Contents

[**part 01 – NG14 Setup** 2](#_Toc121653532)

[**part 02 – Configuring New App** 4](#_Toc121653533)

[**part 03 – Routing** 6](#_Toc121653534)

[**part 04 – JSON Server** 7](#_Toc121653535)

[**part 05 – Add Bootstrap** 10](#_Toc121653536)

[**part 06 – NG14 Forms** 10](#_Toc121653537)

[**part 07 – Validation** 15](#_Toc121653538)

[**part 08 – Fixing the HTML and Complete Routing** 16](#_Toc121653539)

[**Appendix A – Install Angular 14 on Linux Ubuntu 20** 17](#_Toc121653540)

[**Appendix B – Angular Architectural Concepts** 18](#_Toc121653541)

[**Appendix C – Angular Directives** 20](#_Toc121653542)

[**Appendix D – Installation Issues** 21](#_Toc121653543)

[**Appendix D – @NgModule** 22](#_Toc121653544)

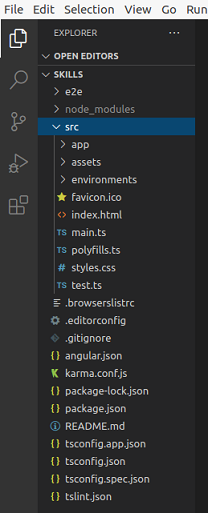
[**Appendix D –** Watchpack Error 22](#_Toc121653545)

[**Appendix E –** Path Match 22](#_Toc121653546)

Day04 Introduction to NG 14

# part 01 – NG14 Setup

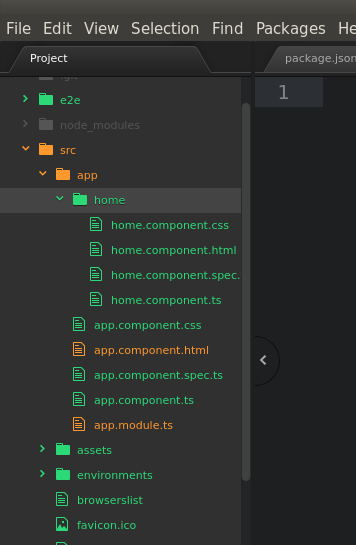
This section assumes that you have already install the latest Angular CLI. If you did not, please run the command **npm install -g @angular/cli** before proceeding.

1. From your root folder (Documents in my case), open a terminal window (or tab) to that folder and type the command **ng new skills**
2. Choose **N** for stricter type checking (if asked) and **Y** for routing. Choose plain **CSS** for styles. To choose CSS use the arrow keys on the keyboard, however CSS should be auto selected, just hit **Enter**.
3. CD into the skills directory and open a terminal window pointing to that folder. After typing **ng serve** in the terminal window, notice the word ‘compiling’, the type script (ts) code is compiled in order to run successfully. (**or ng serve -o**)
4. Open the application in VS Code (or another editor) and most of our work will be in the **app** folder, which acts like the parent folder.

|  |
| --- |
| **<span>{{ title }} app is running!</span>**  **<router-outlet></router-outlet>** |

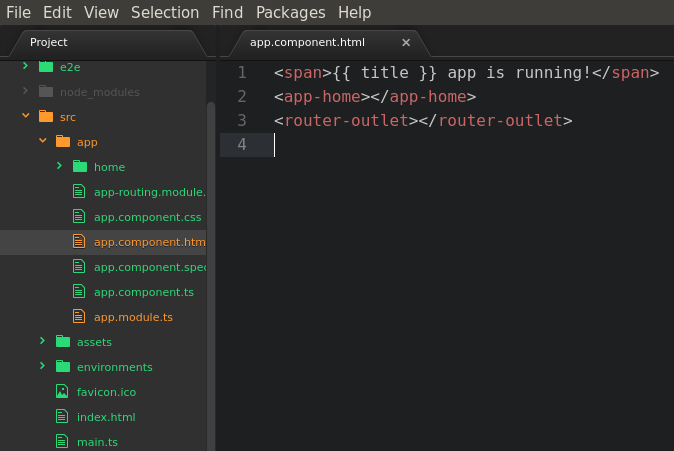
View the code of app.component.html, this is the file that feeds the default page that shows up on the browser at port 4200. Remove everything except the <span> tag which has the code **{{title}} app is running!** And the <router-outlet> tags. (around line 343)

