI IOC



Here is a simple demo we will implement in our project to see the importance of Provider DI. So we will create a simple parent “Logger” class and we will create two flavour’s one which displays messages one on dialog boxes and other to the console of the browser.
We will then try to figure out how easy it is to make changes at one place & replicate it across all places.

Step 1: - Create the logger utility classes with Injectable attribute

So let's create a folder called as "Utility" inside which we will create these logger classes.



Let's create a parent "Logger" class from which we will inherit and create different flavours of "Logger" class. Now because this class will be injected in to the components we need to mark it with "@Injectable()" decorator. "@Injectable()" decorator is available in "Injectable" in "angular/core".

```
import { Injectable } from '@angular/core';
@Injectable()
export class Logger{
   public  Log(){
    // some default logging
   }
}
```

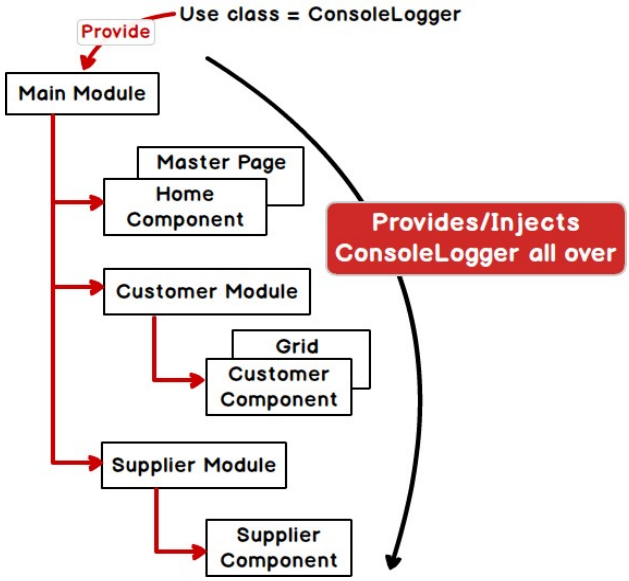
Below is the code for "ConsoleLogger" and "DialogLogger" classes which inherit from "Logger" class. For now both classes do not have any functionality as such. At this moment let us focus on DI rather than logging functionality.

```
@Injectable()
export class ConsoleLogger extends Logger{
    public Log(){
        // do console logging here
    }
}
@Injectable()
export class DialogLogger extends Logger{
    public Log(){
        // do console logging here
    }}
```

Step 2: - Define providers and injection in the modules

So as we discussed previously the goal is that we want to change at one place and the logger types should change in all components.

Now the whole loading (bootstrapping) of our project starts with "MainModuleLibrary", so if we can inject at the level of "MainModuleLibrary' it will be available to all components and modules down below.



In the "MainModuleLibrary" using "@NgModule" decorator we need to provide the providers which will be trickled down to all components. In providers we need to specify two things: -

- First the parent class which be referred in all components. So using the "provide" we have specified "Logger" class.
- Second need to provide which child implementation object we want to inject in all components. That's provided in "useClass".

```
providers: [
      {
          provide: Logger,
        useClass: ConsoleLogger
      }  ]
```

The complete code of the main module with "providers" looks something as shown below.

```
// Imports have been removed for clarity
import {Logger,DialogLogger,ConsoleLogger} from "../Utility/Utility"
@NgModule({
    imports: [RouterModule.forRoot(ApplicationRoutes),
```

```
            HttpModule,
            InMemoryWebApiModule.forRoot(CustomerApiService),
             BrowserModule,
             FormsModule,
             ReactiveFormsModule],
    declarations: [
                    MasterPageComponent,
                WelcomeComponent],
    bootstrap: [MasterPageComponent],
    providers: [
      {
        provide: Logger,
        useClass: ConsoleLogger
      }
    ]
}) export class MainModuleLibrary { }
```

Step 3: - Define constructors for injection

Now that we have define on the main module which object will inject. In the components we need to expose the “logger” class in the constructor for DI purpose.

