Learn Angular step by step
(Covering Angular 2 and Angular 4)

By

Version 1 (Beta)
This book is in a preliminary stage. We have covered 5 labs till now. And our goal is to have 10 Labs.
There is TBD section (To be done) at the last which we will be completing in the coming 1 month.

Contents

Technical support for this book

If you have any issues this book , any correction please log a issues at
https://github.com/shivkoirala/LearnAngular/ and we will be more than happy to support it. When
you a log a issue please mention the lab number and step number and in detail what error and issue
you are facing.

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.
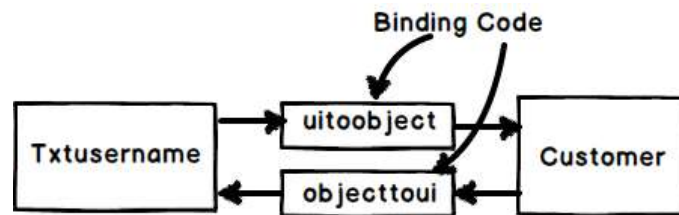
Binding Code

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.
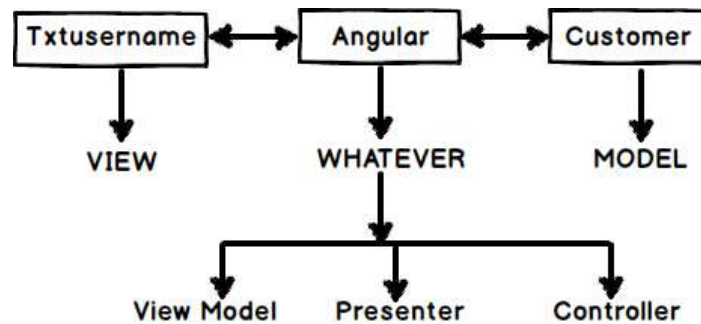
Now that binding code have different vocabularies.

- 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.

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.

We will create a Customer data entry project as shown in screen below with below features:-

- Application should have capability of accepting Customer name , Customer and Customer Amount values.
- Customer name and Customer codes are compulsory fields and it should be validated.
- Application will have a "Add" button which help us to post the current customer data to a Server API. Once the data is added to the server it should display in a grid.
- Application will have a navigation structure where in 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.

Questpond.com Private limited

Left Menu
Supplier
Customer
Home

Customer Name: [   ]
Customer name is required
Customer Code: [   ]
Customer code is required
Customer Amount: [0]

0
[Add]

CustomerCode	CustomerName	CustomerAmount	
1001	Shiv	100.23	Select
1002	Shiv1	1.23	Select
1003	Shiv2	10.23	Select
1004	Shiv3	700.23	Select

Copy right @Questpond

Angular is easy but his friends are difficult

In my initial journey of learning Angular I realized that Angular by itself is easy to understand. But Angular uses lot of JavaScript open sources and if you do not know these open sources Learning Angular would become extremely difficult.

So we would recommend spending your some time going through the below videos and making notes around the same:-

Topic name	YouTube URL source
Node JS	https://www.youtube.com/watch?v=-LF_43_Mqnw
Type Script	https://www.youtube.com/watch?v=xqYD8QXJX9U
VS code	https://www.youtube.com/watch?v=gQ9CiRlRPKs
System JS	https://www.youtube.com/watch?v=nQGhdoIMKaM
Common JS concept	https://www.youtube.com/watch?v=jN4IM5tp1SE

Lab 1 :- Getting started with Basics of Angular

Introduction

Now this is a 200 page book , so reading needs patience.so if you want to go super-fast you can watch the below Angular video. This video explains how to get your first Angular program running. Please note the youtube video uses Visual studio as tool while this book uses VS code.

So I have kept both options open so that Microsoft guys are happy and the non-Microsoft people do not feel left over.

https://www.youtube.com/watch?v=oMgvi-AV-eY

Step 1:- Installing VS Code, Typescript and Node

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 in order to expedite the learning process we would be using some tools and software's.

The first tool which we would need is Visual studio code editor or in short you can say VS code editor. 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.
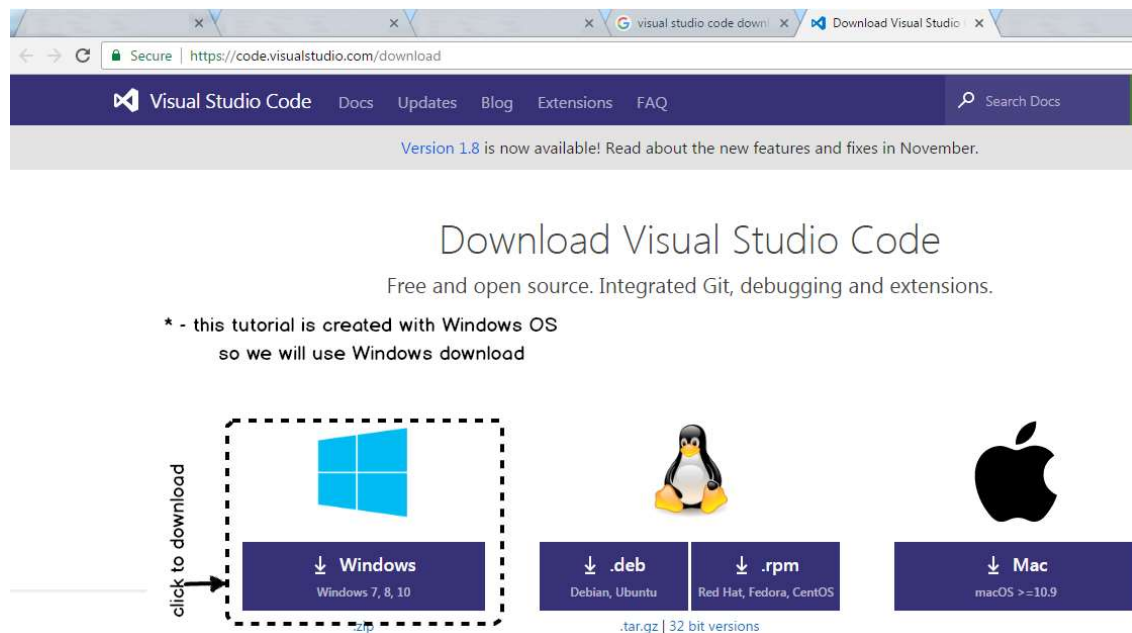
Once you download the setup it's a simple setup EXE run it and just hit next , next and finish.
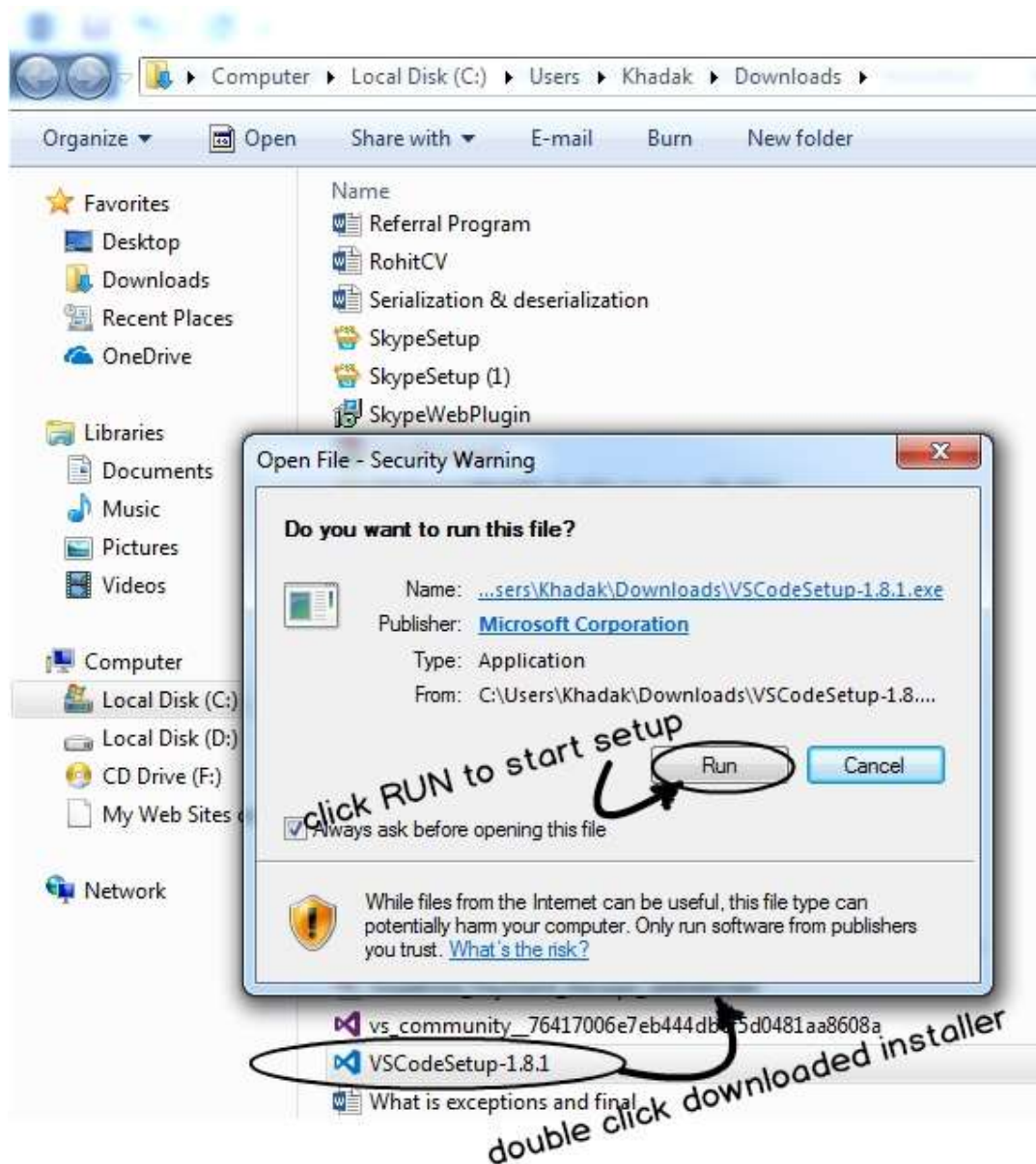
The second thing we need is "Nodejs". We would encourage you to watch this video which explains the basic use of Node JS , Node JS Explained.

If I put in simple words NodeJs does the following two things:-

1. It has something called as "NPM" node package manager. If you watch around you will see lot of JS frameworks coming up every day. Each one of these frameworks have their own website, github repository with weird names and they are releasing new versions now and then. So to get these frameworks we need to go to their site download the JS files and so on. NPM is a central repository where all these things are registered. And as developer if you want to get a specific JS open source you can just type NPM commands like "npm install jquery". This command will bring the recent version of Jquery in your computer folder.

2. The second thing which node does is it helps you to run JavaScript outside the browser. So tomorrow if you want to build a server application or windows application using JavaScript then this thing might help.