Leave these two lines in app.component.html

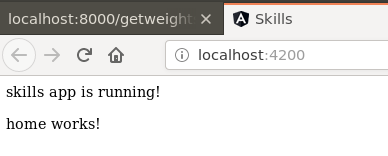
1. From a terminal window, CD into the skills folder, if you have not done so already. Your terminal must be pointing to your skills folder. Once there, type **ng generate component home.** Note, if you have the app already running, just open a new tab or a second terminal window.
2. Back on VS Code, you should see a new folder called “home”  
   

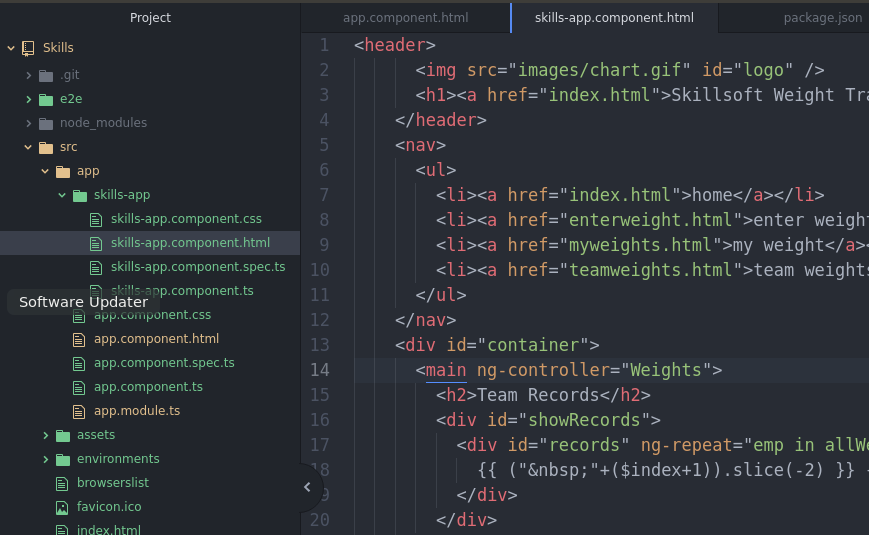
# part 02 – Configuring New App

1. Open home.component.html file and notice that a line of code was already inserted automatically. This .html file is also referred to as the *template* in Angular’s world.
2. Go back to the parent folder and look at the file app.component.ts there is a **selector** property with a value of **app-root**. Now open index.html under the original **app** folder and notice that between the **body** tags, this name appears between angle brackets. This means that where-ever Angular sees the directive **<app-root>** in the HTML, it will replace **<app-root>** with the template being pointed to, so app.component.html in this case.
3. Now go back to our component and inside of home.component.ts file there is a **selector** with the name **app-home**, lets insert this name in the app.component.html file with angle brackets. So now, angular is building up components to deliver the main html page with the **<app-root>** directive.



1. Once you save, the browser will reload the new page



1. We will use the original index.html file from a previous bootcamp as an example for the next step. Open that page in an editor and copy everything between the **<body>** tags. Paste all this code in the home.component.html file **replacing** the original text that was there.  
     
   

During copy, do not copy the actual <body> tags and be careful to not copy any JavaScript tags.

1. Copy the image from the original images folder into the assets folder of the new NG7 skills app. Change the image link in home.component.html file accordingly

|  |
| --- |
| **<img src="./assets/chart.gif" id="logo" />** |

1. Copy the original **css** file and replace the NG14 css file in the **src** folder
2. Finally for this part, remove the *skills app is running* text from the parent app, app.component.html, also fix any path issues for the image.