Note: - All your components should only refer “Logger” parent class and not the child classes. You can see in the below code we have imported only “Logger” class and through constructor only “logger” class is referred.

```
// code removed for clarity
import {Logger} from "../Utility/Utility"
@Component({
    templateUrl: "../UI/Customer.html"
})
export class CustomerComponent {
    constructor(public http:Http , logger:Logger){
        this.Display();
```

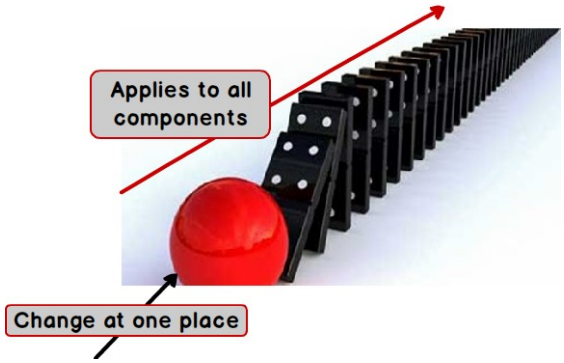
```
    }
// code removed for clarity
}
```

Step 4: - Test if things are working

Run the application, put a debug point and see the “ConsoleLogger” object getting injected. If you change it to “DialogLogger” you will see the same injected.

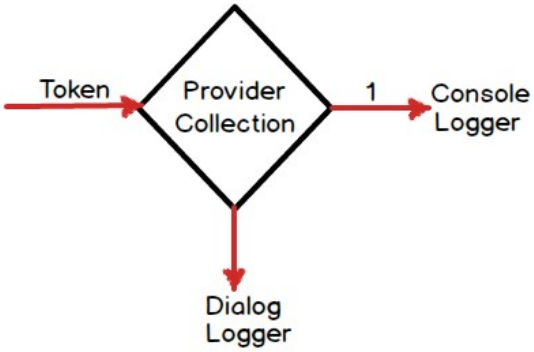
Take a deep breathe , close your eyes and think . You are changing in the main module and its getting propogated all over.

As we started this lab saying a “Good architecture is all about changing at one place and the changes are reflected through out”.



A word about token based providers

In real time scenarios dependency injection of objects will not be static, but rather dynamic. Angular has provided something termed as "token" for such scenarios.



From application we can provide "tokens" which can be objects, values and then depending on the token respective object will be injected.

For example, in the above lab scenario we would like to inject "ConsoleLogger" when someone passes "1" & "DialogLogger" when we pass "2".

In the module class we need to provide the providers with "token" value as shown in the below code. In the "providers" we have provided collection of all dependencies. So "DialogLogger" can be fetched using "1" and "ConsoleLogger" using "2".

```
@NgModule({
    // code removed for clarity
    providers: [
      {provide:'1', useClass : DialogLogger},
      {provide:'2', useClass : ConsoleLogger}
    ]
})
export class MainModuleLibrary { }
```

Once the providers are loaded we can then look up using the "get" function of the "injector" object.

```
import {  Injector } from '@angular/core';
constructor(public injector: Injector){
        this.logger = this.injector.get("2");
    }
```

You can also make it more dynamic by creating a collection separately and loading it using HTTP calls and then connect that collection to the “providers” property in the module. In the below code we have created a separate “providerscoll” collection and that collection is provided to the provider property.

```
export const providerscoll= [
    { provide: "1", useClass: DialogLogger },
    { provide: "2", useClass: ConsoleLogger },
  ];
@NgModule({
    // code removed for clarity
    providers: [providerscoll]
})
export class MainModuleLibrary { }
```

You can use the “push” method of JavaScript collection and load the providers dynamically as well.

```
var providerscoll:any = [];
providerscoll.push({ provide: "1", useClass: DialogLogger });
providerscoll.push({ provide: "2", useClass: ConsoleLogger });
@NgModule({
    // code removed for clarity
    providers: [providerscoll]
})
export class MainModuleLibrary { }
```

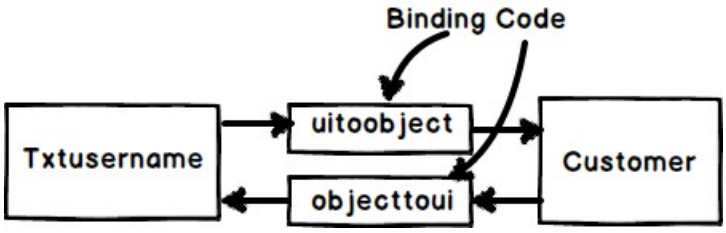
Angular Interview Questions and Answers

What is the goal of Angular?