So to get node you can go to www.nodejs.org and depending on your OS select the appropriate download as shown in the below figure.
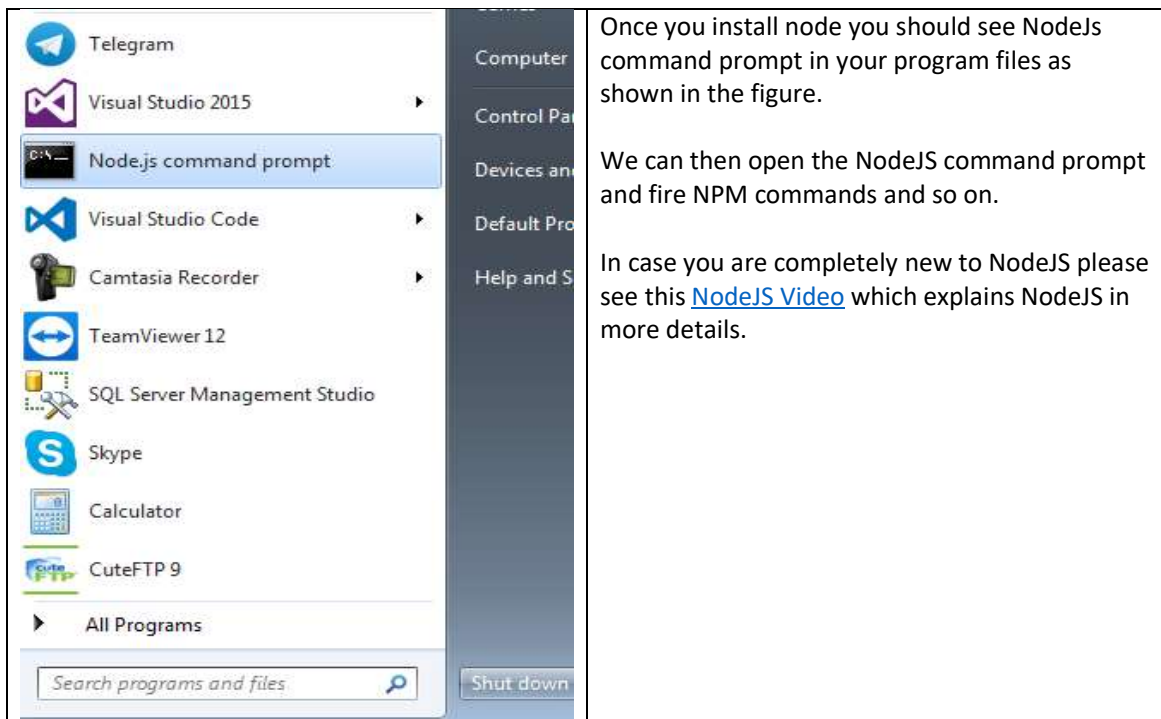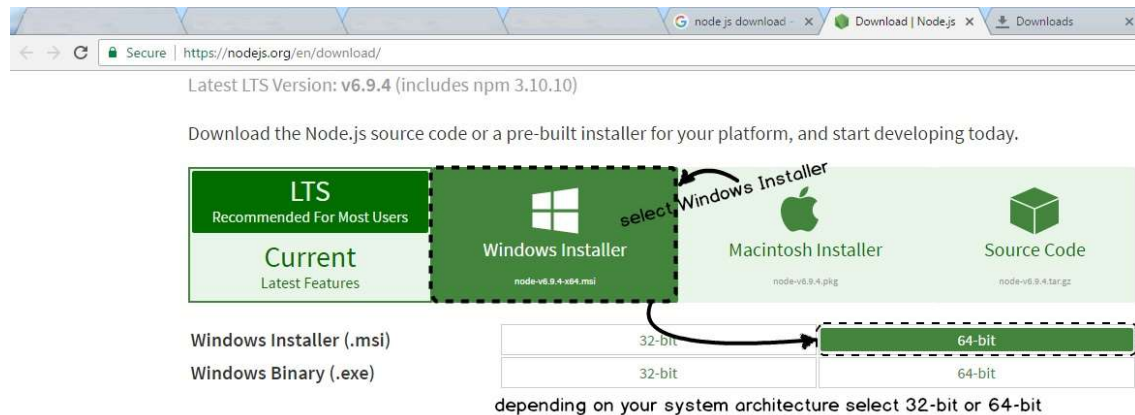
	Once you install node you should see NodeJs command prompt in your program files as shown in the figure.
	We can then open the NodeJS command prompt and fire NPM commands and so on. In case you are completely new to NodeJS please see this NodeJS Video which explains NodeJS in more details.

The third thing for Angular which we need is typescript.   If I put in simple words:-

"TypeScript is a sugar coated Object oriented programming language over JavaScript."

class Customer{
}
TypeScript Code
 	
Transpiling
 	
var Customer = (function () {
    function Customer() {
    }
    return Customer;
}());
JavaScript Code
	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.
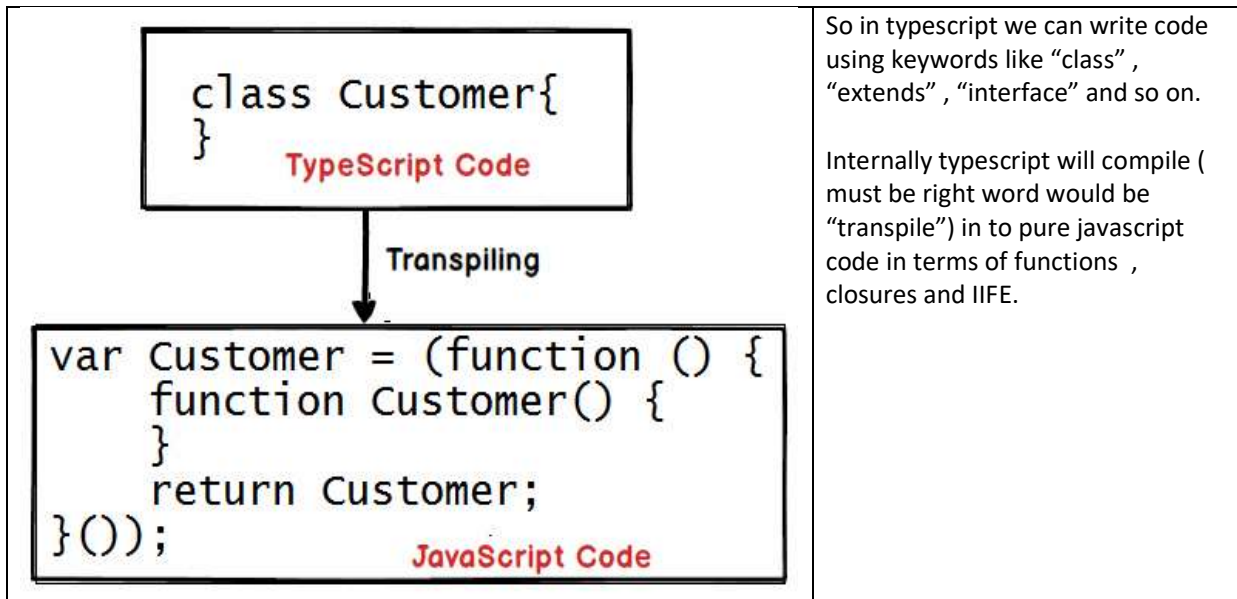
Please do watch this 1 hour Training video on TypeScript which explains Typescript in more detail.

So as we said TypeScript is javascript open source we can get the same from the node. Remember in the previous section we discussed that node has NPM by which we can get latest versions of JS framework.

So click on node command prompt and in the command prompt type "npm install –g typescript". This command will install typescript in your PC and make it available global throughout the computer.

Note :- The "-g" command makes typescript available throughout the computer from any folder command prompt.
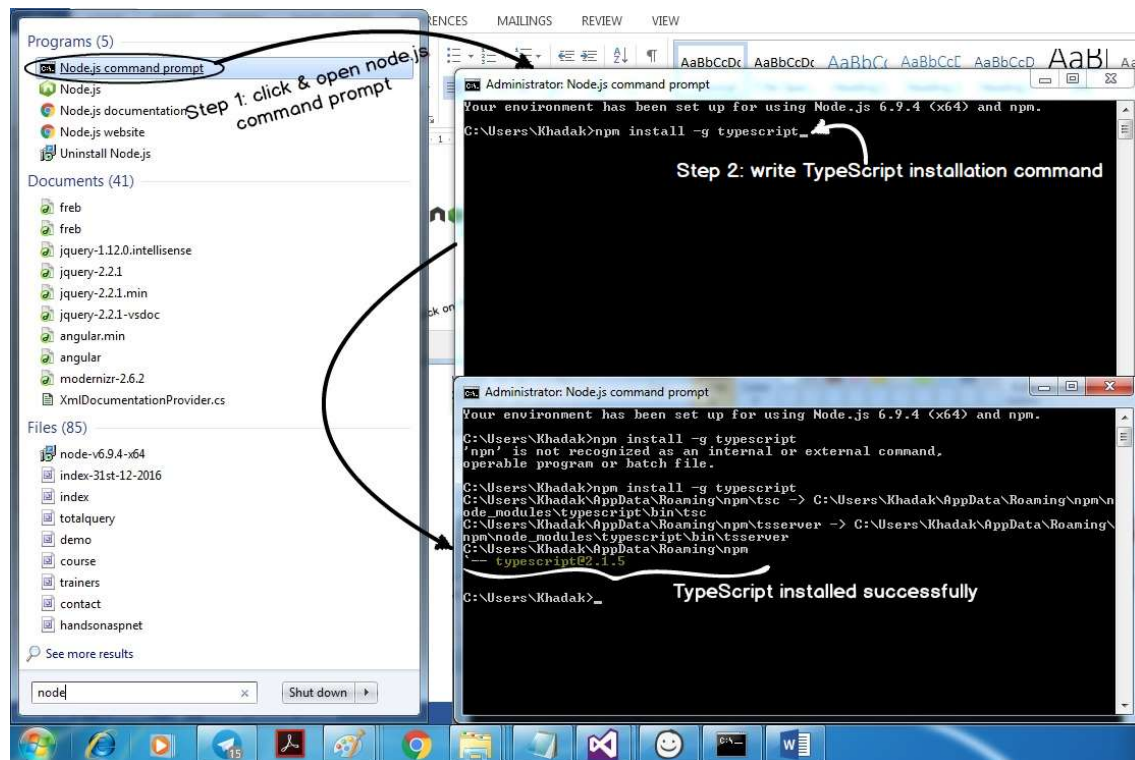
So to start with Angular we need at least these three things one editor we have chosen VS Code , Node (NPM) for getting JavaScript frameworks and Typescript so that we can code JavaScript faster and with Object Oriented coding approach.