# part 03 – Routing

At this point we do not have true routing, we hard coded our home page to show up on the first hit to :4200, we will fix this by changing app-routing.ts

1. In app-routing.module.ts import the **HomeComponent** component

|  |
| --- |
| **import { NgModule } from '@angular/core';**  **import { Routes, RouterModule } from '@angular/router';**  **import { HomeComponent } from './home/home.component';**  **const routes: Routes = [];** |

1. Enter the home route in the Routes[] array as a JavaScript object

|  |
| --- |
| **import { HomeComponent } from './home/home.component';**  **const routes: Routes = [**  **{ path: 'home', component: HomeComponent }**  **];** |

1. Now we can remove the **<app-home>** element from app.component.html. So this file should only have the **<router-outlet>** element, nothing else. However when the Angular home page refreshes the content is gone. So go to <http://localhost:4200/home> and it should return
2. While we are on this topic lets add a default route, so if the user goes to just :4200, they should see the home page. So back in app-routing.module.ts add the default path as shown below:

|  |
| --- |
| **import { HomeComponent } from './home/home.component';**  **const routes: Routes = [**  **{ path: '', redirectTo: '/home', pathMatch: 'full' },**  **{ path: 'home', component: HomeComponent }**  **];** |

Notice the comma after the first path. Also, it is **above** the home path.

# part 04 – JSON Server

1. We would need a mock server so that we can make API calls. Install the JSON Server using the command: **npm install json-server --save-dev**

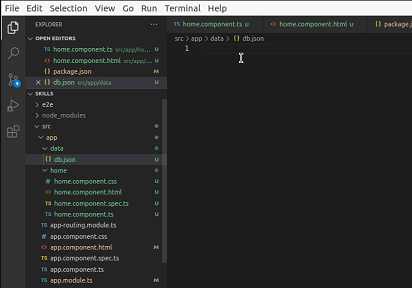
Make sure that you are in the skills folder when you do this.

1. This server will need a database file to work with, lets **simulate** a database with just a plain text file holding JSON data in it. First we need to configure our package.json file to run our server. In package.json go to the scripts section and add a new script as shown below:

|  |
| --- |
| **"version": "0.0.0",**  **"scripts": {**  **"ng": "ng",**  **"start": "ng serve",**  **"build": "ng build",**  **"test": "ng test",**  **"lint": "ng lint",**  **"e2e": "ng e2e",**  **"server": "json-server --watch ./src/app/data/db.json"**  **},**  **"private": true,** |

Remember to insert a comma at the line above. Notice that the server is watching a file called db.json, we will create this file and its path shortly.

1. Back in the editor, create a new folder under the app folder called data. Inside of the data folder create a new text file called db.json



1. There are several ways to create json files but lets follow the example below. First we name our mock database, *employees* in this case and then point that to an array of employees:

|  |
| --- |
| **{**  **"employees":[ ]**  **}** |

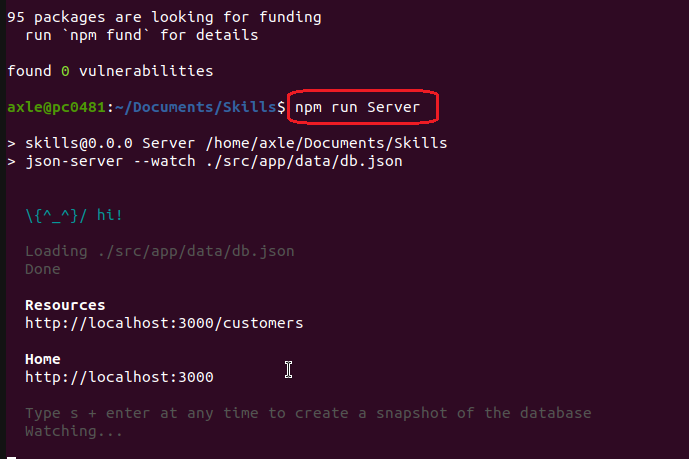
Note, you must have the enclosing curly braces to encapsulate the employees

1. Enter the first employee like this (use your own name but the password can be something simple):

|  |
| --- |
| **{**  **"employees":[**  **{**  **"username":"Axle",**  **"password":"1234"**  **}**  **]**  **}** |

