Contents

[**part 01 – NG16 Setup** 2](#_Toc141940556)

[**part 02 – HTML Element Objects** 3](#_Toc141940557)

[**part 03 – Event Binding** 4](#_Toc141940558)

[**part 04 – Property Binding** 5](#_Toc141940559)

[**part 05 – Data Binding** 6](#_Toc141940560)

[**part 06 – Two-Way Binding** 9](#_Toc141940561)

[**part 07 – Selector as Attribute and as Class** 11](#_Toc141940562)

[**part 08 – Introduction to Directives** 12](#_Toc141940563)

[**part 09 – Introduction to Components** 14](#_Toc141940564)

[**part 10 – Components Child to Parent Communication (Static Data)** 16](#_Toc141940565)

[**Appendix A – Install Angular 16 on Linux Ubuntu 20** 18](#_Toc141940566)

[**Appendix B – Angular Architectural Concepts** 18](#_Toc141940567)

[**Appendix C – Angular Directives** 21](#_Toc141940568)

[**Appendix D – Installation Issues** 22](#_Toc141940569)

[**Appendix D – @NgModule** 22](#_Toc141940570)

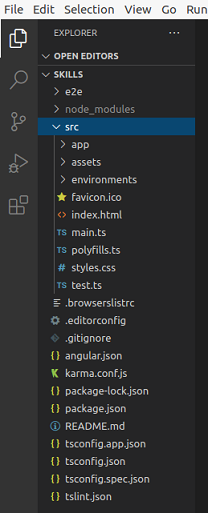
[**Appendix E – Add Bootstrap** 23](#_Toc141940571)

[**Appendix F – Components Child to Parent Communication (Dynamic Data)** 24](#_Toc141940572)

Day01 HTML Elements and Angular Components

# part 01 – NG16 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. For this particular boot camp, I will use *skills2* as the app folder, but you may use just *skills* or any other name you wish.

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 **N** 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>** |

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

# part 02 – HTML Element Objects

1. A typical HTML document is made up of HTML elements or tags (subtle differences).
2. As an object, these elements can have attributes (properties) and generate events (methods).

|  |
| --- |
| **id, border, href, width, height, src, style, alt, title, class, checked, draggable, onblur, onclick, onkeypress, onsubmit, placeholder, value, selected** |

1. In this example we see two attributes and their values as well as an event, the onclick() event:

|  |
| --- |
| **<p id="myparagraph" class="myclass" onclick="alert('Hello')">Lorem ipsum dolor sit amet...</p>** |

1. In index.html, you will see a pair of custom elements between the <body> tags. This element will get replaced by a template via app.component.ts file:

|  |
| --- |
| **@Component({**  **selector: 'app-root',**  **templateUrl: './app.component.html',**  **styleUrls: ['./app.component.css']**  **})**  **export class AppComponent {** |

The index.html file is in the root of your Angular app, inside the src folder.

1. When app.component.ts file gets executed, it looks for app-root on the html file and replaces that custom element by whatever regular HTMLHTML happens to be in app.component.html file (including any CSS)

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

Note, if you duplicate or triplicate this line of code, then the HTML output will duplicate as well.

1. At the same time, whatever is between {{}} gets a text value that is currently stored in the app.component.ts file. In this case line 9 has the title property initialized to ‘skills2’ so that gets printed in addition to *app is running*.

Note: in my case I am using **skills2** to distinguish it from the original **skills** app from the first bootcamp.

1. This means that we can pass values from our TS code into the HTML content. This also means that we can call functions on our TS code:

|  |
| --- |
| **export class AppComponent {**  **title = 'skills2';**  **getTitle(){**  **return this.title;**  **}** |

1. Change title to getTitle() on the app.component.html file:

|  |
| --- |
| **<span>{{ getTitle() }} app is running!</span>** |

The outcome is the same, but now we are using a method

|  |
| --- |
| **String interpolation is the use of {{}} to interpret properties and events on the corresponding .ts file. Insert the name of a Class property or a function inside of these curly braces in order for the value of that property or function to show on the HTML side. This is all happening in the same component.** |

# part 03 – Event Binding

Angular has made it possible for us to bind to events on the DOM. This means that most of the events like clicks, mouse events and timer events can be hooked into. Remember that events can occur on a single element, a group of elements or the entire HTML document within a browser.