With modularity comes great responsibility
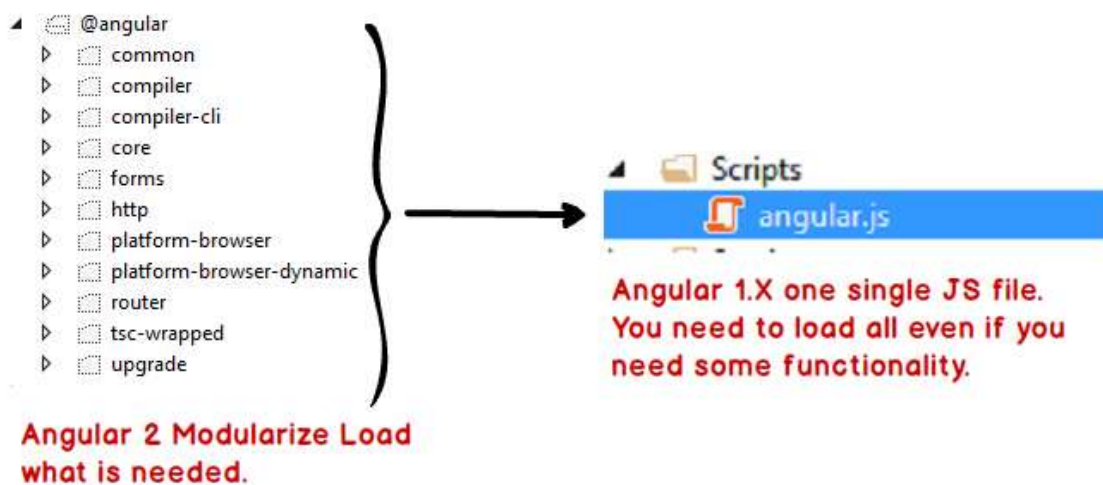
In Angular 1.X we just had [one JS file](#) which had the whole framework. So you drag and drop that single Angular 1.X JS file on HTML page and you are ready with the Angular 1.X environment. But the problem with having whole framework in one JS files was that you have to include all features even if 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. These are self-contained components which can be loaded in an isolated manner rather than loading the whole JS file.

Now because we have lot of self-contained components in other words we have lot of single JS files creating a proper environment itself is a challenge. So let us first understand how to setup angular environment.

Step 2:- Setting up Angular environment

So the first step is to create a folder and open the folder using VS Code as shown in the below image.
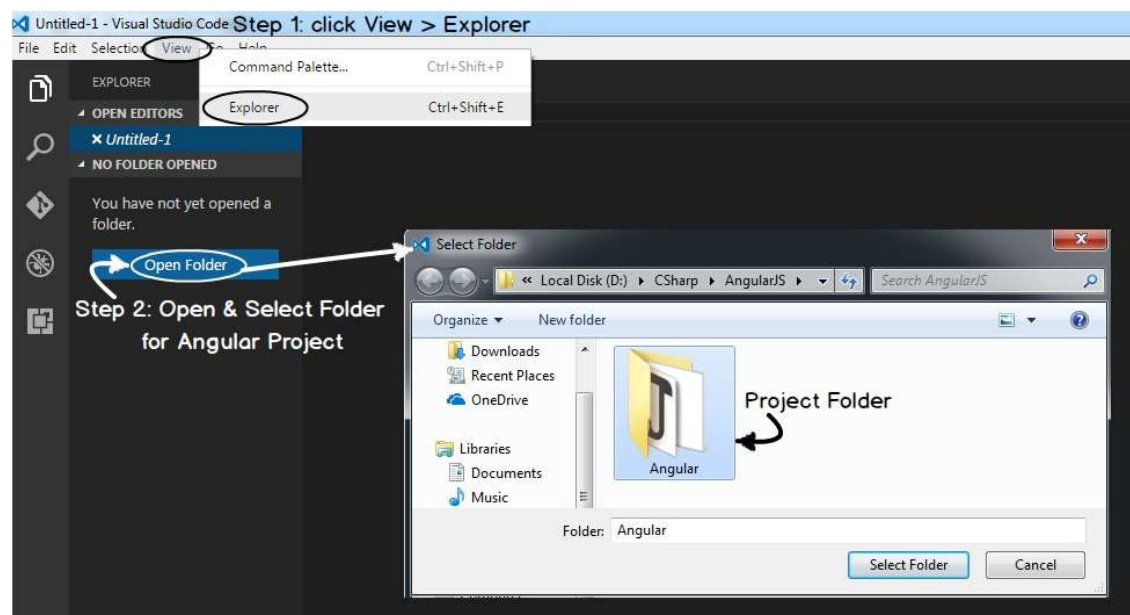
Update the JSON file to angular 4 and remove the typings error.

Inside this folder we need to paste the 4 JSON files. You can download these files from this URL http://tinyurl.com/jeehlqo .
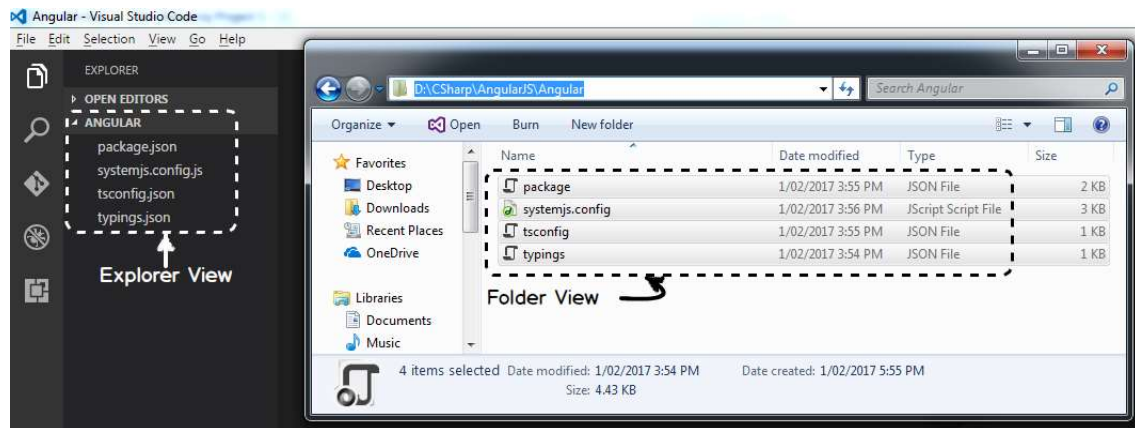
Please note these files are not created by me but I have got them from Angular website or the respective open source website.

Below goes the explanation of those 4 JSON files:-

JSON File Name	What is the use of the file?
package.json	This file has the references of all the Angular modules. Node will use this file to download all the Angular modular files. If you are new to node please go through the pre-requisite video sectio.
tsconfig.json	This is a configuration file for typescript and will be used by typescript to define how transpiling process takes place in typescript. If you are not aware of typescript please go through the pre-requisite videos.
typings.json	Typescript is a new language some of the old JS frameworks cannot be consumed in typescript. For those frameworks we need to define the typings.
systemjs.config	SystemJS is a module loader. This configuration file defines the configuration for SystemJS module loader. We will be talking more about it in the further coming steps.

So now lets open "node" command prompt and type "npm install" in the command line. This command tries to get all the files which are mentioned in the package.json file.
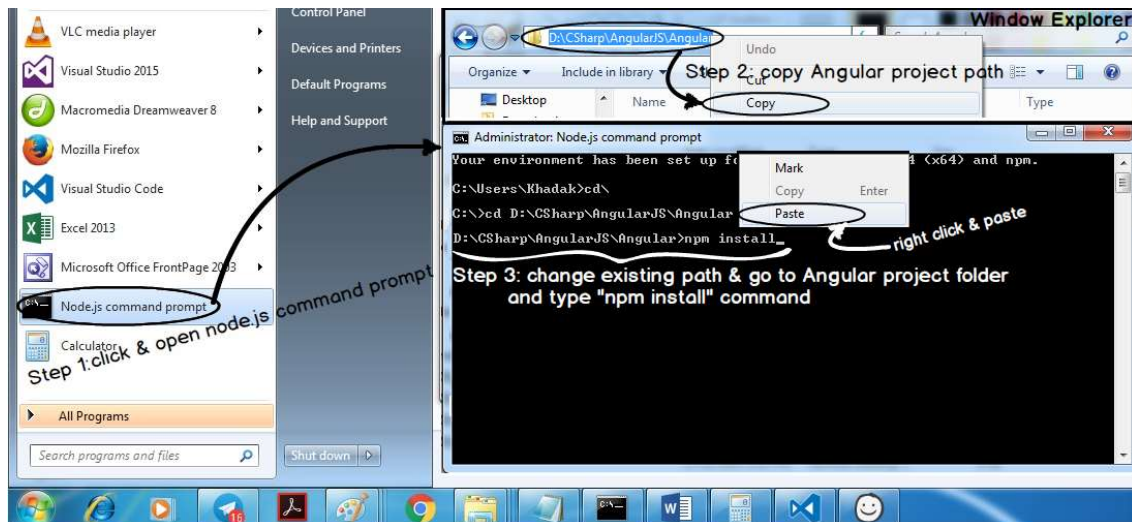
You need to be connected to internet to get these files. If you are not connected to internet you would land up in to some kind of an error as shown in the below figure.
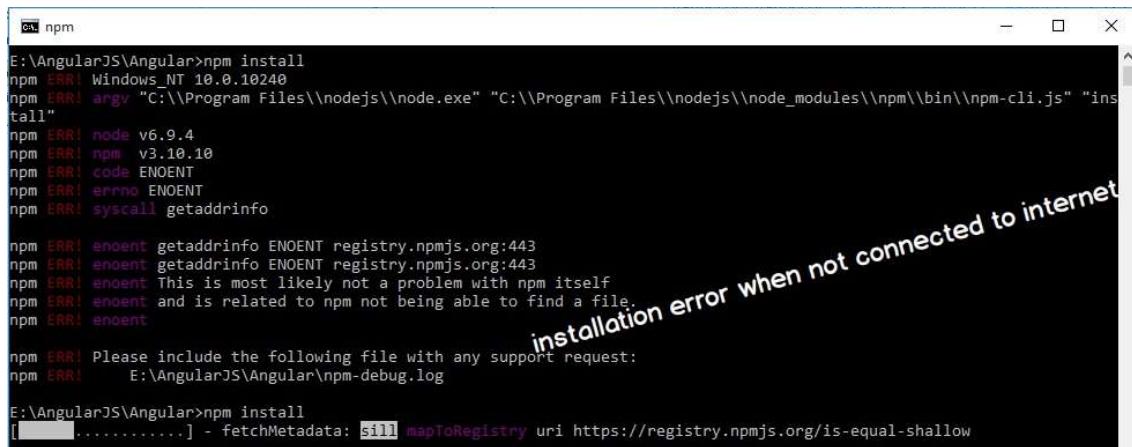
After the Angular installation is done successfully you will see "node_modules" folder created. In this folder all the JS files of Angular has been downloaded.
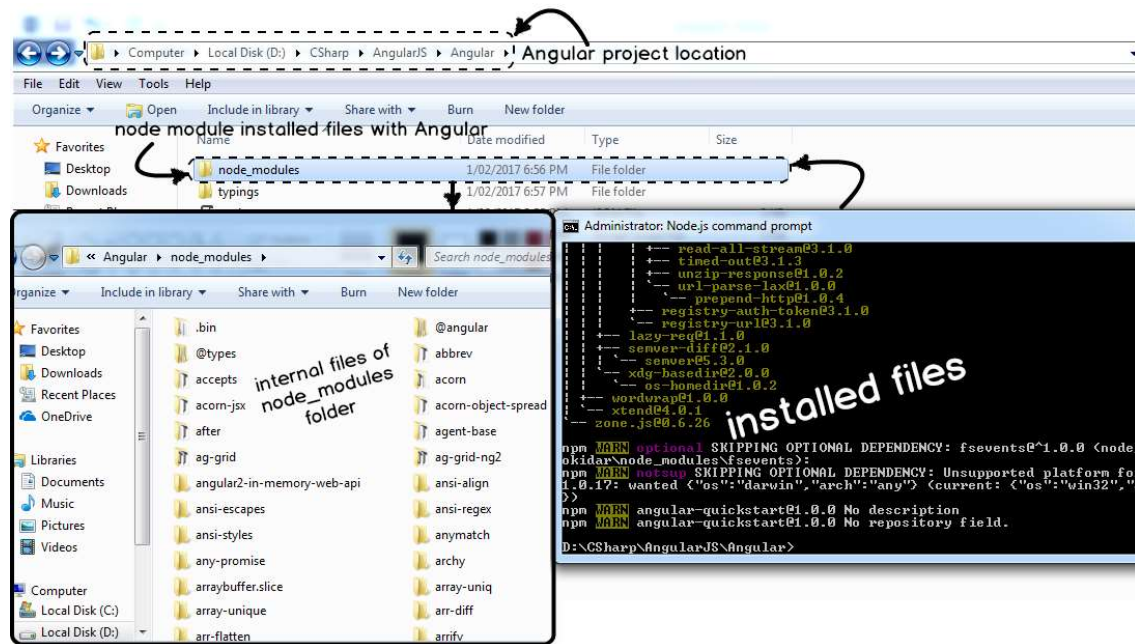
Step 3:- Configuring the task runner