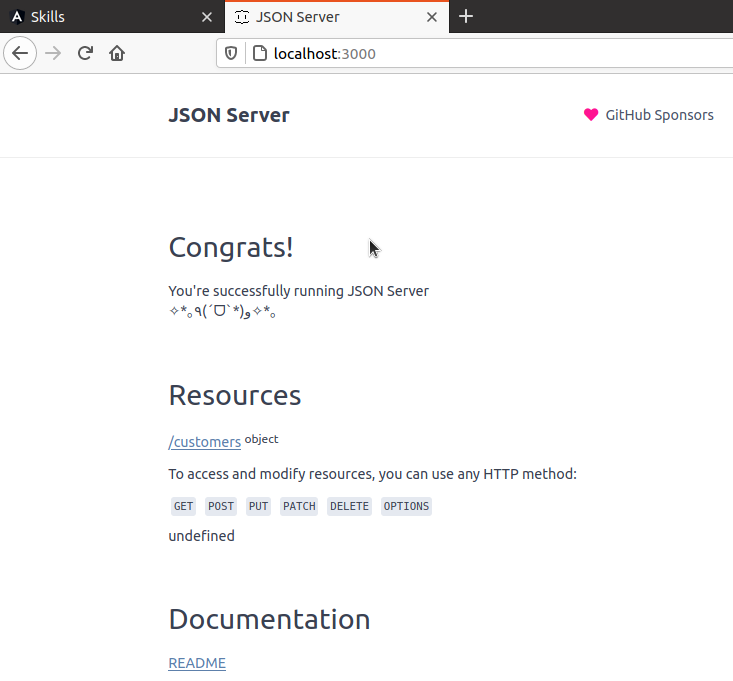
This is NOT a secure site, yet, so use simple passwords for now.

1. Start the server by going back to a terminal window and run the command:  
   **npm run Server**



The computer should respond with a message and a location of where it can be accessed with a browser, localhost:3000 in this case

1. Now go to that location using a browser

  
Notice that under **Resources** the server found our *employees* database

Note to stop any process in Linux just hold down the CTRL button and then hit the letter C on the keyboard.

# part 05 – Add Bootstrap

1. Run the following command in a terminal window to add Bootstrap, make sure your terminal window is pointing to your application folder:  
   **npm install bootstrap**

Then install jQuery in the same manner

**npm install jquery**

1. Make the following changes to the angular.json file:

|  |
| --- |
| **"assets": [**  **"src/favicon.ico",**  **"src/assets"**  **],**  **"styles": [**  **"src/styles.css",**  **"node\_modules/bootstrap/dist/css/bootstrap.css"**  **],**  **"scripts": [**  **"node\_modules/jquery/dist/jquery.min.js",**  **"node\_modules/bootstrap/dist/js/bootstrap.js"**  **]**  **},**  **"configurations": {**  **"production": {** |

# part 06 – NG14 Forms

We will create a simple register form and a login form to test our server and at the same time learn valuable skills in Angular 14

1. Create a new *register* component like you did for the *home* component, so from a terminal window:  
   **ng g c register**
2. Copy all the code from [home.component.html](http://home.component.html) file into register.component.html. Remove all of the code between the <main> tags. Replace all the code you removed with the code highlighted below:

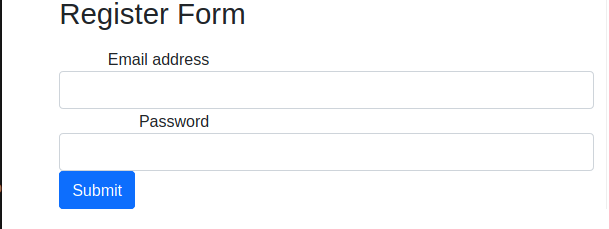
|  |
| --- |
| **<form>**  **<div class="form-group">**  **<label for="username">User name</label>**  **<input type="text" class="form-control" id="username">**  **</div>**  **<div class="form-group">**  **<label for="password">Password</label>**  **<input type="password" class="form-control" id="password">**  **</div>**  **<button type="submit" class="btn btn-primary">Submit</button>**  **</form>** |

1. Follow the steps from Part03 to add a new route to your register component:

|  |
| --- |
| **import { HomeComponent } from "./home/home.component";**  **import { RegisterComponent } from "./register/register.component";**  **const routes: Routes = [**  **{ path: '', redirectTo: '/home', pathMatch: 'full' },**  **{ path: 'home', component: HomeComponent },**  **{ path: 'register', component: RegisterComponent }**  **];**  **@NgModule({** |

1. Wrap the form inside of a pair of <div> tags and add the class of container to this new div, also add an <h2> tag with class of pb-2 and give it a title:

|  |
| --- |
| **</nav>**  **<div id="container">**  **<main>**  **<div class="container">**  **<h2 class="pb-2">Register Form</h2>**  **<form>**  **<div class="form-group">** |

Note, that the name of the fields on this form match the database.   
  