Angular is a binding framework which helps to bind HTML UI with JavaScript objects.
In complicated JavaScript projects, you end up writing such kinds of below functions for binding. These functions bind the UI with JavaScript objects and vice versa.

```
function UitoObject()
{
      Cust.CustomerName = $("#TxtCustomerName").val();
}
function ObjecttoUi()
{
      $("#TxtCustomerName").val(Cust.CustomerName);
}
```

These functions become complicated as the UI gets complicated. Angular is a binding framework and makes your binding easy.



So when you use Angular you do not need to write the above complicated functions you end up writing directives as shown in the below code. The below directive “ngModel” will bind data from the textbox to the customer objects and vice versa.

```
<input type="text" [(ngModel)]="cust.CustomerName">
```

What is the difference between AngularJS and Angular?

Angular 1 is called as “AngularJS” while Angular 2/4 is termed as “Angular”. So when you say “AngularJS” you are referring the “1.X” version and when you say Angular you are talking about the new version.

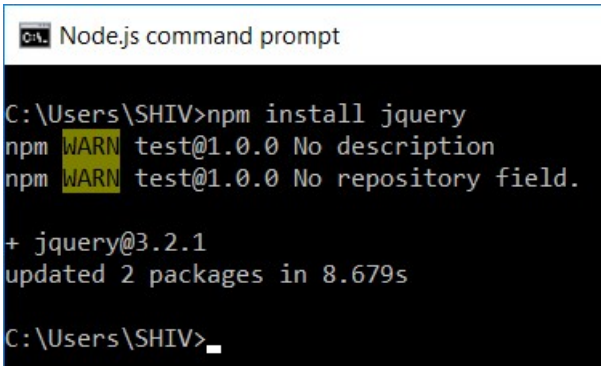
How different is Angular 1.X from Angular 2.X and 4.X?

Angular 2 is completely different from Angular 1. The syntaxes and project structure are completely different. Also automatic migration of an Angular 1 project to Angular 2 is not possible.

While Angular 2 and Angular 4 are backward compatible. Angular 2 projects can work with Angular 4 framework without any tweak.

What is NodeJS and why do we need it for Angular?

NodeJS has something called as NPM (Node Package Manager). Using NPM we can install any JavaScript framework. So once you install node, you can go to Node command prompt and type command as shown in the below figure.



So in Angular projects NPM is used to install Angular framework.

What is the importance of package.json file?

"package.json" file is a feature provided by Node. Rather than writing NPM commands again and again. You can create a "package.json" file listing all dependencies as shown in below code and just fire "NPM INSTALL" command. That will install all the packages mentioned in the file.

```
{
  "name": "test",
  "version": "1.0.0",
  "description": "",
  "dependencies": {
    "jquery": "^3.2.1",
    "knockout": "^3.4.2",
    "lodash": "~4.17.4"
  }
}
```

So how do we install Angular using NodeJS?

To install Angular you need to create a “package.json” file with all Angular dependencies listed and then do a NPM INSTALL on the folder where this file is. This file is available on the main site of Angular as well.

```
{
  "name": "angular-quickstart",
  "version": "1.0.0",
  "license": "ISC",
  "dependencies": {
    "@angular/common": "4.0.0",
    "@angular/compiler": "4.0.0",
    "@angular/core": "4.0.0",
    "@angular/forms": "4.0.0",
    "@angular/http": "4.0.0",
    "@angular/platform-browser": "4.0.0",
    "@angular/platform-browser-dynamic": "4.0.0",
    "@angular/router": "4.0.0",
    "@angular/upgrade": "4.0.0",
    "bootstrap": "^3.3.6",
    "core-js": "^2.4.1",
    "http-server": "^0.10.0",
    "reflect-metadata": "^0.1.3",
    "rxjs": "^5.4.1",
    "systemjs": "0.19.27",
    "zone.js": "^0.8.4"
  }
}
```

Which NPM commands do you use frequently?