VS Code is just a code editor. It has no idea how to compile typescript code, how to run TSC and so on. So we need to create a GRUNT task which will run TSC command and transpile typescript code to JavaScript.

Press CONTROL + SHIFT + B and click on configure task runner as shown in the below figure.
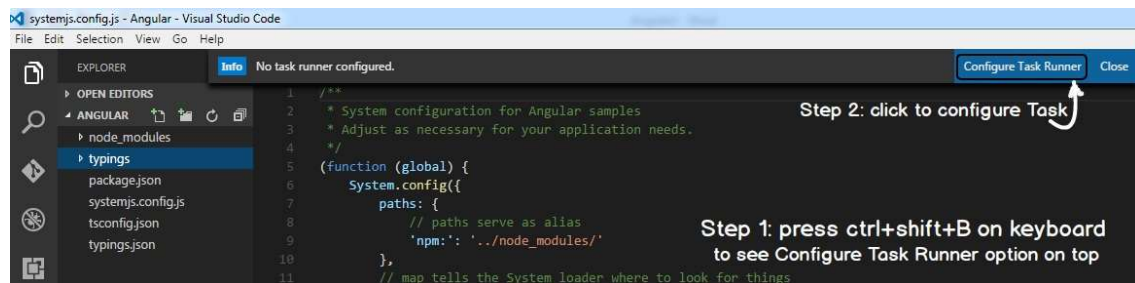
VS Code then pops up lot of task runner options saying what kind of task is it, is it a GRUNT task, GULP task, MSBUILD, MAVEN etc. and so on.
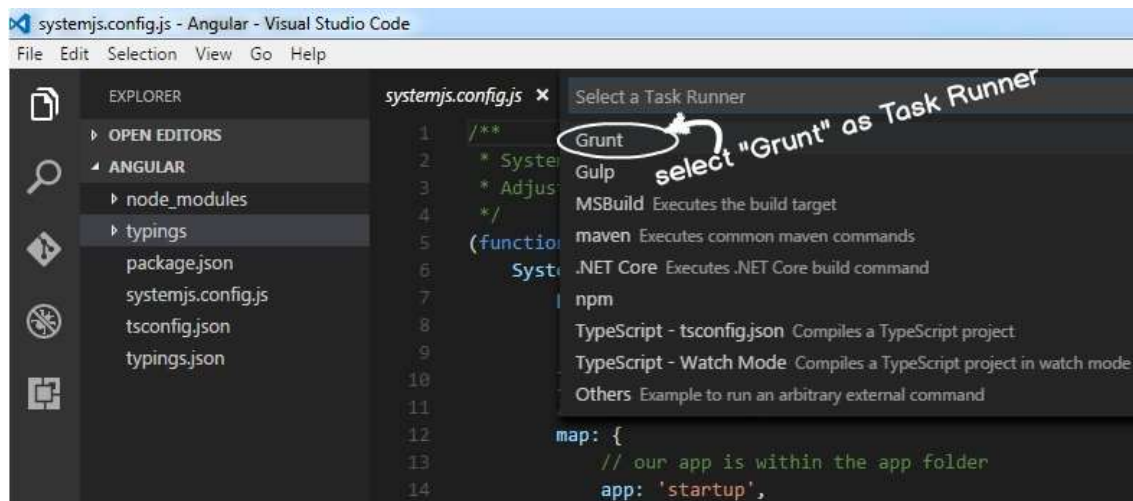
Let's select GRUNT Task and paste the below JSON code. Once you paste it you will see it has created a file called as "tasks.json" file inside ".vscode" folder.
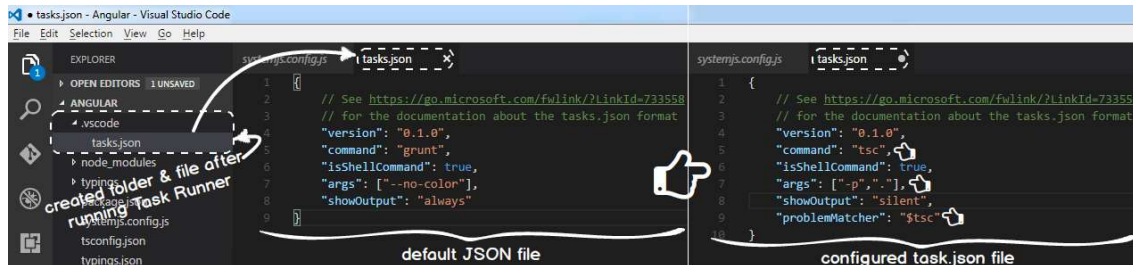
Below is the paste of "tasks.json" file. You can see in the command attribute we have given "tsc" as the command line, the "isShellCommand" specifies that this has to be executed in command line and 'args' specifies the argument.

When typescript will execute it will run using the configuration specified in the tasks.json file.

```
{

    // See https://go.microsoft.com/fwlink/?LinkId=733558

    // for the documentation about the tasks.json format

    "version": "0.1.0",

    "command": "tsc",

    "isShellCommand": true,

    "args": ["-p","."],

    "showOutput": "silent",

    "problemMatcher": "$tsc"

}
```
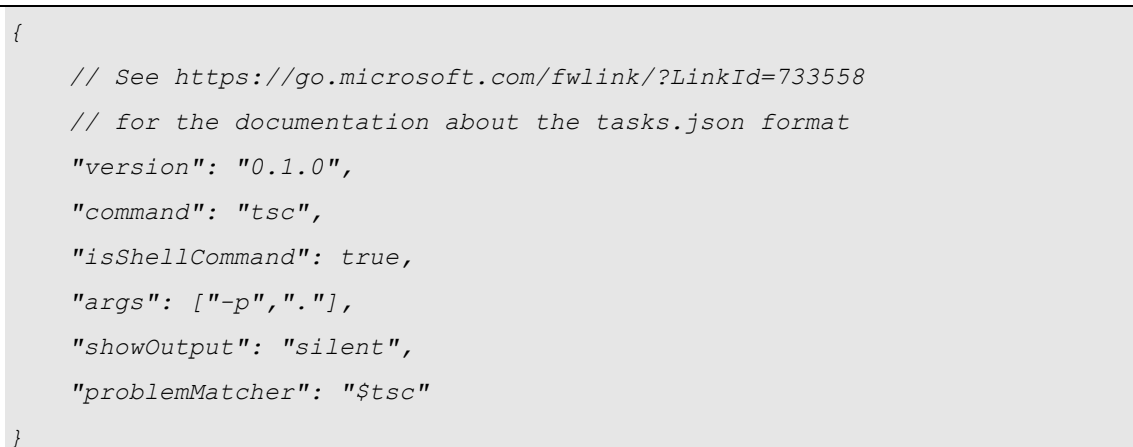
So if you now press CONTROL + B it will build the TS files to JS file.
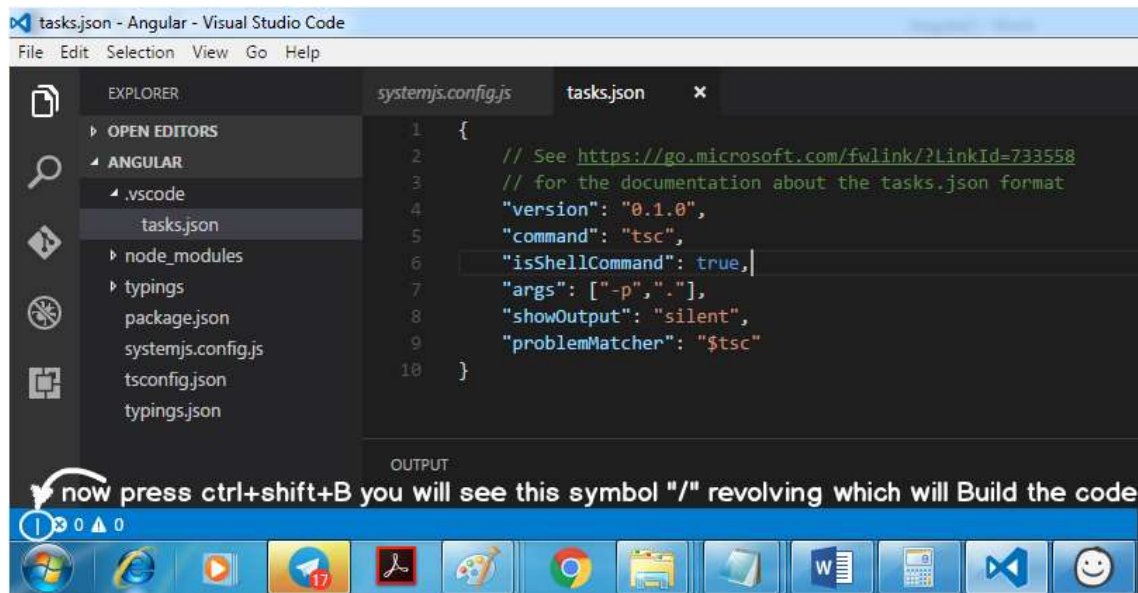
Understanding Angular Component and module architecture