1. Add the **ReactiveFormsModule** module to the application from the app.module.ts file:

|  |
| --- |
| **import { RegisterComponent } from './register/register.component';**  **import { ReactiveFormsModule } from '@angular/forms';**  **@NgModule({**  **declarations: [**  **AppComponent,**  **HomeComponent,**  **RegisterComponent**  **],**  **imports: [**  **BrowserModule,**  **AppRoutingModule,**  **ReactiveFormsModule**  **],**  **providers: [],** |

1. In the register.component.ts file, import the FormGroup, FormControl, Validators modules from @angular/forms:

|  |
| --- |
| **import { Component, OnInit } from '@angular/core';**  **import { FormGroup, FormControl, Validators} from "@angular/forms";**  **@Component({**  **selector: 'app-register',** |

1. Still in register.component.ts file, add a property to the form of the FormGroup type

|  |
| --- |
| **export class RegisterComponent implements OnInit {**  **frmRegister: FormGroup;**  **constructor() { }** |

Note: the IDE may show an error here but it will go away once we initialize frmRegister in the constructor.

1. Now add a function to create a FormGroup by defining the controls you want to use in your form:

|  |
| --- |
| **constructor() { }**  **createFormGroup() {**  **return new FormGroup({**  **username: new FormControl(),**  **password: new FormControl()**  **});**  **}**  **ngOnInit(): void {** |

1. We also need to instruct the new FormGroup to be instantiated, which we can do from the constructor:

|  |
| --- |
| **constructor() {**  **this.frmRegister = this.createFormGroup();**  **}** |

1. Create an onSubmit() method and for now we will simply log the values of the form.

|  |
| --- |
| **onSubmit(): void {**  **console.log(this.frmRegister.value);**  **}** |

1. In the template we need to bind our form and its controls to our class properties:

|  |
| --- |
| **<main>**  **<div class="container">**  **<h2 class="pb-2">Register Form</h2>**  **<form [formGroup]="frmRegister" (ngSubmit)="onSubmit()">**  **<div class="form-group">** |

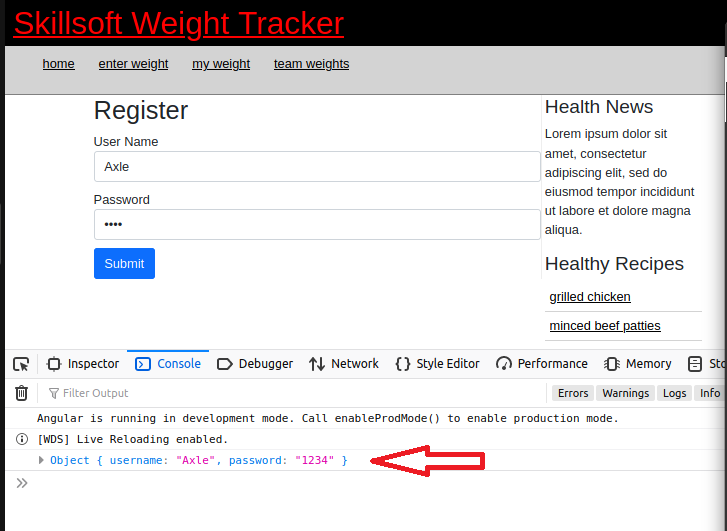
The **formGroup** directive is responsible for tracking the overall value of the form , which contains the values of all of its form fields. It will also keep track of the overall validity state of the form, which is dependent on the validity state of its form fields.

1. Rename each control to be of the formControlName type:

|  |
| --- |
| **<form [formGroup]="frmRegister" (ngSubmit)="onSubmit()">**  **<div class="form-group">**  **<label for="uname">User name</label>**  **<input type="text" class="form-control" id="uname" formControlName="username">**  **</div>**  **<div class="form-group">**  **<label for="exampleInputPassword1">Password</label>**  **<input type="password" class="form-control" id="exampleInputPassword1" formControlName="password">**  **</div>**  **<button type="submit" class="btn btn-primary">Submit</button>**  **</form>** |

The **[formGroup]** directive binds the entire form to the formGroup component.