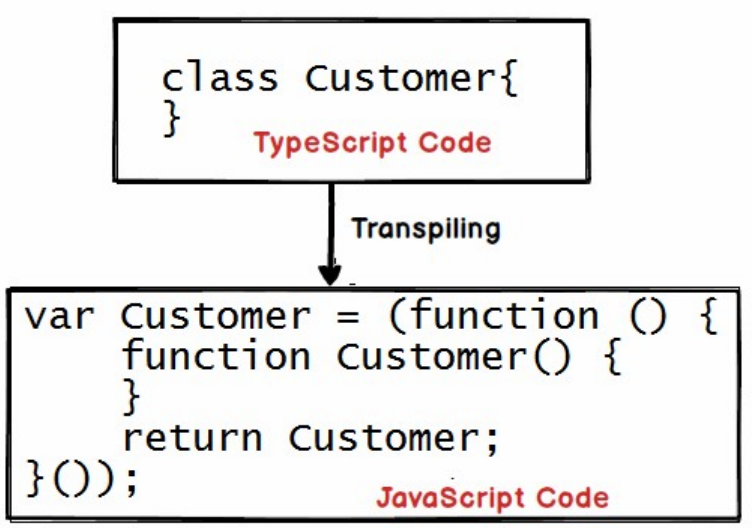
NPM has huge command list. In these kinds of questions interviewer is expecting you to talk about common NPM commands. So below are some important ones.

Command	Explanation
npm install -g typescript	This installs the package globally.
npm install -save jquery	This will install the package and also make a entry in “package.json” file.
npm view -version jquery npm view -versions jquery	This first command will show you latest jQuery version on GitHub and the second one will show all version in a ascending manner.
npm install -g npm	This upgraded NPM to latest version

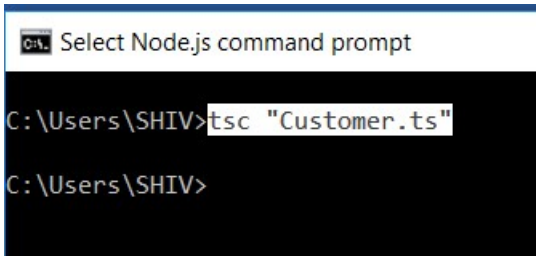
What is the use of TypeScript in Angular?

TypeScript is a transpiler over JavaScript. It provides sugar coated syntax over JavaScript where developers can apply OOP principles like class, inheritance, interfaces and so on.

Once you hit compile TypeScript will transpile the code to pure JavaScript.



To compile code we need run “tsc” from command line.



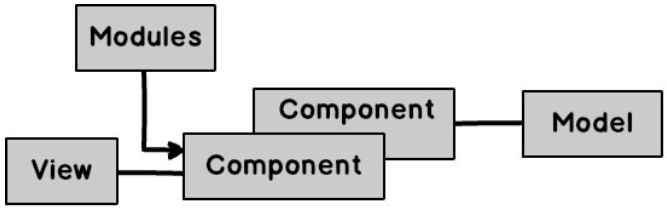
What is the importance of tsconfig.json?

“tsconfig.json” defines how TypeScript compiler will compile. Below is a simple code snippet of tsconfig file, which do compile using ES 5 specifications, remove comments and put all compiled JS files to “Shiv” directory.

```
{
  "compilerOptions": {
    "target": "es5",
    "removeComments": false,
    "outDir": "/Shiv"
  }
}
```