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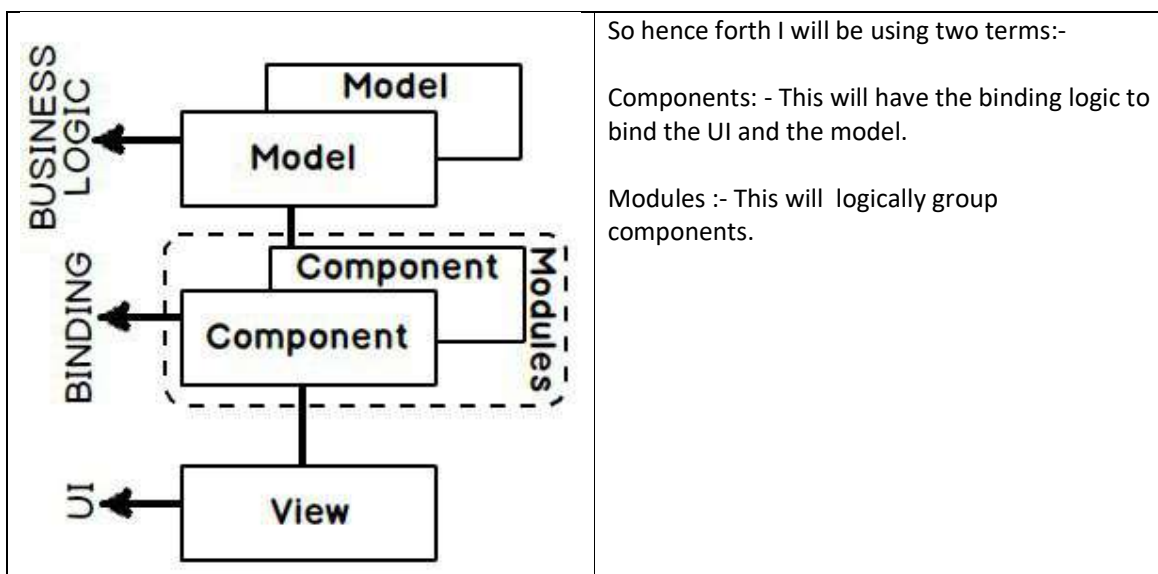
Step 4:- Following MVW Step by Step – Creating the folders

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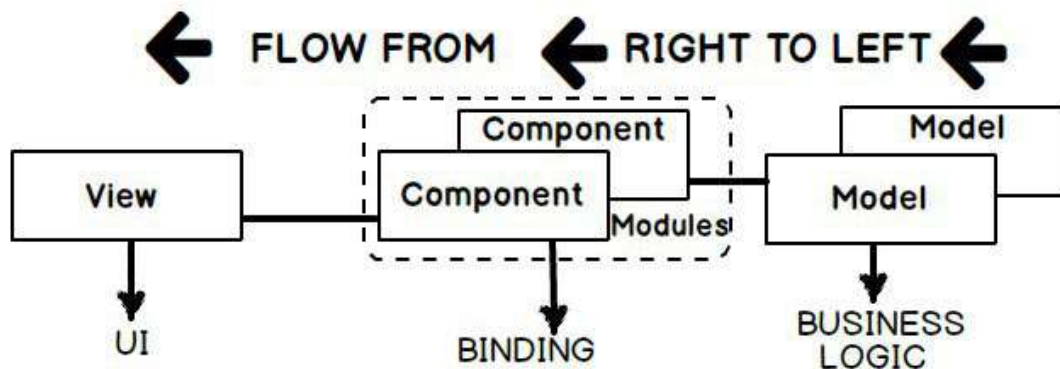
So let's first create four folders in our project:-

- 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.
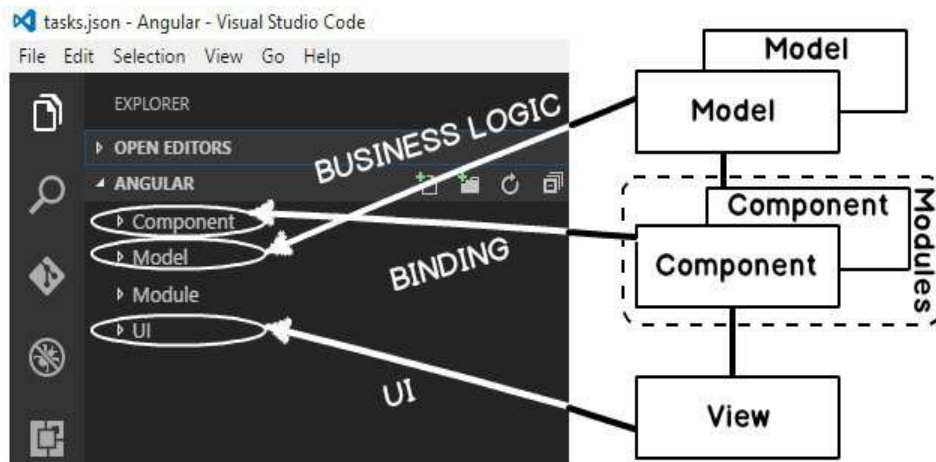
In order to create a folder in VS code you can use the "New folder" icon or you can right click and also create a folder.
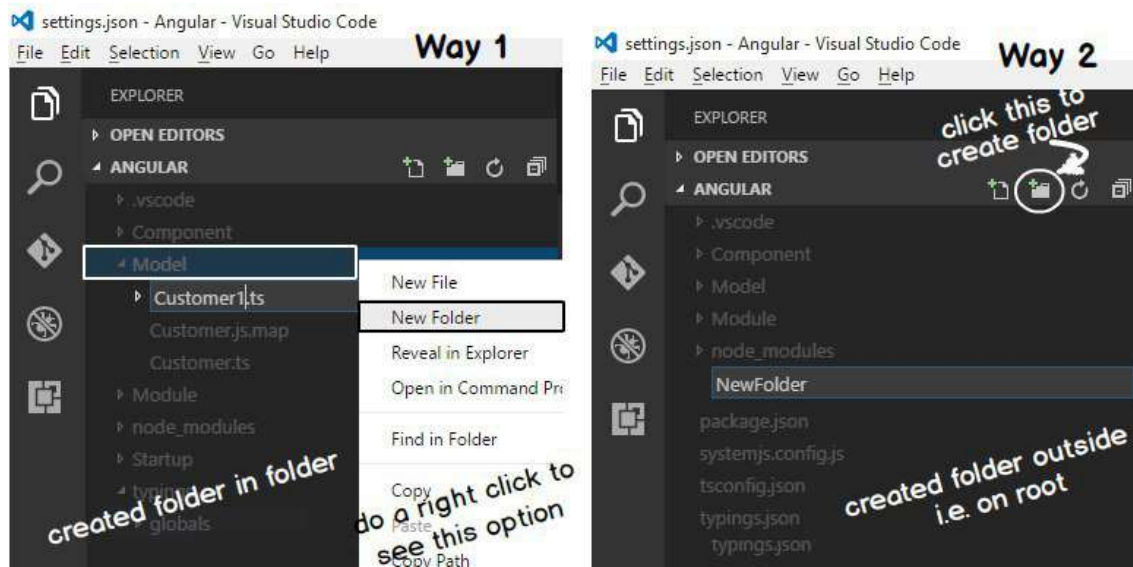
Step 5:- Creating the model

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.

```typescript
export class Customer {

    CustomerName: string = "";

    CustomerCode: string = "";

    CustomerAmount: number = 0;

}
```

Step 6:- Creating the Component
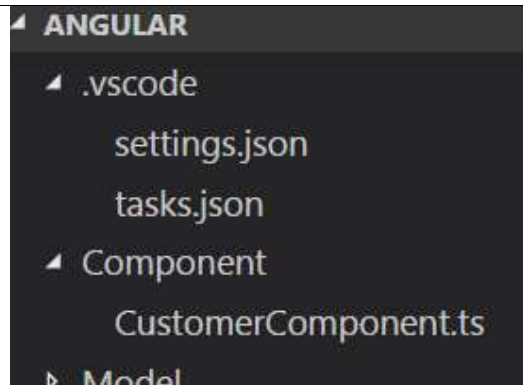
 The next thing we need to code is the binding code. Binding code in Angular is represented by something termed as "COMPONENTS". Angular components has the logic which helps to bind the UI with the model.

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.

```typescript
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.

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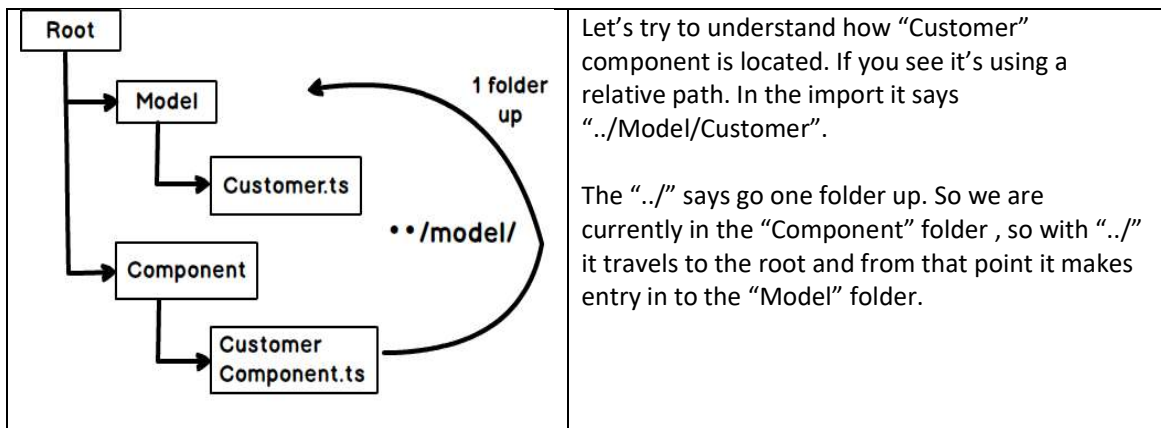
The next import command imports angular core components. In this we have not given any relative path using "../" etc. So how does typescript locate the angular core components ?.

```
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.
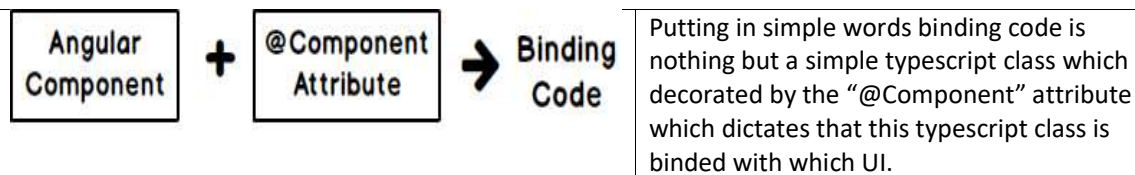
Below goes the full code of the Angular component.

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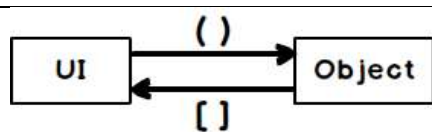
	- Round brackets indicate data sent from UI to object. - Square brackets indicate data is sent from object to UI. - If both are present then it's a two way binding.