1. Test the form, see results below



1. Alternative way to create the form:

|  |
| --- |
| **export class RegisterComponent implements OnInit {**  **myForm: FormGroup;**  **ngOnInit() {**  **this.myForm = new FormGroup({**  **name: new FormControl('Sammy'),**  **email: new FormControl(''),**  **message: new FormControl('')**  **});**  **}**  **}** |

# part 07 – Validation

1. In the Register component file, add the Validators module by importing it (if you do not already have it):

|  |
| --- |
| **import { Component, OnInit } from '@angular/core';**  **import { FormControl, FormGroup, Validators } from '@angular/forms';**  **@Component({console.log(this.frmRegister.value);**  **}** |

1. For each control, add the *required* validator in an array as the second argument to the **FormControl()** method. The first parameter to the method is empty for now, but you could pass in a prompt to the user using that method:

|  |
| --- |
| **createFormGroup() {**  **return new FormGroup({**  **username: new FormControl('', [Validators.required]),**  **password: new FormControl('', [Validators.required])**  **});**  **}** |

1. For our first validation we will disable the submit button unless something is entered in the fields:

|  |
| --- |
| **</div>**  **<button type="submit" class="btn btn-primary" [disabled]="!frmRegister.valid">Submit</button>**  **</form>** |

1. Let’s add a pair of *span* tags that will check the controls object for errors, username first

|  |
| --- |
| **<form [formGroup]="frmRegister" (ngSubmit)="onSubmit()">**  **<div class="form-group">**  **<label for="username">User name</label>**  **<input type="text" class="form-control" id="username" formControlName="username">**  **<span \*ngIf="frmRegister.controls['username'].errors"> Invalid**  **</span>** |

\***ngIf** is what they call a structural directive

1. Do the same for password

|  |
| --- |
| **<div class="form-group">**  **<label for="password">Password</label>**  **<input type="password" class="form-control" id="password" formControlName="password">**  **<span \*ngIf="frmRegister.controls['password'].errors"> Invalid**  **</span>**  **</div>**  **<button type="submit" class="btn btn-primary" [disabled]="!frmRegister.valid">Submit</button>**  **</form>** |

Test before moving to #6

1. The problem now is that the ‘invalid’ word appears as long as the form is on the screen. We can control the appearance of ‘invalid’ by implementing Angular’s form control properties:

|  |
| --- |
| **<div class="form-group">**  **<label for="username">User name</label>**  **<input type="text" class="form-control" id="username" formControlName="username">**  **<span \*ngIf="frmRegister.controls['username'].errors && !frmRegister.controls['username'].pristine">**  **Invalid**  **</span>**  **</div>**  **<div class="form-group">**  **<label for="password">Password</label>**  **<input type="password" class="form-control" id="password" formControlName="password">**  **<span \*ngIf="frmRegister.controls['password'].errors && !frmRegister.controls['password'].pristine">**  **Invalid**  **</span>**  **</div>** |

# part 08 – Fixing the HTML and Complete Routing

1. Make the following changes to the [home.component.html](http://home.component.html) file:

|  |
| --- |
| **</header>**  **<nav>**  **<ul>**  **<li><a href="index.html">home</a></li>**  **<li><a href="enterweight.html">register</a></li>**  **<li><a href="myweights.html">login</a></li>**  **</ul>**  **</nav>** |

1. Also in [home.component.html](http://home.component.html) change the physical anchor link to use *routerLink*:

|  |
| --- |
| **</header>**  **<nav>**  **<ul>**  **<li><a routerLink="/home">home</a></li>**  **<li><a routerLink="/register">register</a></li>**  **<li><a routerLink="/login">login</a></li>**  **</ul>**  **</nav>**  **<div id="container">** |

Once you have made this change on the *home* component, do it to the other component i.e. *register*. We will do the login component tomorrow.