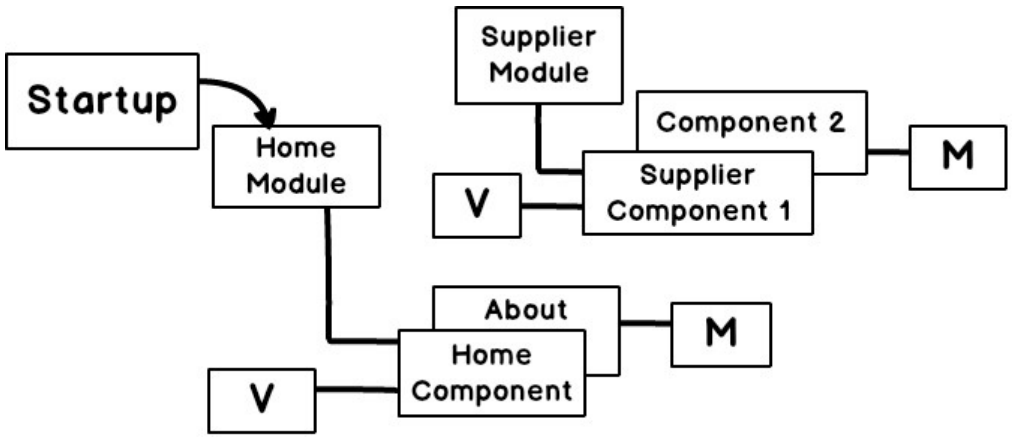
Explain Components and Modules and how to create them?

Components have the binding code which binds the view with the model. While modules group these components. So in complex angular project can have “Customer component”, “Customer reporting component” which is grouped in “Customer module”. In the same project you can have supplier module which groups 1 or many supplier components.

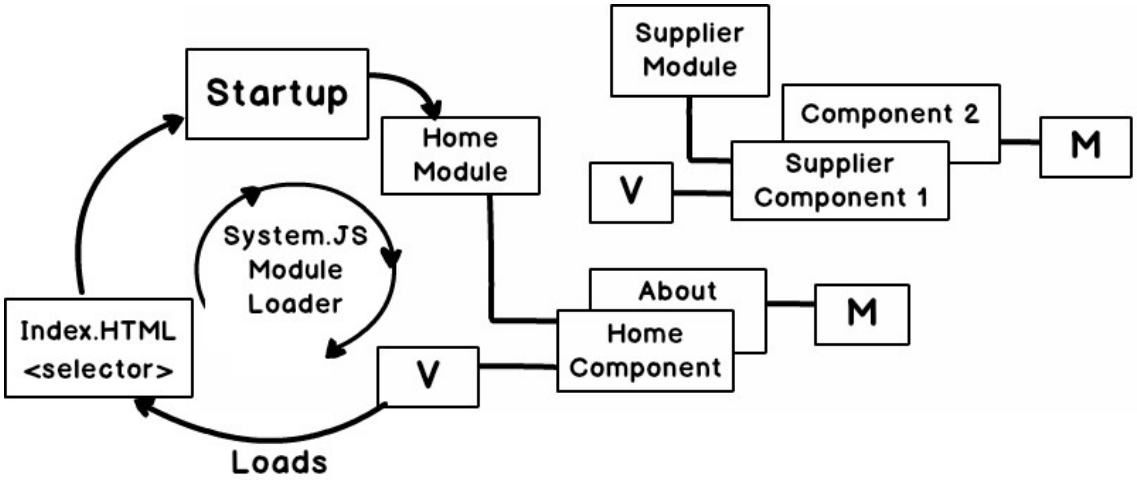


What is the importance of Startup.ts file?

In a complex angular project you can have lot of modules like home module, supplier module and so on. “Startup.ts” file determine which states of the module will become your startup module.



Explain how a typical angular project loads?



Future road map for Edition 3

Lab 15:- Pathlocation and HashLocation
Lab 16:- Auxiliary router outlet
Lab 17:- Going live to production using Webpack
Lab 18:- Communicating between components using viewChild
Lab 19:- Sharing data between modules
Lab 20:- ElementRef
Lab 21:- Angular CLI

Acronym used in this book

- NPM: - Node Package Manager.
- TS: - TypeScript.
- JS: - JavaScript.
- VS: - Visual Studio.
- WP: - Webpack
- OS: - Operating system.
- SPA: - Single page application.
- AOT: - Ahead of Time compilation
- DI: - Dependency injection
- IOC: - Inversion of Control