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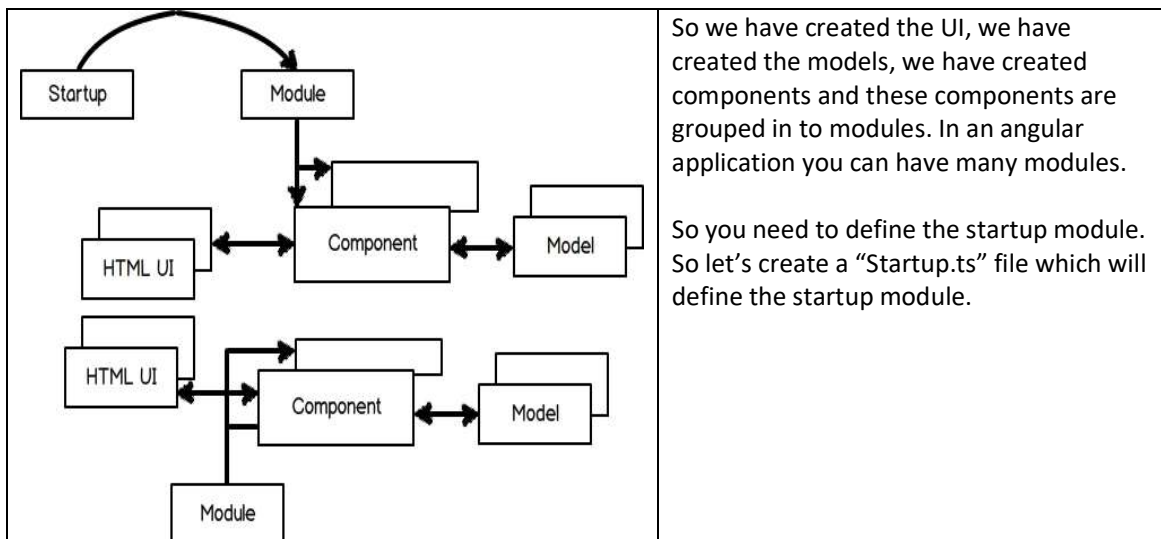
So we have created the UI, we have created the models, we have created components and these components are grouped in to modules. In an angular application you can have many modules.

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>
```

Below are the use 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 in 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"
})
```
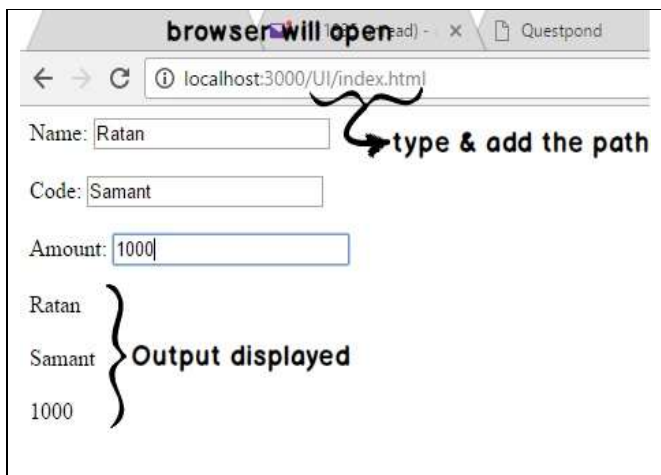
Below goes the full HTML page with all scripts and the place holder tag.

```
<!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:- Installing http-server and running the application

In order to run the application we need a web server. So go to integrated terminal and type "npm install http-server". "http-server" is a simple, zero-configuration command-line http server which we can use for testing, local development and learning. For more details visit https://www.npmjs.com/package/http-server

To run this server we need to type "http" 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 "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.

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. Please read step 2 again to understand how node works.

Lab 2 :- Implementing SPA using Angular routing

Fundamental of Single page application

Now a days 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.	
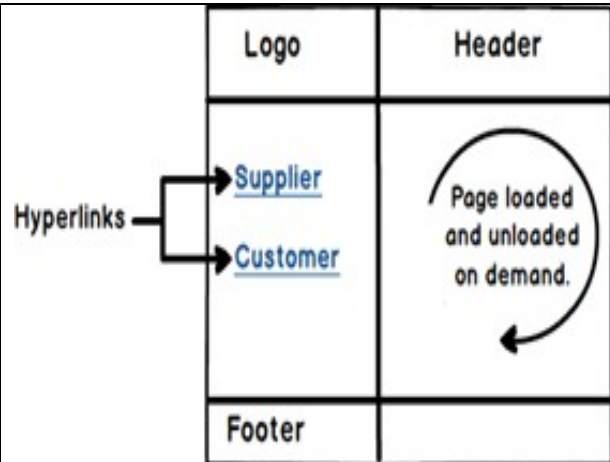
Inclide the bas href step – missed it

Step 1 :- Creating the Master Page

As everything revolves around the Master Page so the first logical step would be is to create the "MasterPage".
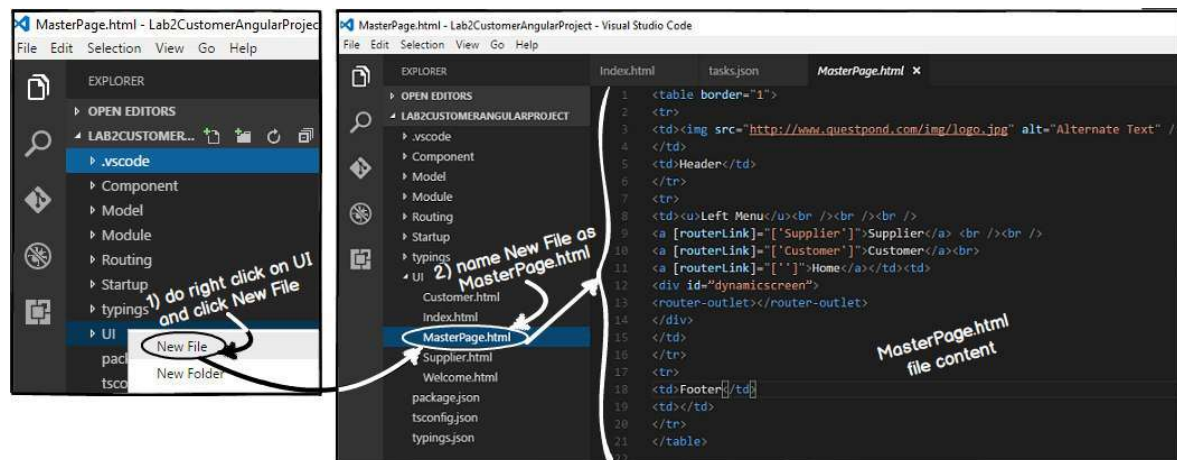
In this master page we will create placeholders for logo , header , menu , footer , copyright and so on. These sections will be loaded only once when the user browses the website first time. And in the later times only pages which are needed will be loaded on demand.

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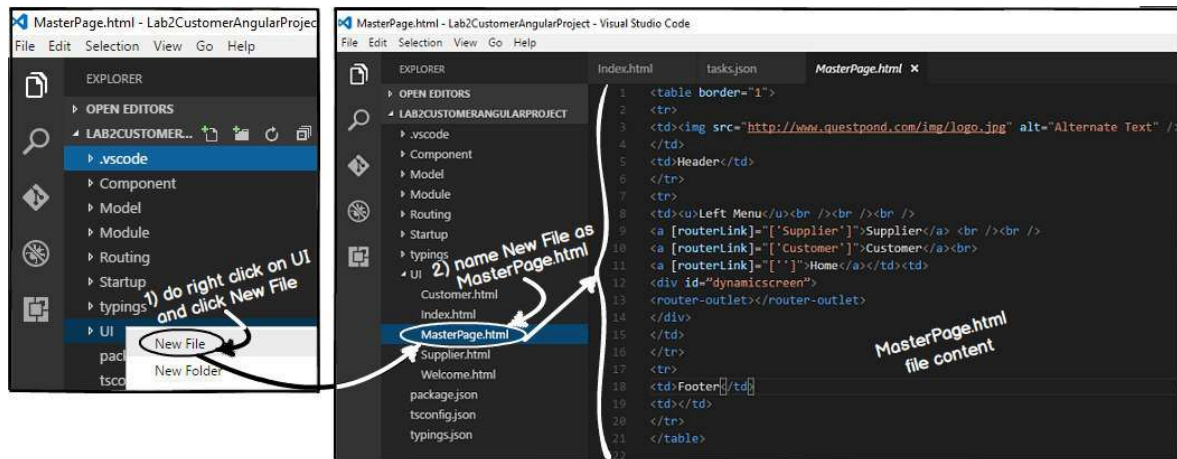
Below is the overall main sections of "MasterPage". Please note the "DIV" section with name "dynamicscreen" where we intend to load screens dynamically. We will fill these sections later.

```
<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.	
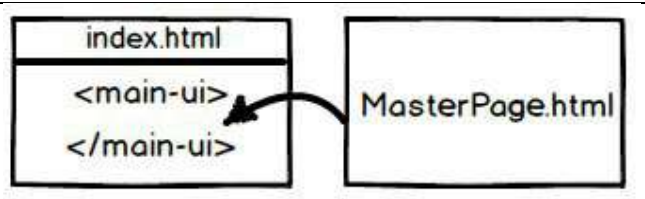
Step 4:- Removing selector from CustomerComponent