1. Make the following changes to the [styles.css](http://home.component.html) file:

|  |
| --- |
| **header h1 a {**  **color:red;**  **text-decoration:none;**  **}** |

1. Comment out the following lines in the styles.css file:

|  |
| --- |
| **/\* label{**  **display:inline-block;**  **width:150px;**  **text-align:right;**  **}**  **button{**  **margin-left:155px;**  **margin-top:10px;**  **}**  **\*/** |

1. Make the following changes to the styles.css file:

|  |
| --- |
| **nav{**  **text-align:left;**  **background-color:lightgray;**  **border-bottom:1px solid gray;**  **height: 46px;**  **}**  **…**  **#logo{**  **float:right;**  **width:160px;**  **height:100px;**  **}** |

Feel free to change these numbers according to your browser and layout

# Appendix A – Install Angular 14 on Linux Ubuntu 20

First install NodeJS if it is not already installed, but update the profile first, so:

1. sudo apt update
2. sudo apt install nodejs
3. sudo apt install npm

At this point you can install the Angular CLI

1. npm install -g @angular/cli

Verify that NodeJS, NPM and Angular was installed, run these commands:

nodejs -v

npm -v

ng --version

The entire process could take between 5 to 15 minutes depending on your system and internet connection

If you are using VS Code you may get a message to install **Angular Language Service**, please install it.

# Appendix B – Angular Architectural Concepts

Angular uses the concept of modules (Ng Modules) into which components are placed. There are built-in modules that come with the installation of Angular. Some of these modules we will be using in the course include the HttpClientModule and the FormsModule. An Ng Module is just a TypeScript class with an @NgModule decorator. Most decorators add metadata to the class and in come cases functionality. By default we get the AppModule to help us kickstart our customized development.

Decorators may contain declarations, exports, imports, providers and bootstrap classes. Declarations handle views like component views and directive views. Export classes ensure that a class can be accessed by other classes. Imports exposes modules required by a class. Providers handle Services which are mostly logic required by some class. Bootstrap is in the root component and provides the initial view.

There are several JS modules used as libraries in an Agular application. Libraries such as @angular/core, @angular/router and Material are used to add functionality. These libraries are simply imported.

Components comprise of a TypeScript class, some kind of HTML template for display and a stylesheet. A component will have the @Component decorator to define it as a component.

A customized component will usually have a selector which is an instructor to Angular to insert this particular component where ever it finds the selector. The selector tag within the HTML is usually written as <app-root></app-root>.

The templateUrl will point to an html file which acts as the template for a component. styleUrls of course does the same for CSS files.

Directives:

Directives are instructions that instruct the DOM as to how to place your components and business logic in the Angular project. Directives are just JS class which are declared as @directive. There are 3 directives in Angular: Component Directives, Structural Directives and Attribute Directives.

Component Directives look like this @Component. They contain the detail of how the component should be processed, instantiated and used at runtime.

Structural directives start with a \* sign. These directives are used to manipulate and change the structure of the DOM elements. For example, \*ngIf and \*ngFor.

Attribute directives are used to change the look and behavior of the DOM elements. For example: ngClass, ngStyle etc.

The main building blocks of Angular are:

* Modules
* Components
* Templates
* Services
* Metadata
* Directives
* Data binding
* Dependency injection

Here are a few Angular CLI commands that we will be using

|  |  |  |
| --- | --- | --- |
| *add* |  | Used to add support for an external library to your project. |
| *build* | Will compile an Angular app into an output directory named dist/ at the given output path. |
| *generate* | Generates and possibly modifie files based on a schematic. |
| *new* | Creates a new workspace and a boilerplate Angular app. |
| *run* | Runs an Architect target |
| *serve* | Builds and serves your app via http, also re-compiles when it detects changes. |
| *test* | Executes unit tests in a project |
| *update* | Updates your application and its dependencies |