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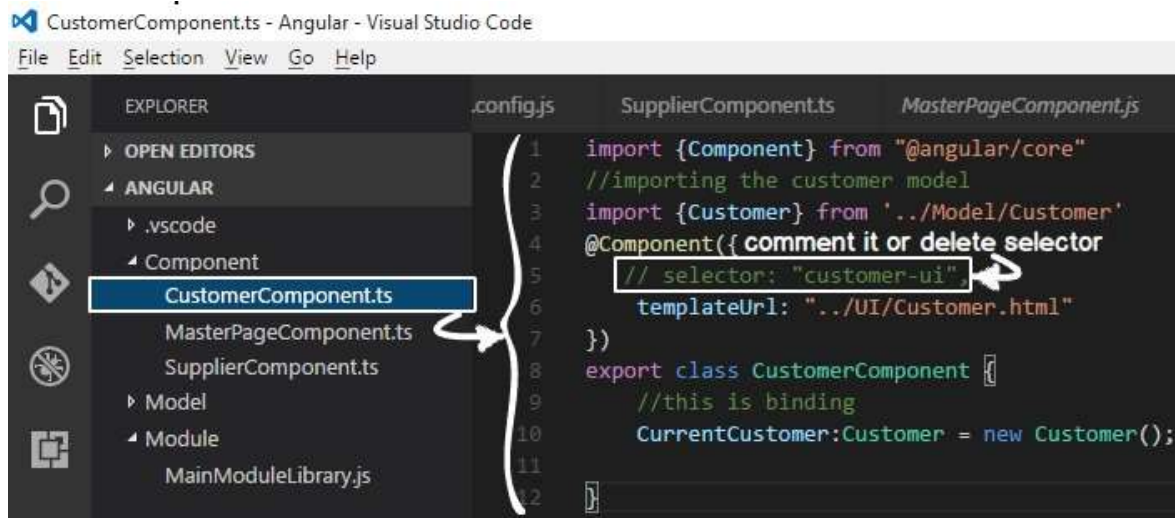
The final code of "CustomerComponent.ts" would look something as show below.

```
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.	
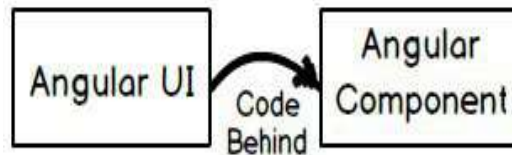
So first let's start with "MasterPage.html" component which we have named as "MasterPageComponent.ts". This master page will get loaded in "Index.html" in the initial bootstrapping process. You can see in this component we have put the selector and this will be the only component which will have the 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 loadedin 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 all the above code we have defined in a separate folder "routing" and in a separate file "routing.ts".
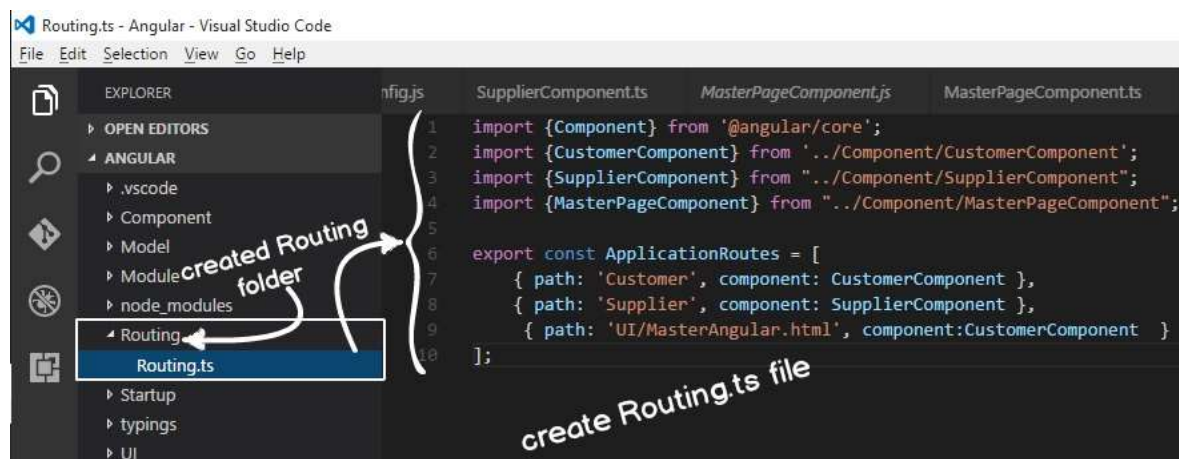
Step 7: - Defining routerLink and router-outlet

The navigation (routes) defined in "Step 6" in the collection needs to 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 path specified in the routes 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 [routerLink]="['Supplier']">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 { }
```
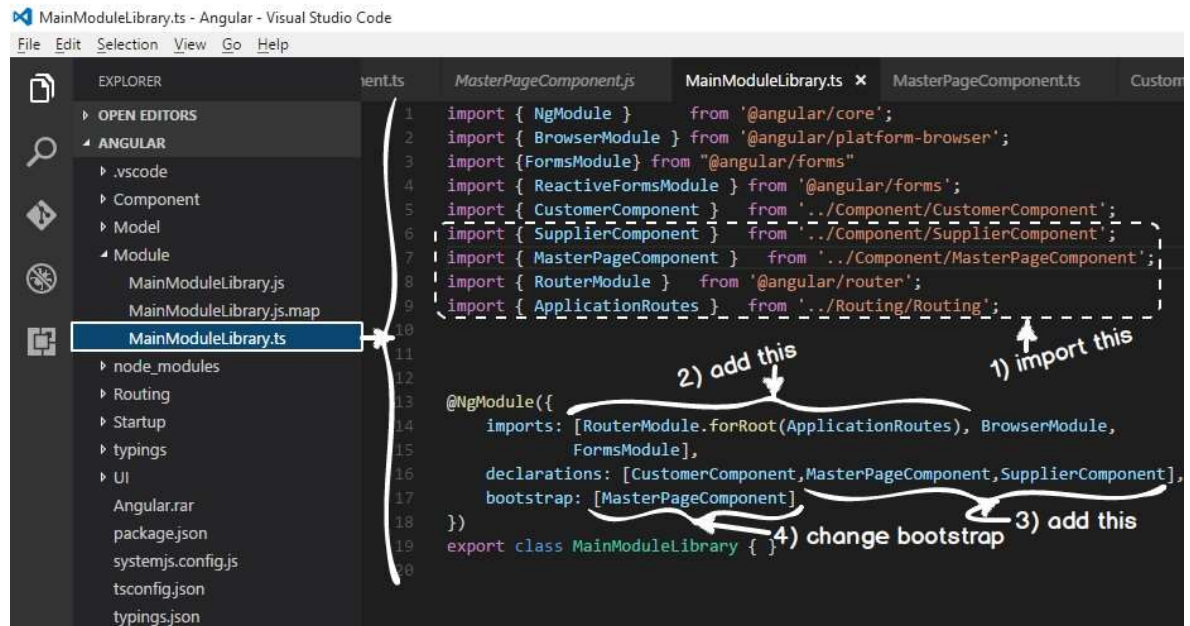

Step 9:- Seeing the output

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.
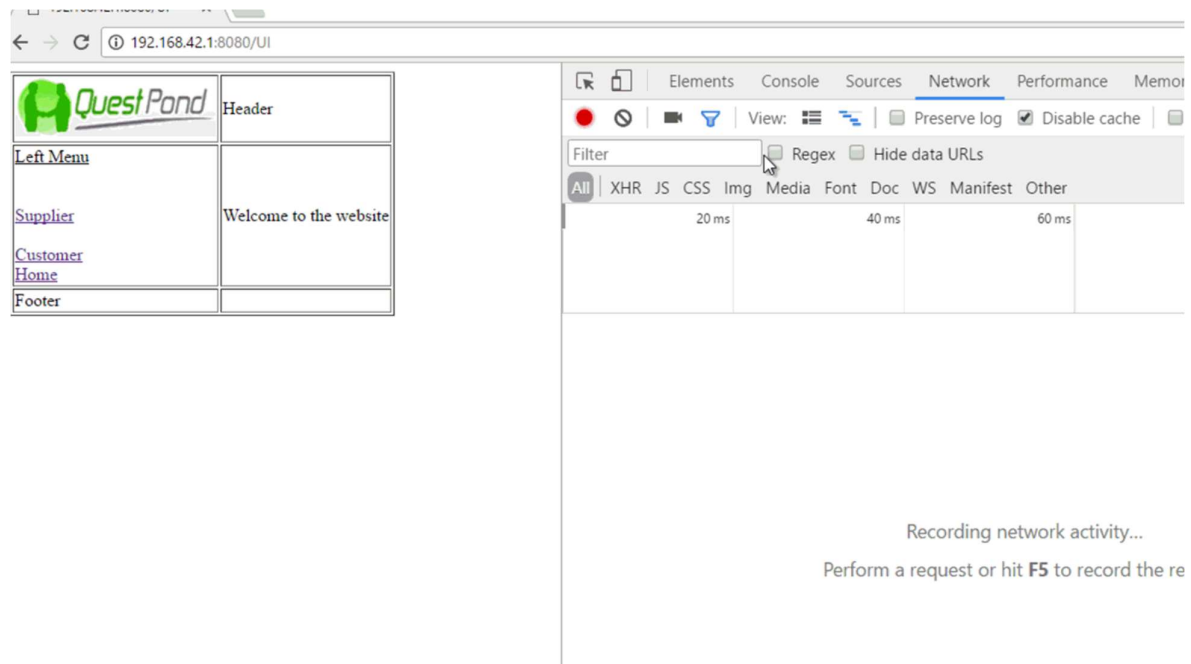
Step 10:- Fixing Cannot match any routes error

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.	192.168.42.1:8080 × ← → C 192.168.42.1:8080/UI/ 192.168.42.1:8080 - Index of / 192.168.42.1:8080/UI/ 192.168.42.1:8080/UI
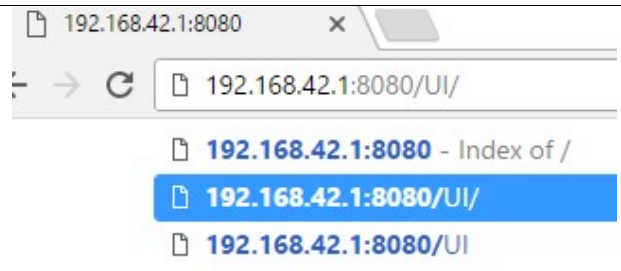
In order to fix the same make one more entry for "UI" path and point it to "WelcomeComponent".

```
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.
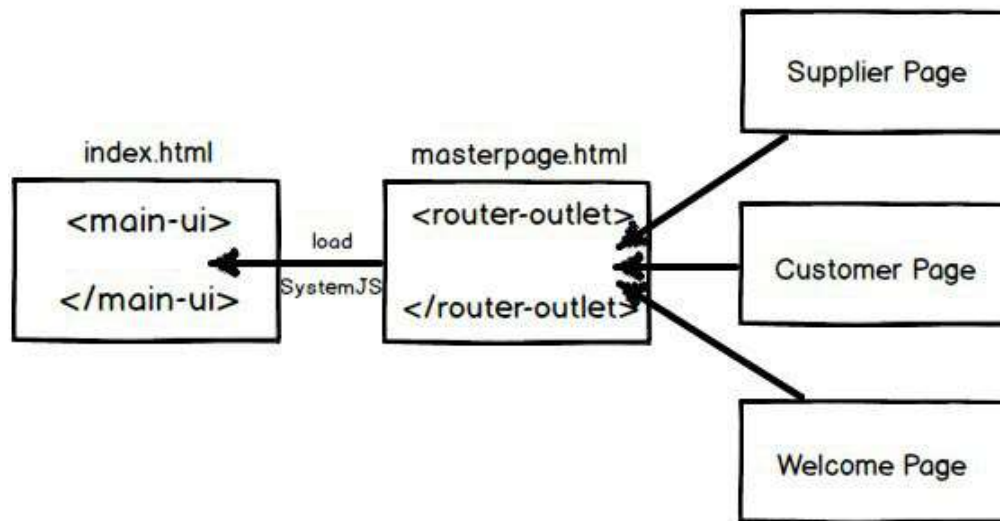
Lab 3 :- Implementing validation using Angular form

Shiv:- Put Reacttiveform import

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.

Form Group → Form Control → Required

Form Control → Required → Length Check
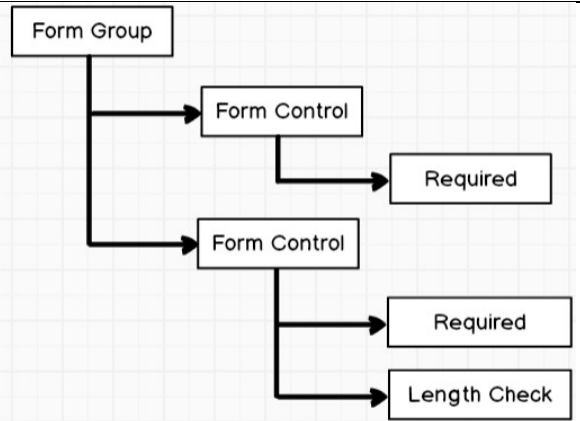
There are 3 broader steps to implement Angular validation

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 an capital alphabet followed by 4 letter numeric.

Note :- Customer code has composite validations.

Where to put Validations ?

Before even we start with validations we need to decide which is the right place to put validations. If you see in Angular we have three broader section 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.
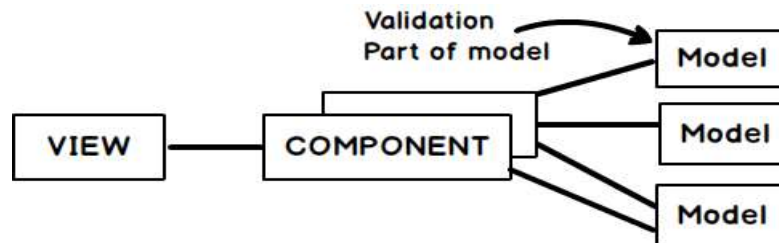
Step 1 :- Import necessary components for Angular validators

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 validation 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 :- 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 6:- 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 7:- 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 8:- Checking individual validations

"CurrentCustomer.formGroup.valid" check 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'].hasEr
ror('required'))">Customer name is required </div>
```