Angular 14 File Explanation

* src folder: all the action takes place here
* app folder: all the files, that support app components.
* app.component.css: the cascading style sheets code for your app component.
* app.component.html: the template html file connected to app component and is used by angular to do any data binding.
* app.component.spec.ts: use the command ng test to see this file in action. It is a unit testing file related to app component. All files that have .spec in the middle is a test file
* app.component.ts: probably the most important typescript file which contains the view logic driving the component.
* app.module.ts: a file which includes all the dependencies for the entire website. This file defines any modules to be imported, components to be declared and the main component to start the app
* karma.config.js: This file specifies the config file for the Karma Test Runner, Karma has been developed by the AngularJS team which can run tests for both AngularJS and Angular 2+
* main.ts: As defined in angular.json file, this is the main ts file that will first run. This file bootstraps (starts) the AppModule from app.module.ts , and it can be used to define global configurations.
* polyfills.ts: This file is a set of code that can be used to provide compatibility support for older browsers. Angular 7 code is written mainly in ES6+ language specifications which is getting more adopted in front-end development, so since not all browsers support the full ES6+ specifications, pollyfills can be used to cover whatever feature missing from a given browser.
* styles.css:/ This is a global css file which is used by the angular application.
* tests.ts: This is the main test file that the Angular CLI command ng test will use to traverse all the unit tests within the application and run them.
* tsconfig.json: This is a typescript compiler configuration file.
* tsconfig.app.json: This is used to override the tsconfig.json file with app specific configurations.

tsconfig.spec.json: This overrides the tsconfig.json file with app specific unit test configurations

# Appendix C – Angular Directives

Directives are functions used reinforce HTML, make it do much more than what it was designed for. These directives have names like \**ngFor* and *ngStyles* but can be any name you make up and they are specific to an HTML element, an attribute or class

DOM manipulation directives are called attribute or structural directives.

Attribute directives manipulate the DOM by changing its behavior and appearance.

Using the Existing Angular Directives in an example:

|  |
| --- |
| **<div [**[**ngStyle**](https://angular.io/api/common/NgStyle)**]="myStyles">**  **Content goes here**  **</div>** |

You can now define myStyles somewhere in your .ts file as a function.

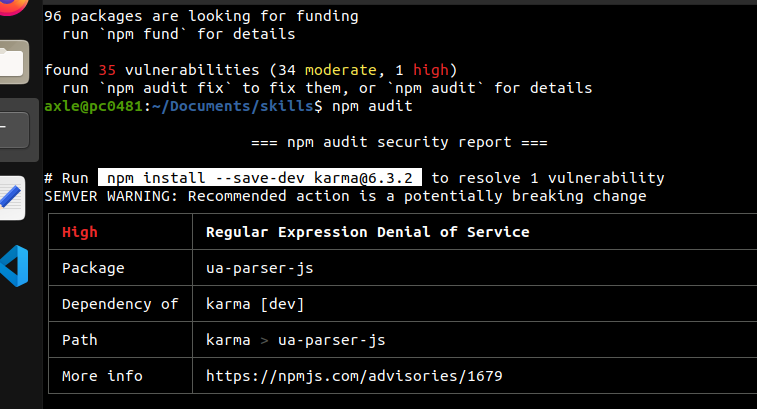
Structural directives are meant to create and destroy DOM elements and usually start with the \* character such as *\*ngIf*

|  |
| --- |
| **<div \*ngIf="condition">Content to render when condition is true.</div>** |

Components are also special directives

# Appendix D – Installation Issues

If you get issues while installing json-server run npm audit to see what might be stopping the installation and how you might be able to fix it:



So in this case, I installed [karma@6.3.2](mailto:karma@6.3.2)

# Appendix D – @NgModule

**Declarations** are used to declare components, directives, pipes that belong to the current module. Think of a namespace, declarations create a namespace so all the components in this @NgModule are available to each other in a public but protected way.

**Imports** (and exports) work just like in other programming languages. They are used to import supporting modules like FormsModule, RouterModule and the CommonModule.

**Providers** are used by modules for accessing the services required by components, directives. The process is known as injecting services into the component.

The **bootstrap** property simply points to a component that will be used to start the application.

# Appendix D – Watchpack Error

For some of you on Linux systems, especially VMs that have limited memory, you may get a *watchpack* error.

To solve this, at least temporarily, run the following command:  
**sudo sysctl -w fs.inotify.max\_user\_watches=524288**

# Appendix E – Path Match

Path-matching can be 'prefix' or 'full'. The default is 'prefix'.

Usually, the router parses a URL starting on the left to see if it matches an existing path. Parsing stops when there is a config match.

The path-match option of 'full' looks for the entire URL. It is good practice to do this when configuring empty-path routes. This avoids a potential endless loop.