Step 9 :- 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'].hasEr

ror('required'))">Customer name is required </div>

Code:

<input type="text" formControlName="CustomerCodeControl"

[(ngModel)]="CurrentCustomer.CustomerCode"><br /><br />
```

```
<div
[hidden]="!(CurrentCustomer.formGroup.controls['CustomerCodeControl'].hasEr
ror('required'))">Customer code is required </div>
<div
[hidden]="!(CurrentCustomer.formGroup.controls['CustomerCodeControl'].hasEr
ror('pattern'))">Pattern not proper </div>


Amount:
<input type="text"
[(ngModel)]="CurrentCustomer.CustomerAmount"><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 the 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.
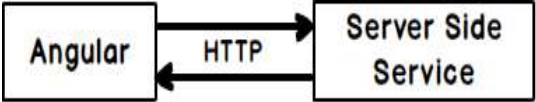
Lab 4 :- Making HTTP calls

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	
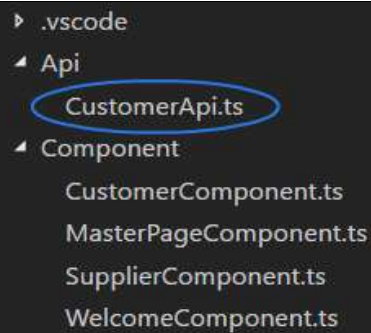
Step 1 :- Creating a fake server side service

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 inmemory web api" has been installed when we did npm. You can check your "package.json" file for the entry of "angular inmemory web api".

So let's create a folder called as "Api" and in that lets 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.	▶ .vscode ▲ Api 　　CustomerApi.ts ▲ Component 　　CustomerComponent.ts 　　MasterPageComponent.ts 　　SupplierComponent.ts 　　WelcomeComponent.ts

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 'angular2-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.
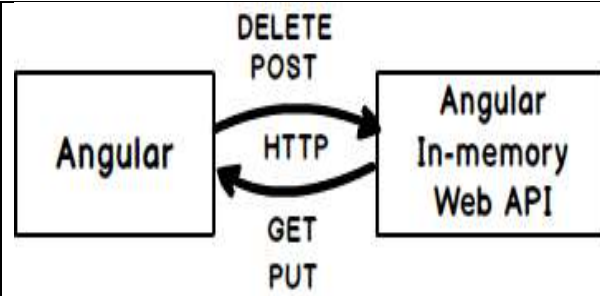
Step 2 :- Importing HTTP and WebAPI module in to main module

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

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.
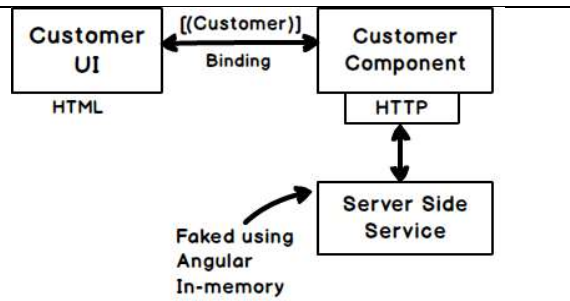
Step 3 :- Importing HTTP in the Customer component

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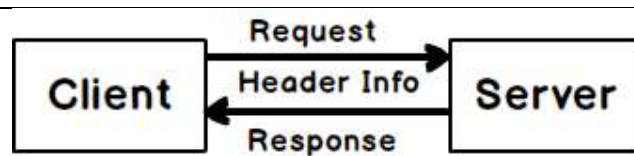
So to create a request we need to first create a header and using that header create a request. One of the header information we need to provide is type of content we are sending to server is it XML , JSON etc.

Below is how to create a 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, its 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 customers 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.
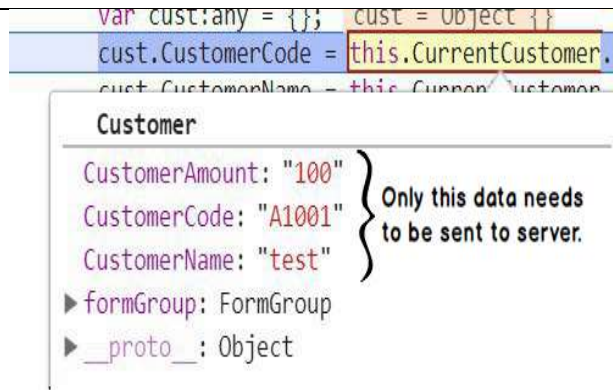
Step 7 :- Connecting components to User interface

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 the customers 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 5 :- 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.	Customer UI, Supplier UI — Reuse — Grid Control (User Control)
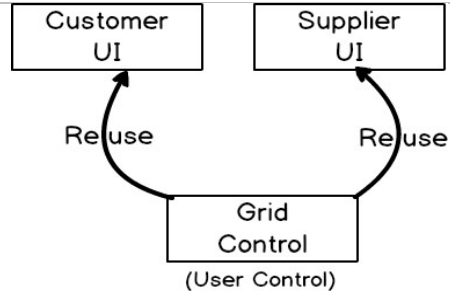
If we want to create a GENERIC reusable grid control you need to think GENERICALLY, you need to think in terms of INPUTS and OUTPUTS. So the first visualization which you should have in your mind is that your GRID control is like a self-contained unit which gets a input of some data collection and when 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.	Customer UI, Supplier UI — Output Selected Customer, Input Customer Collection, Output Selected Supplier, Input (Supplier) Collection — Grid Control (USer Control)
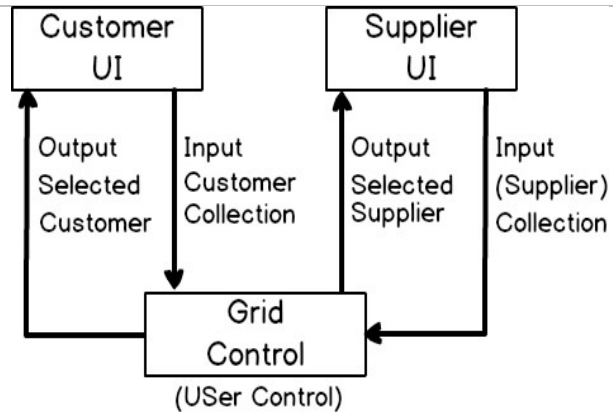
In order to achieve this generic thought process Angular has provided three things Input , Output and Event emitters. Input helps us define the input data to the user control. Output uses event emitters to send data to the UI in which the user control is located.	Output, Input, Event Emitters, User Control
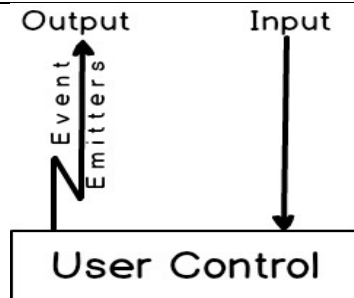
Planning how the component will look like

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 EventEmitter

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.
	▲ Component
	    CustomerComponent.ts
	    GridComponent.ts
	    MasterPageComponent.ts
	    SupplierComponent.ts
	    WelcomeComponent.ts

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 eventemitter 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 store 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>();

}
```

```
// 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;

        }
```

```
}
```

Step 4 :- Defining Event emitters

```typescript
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 2 :- Creating UI for the reusable component

```html
<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 3 :- Consuming the component in the customer UI

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

Step 4 :- Creating events in the main Customer component

```typescript
import {Component} from "@angular/core"
//importing the customer model
import {Customer} from '../Model/Customer'
import { Http, Response, Headers, RequestOptions } from '@angular/http';
@Component({
    selector: "customer-ui",
    templateUrl: "../UI/Customer.html"
})
export class CustomerComponent {
    //this is binding
    CurrentCustomer:Customer = new Customer();
    Customers: Array<Customer> = new Array();

    constructor(public http:Http){
            this.Display();
    }
     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();
    }
    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));
    }
    Select(_selected:Customer) {
        this.CurrentCustomer.CustomerAmount =_selected.CustomerAmount;
        this.CurrentCustomer.CustomerCode = _selected.CustomerCode;
        this.CurrentCustomer.CustomerName = _selected.CustomerName;
    }
    Success1(res) {
        this.CurrentCustomer = new Customer();
    }
    Error(err) {
        console.debug(err.json());
    }
    Success(res) {
        this.Customers = res.json().data;
    }
}
```

Step 5 :- Defining the component in the mainmodule

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

import { RouterModule }   from '@angular/router';
import { ApplicationRoutes }   from '../Routing/Routing';
import { InMemoryWebApiModule } from 'angular2-in-memory-web-api';
import {CustomerApiService} from "../Api/CustomerApi"
import { HttpModule } from '@angular/http';

@NgModule({
    imports: [RouterModule.forRoot(ApplicationRoutes),
        InMemoryWebApiModule.forRoot(CustomerApiService),
            BrowserModule,ReactiveFormsModule,
            FormsModule,HttpModule],
    declarations: [CustomerComponent,
                MasterPageComponent,
                SupplierComponent,
```

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

Acronym used in this book

- NPM :- Node package manager.
- TS :- TypeScript.
- JS :- Javascript.
- VS :- Visual studio.
- VS Code :- Visual studio code.
- WP :- Webpack
- OS :- Operating system.
- SPA :- Single page application.

TBD (To be done)

The following things we are still working and hope to get this completed soon.

- Lazy loading and routing chapter is needed.
- Injectable component chapter is needed.
- How to use webpack
- Jquery with Angular one topic
- ViewChildren topic
- Book should start from node , typescript and basic javascript lessons.
- Book should have final interview question revision
- A discussion about should we learn Angular 2 or 4 should be discussed.