1. In app.component.ts add a new property to the class:

|  |
| --- |
| **export class AppComponent {**  **title = 'skills2';**  **divColor = "Black";**  **getTitle(){** |

1. Just below the getTitle() function add a new function as show below:

|  |
| --- |
| **return this.title;**  **}**  **makeBlue(){**  **this.divColor = "Blue";**  **}**  **}** |

1. In the template, add a new <div> with the following content:

|  |
| --- |
| **<span>{{ getTitle() }} app is running!</span>**  **<div>{{ divColor }} is our colour!</div>** |

1. At this point the template with its string interpolation, is picking up the hard coded colour of *black*. Lets now tap into the click event of our <div> and point that event to our makeBlue() function:

|  |
| --- |
| **<span>{{ getTitle() }} app is running!</span>**  **<div (click)="makeBlue()">{{ divColor }} is our colour!</div>** |

Notice the parenthesis around *click* in the <div> tag.

1. Test the code by clicking on the content in the browser (second line). The words should change and include the property value of Blue.

|  |
| --- |
| **DOM events can be utilized to activate properties or methods in the TS file. This is called binding. In the example we bound the click event of the <div> tag to a method in the Class for this component.** |

# part 04 – Property Binding

1. It is possible to control an HTML element’s property via TS code. Any property can be bound and in this section, we will bind the *checked* property of a checkbox.
2. Add this simple CSS class in the app.component.css file. It is not necessary but will make the *checkbox* a little larger:

|  |
| --- |
| **.checkbox-lg {**  **top: .5rem;**  **width: 1.5rem;**  **height: 1.5rem;**  **}** |

1. Back in the HTML file of the *app* component add the following code which will add a *checkbox* on the browser window:

|  |
| --- |
| **<div (click)="makeBlue()">{{ divColor }} is our colour!</div>**  **<div class="form-check">**  **<input**  **class="checkbox-lg"**  **type="checkbox"**  **id="cb1"**  **checked**  **/>** |

I used separate lines to make the code easier to read

1. To keep the example simple, we will bind this input control to an existing function, the makeBlue() function. But first lets create a new Boolean property on the component:

|  |
| --- |
| **export class AppComponent {**  **title = 'skills2';**  **divColor = "Black";**  **approved = false;**  **getTitle(){** |

1. Then in the makeBlue() function we will toggle this property:

|  |
| --- |
| **}**  **makeBlue(){**  **this.divColor = "Blue";**  **this.approved = !this.approved;**  **}** |

1. Finally for this section bind the two, so in the HTML add square brackets around the checkbox’s checked property and point it to the makeBlue() function:

|  |
| --- |
| **type="checkbox"**  **id="cb1"**  **[checked]="approved"**  **/>**  **</div>** |

|  |
| --- |
| **By placing [ ] around the HTML element’s property, and pointing it to a class property, we are connecting these two items. So whatever value the class has for *approved*, it gets transferred to *checked*. Now, the checkbox’s checked property can be changed by accepting a value that the Class gives it. At the same time, we can control what the Class property is by clicking on a different DOM element and calling a function that was bound earlier. Of course we could accomplish the same thing by clicking on the checkbox element.** |

# part 05 – Data Binding

Data can be generated in any file, but typically in this boot camp, data will usually refer to something that the user wants us to know. The user will communicate with our application via the HTML document we provide.

1. The best way to demonstrate data binding is of course with an input text box and a label or paragraph. Add the following two elements to your existing HTML code

|  |
| --- |
| **[checked]="approved"**  **/>**  **<p></p>**  **<label>Testing Data Binding</label>**  **<input**  **type="text"**  **class="form-control"**  **maxlength="6"**  **/>**  **</div>** |

The <p> tags are just to create some space

1. We already know we can bind to events such as click, mouseover etc. The input control has an event called *oninput*. Angular can bind to the event using (input) and we can then point that event to a function in the TS code:

|  |
| --- |
| **class="form-control"**  **maxlength="6"**  **(input) = "handleInput()"**  **/>**  **</div>** |

We will create handleInput() in the next point. We did something similar with the makeBlue() function in Part 3.

1. Create a function on the component to handle any input coming from this click event:

|  |
| --- |
| **makeBlue(){**  **this.divColor = "Blue";**  **this.approved = !this.approved;**  **}**  **handleInput(){**  **console.log("Event handled!")**  **}**  **}** |

1. At this point if you entered anything into the text box in the browser, the function will fire but it is not useful. We need to pass an Angular reserved object called $event. This variable is loaded with lots of data that we can access. Add this object first on the HTML template:

|  |
| --- |
| **<label>Testing Data Binding</label>**  **<input**  **type="text"**  **class="form-control"**  **maxlength="6"**  **(input) = "handleInput($event)"**  **/>**  **</div>];**  **@NgModule({** |

1. Then on the TS code, accept that object but also specify the object as an Event type:

|  |
| --- |
| **handleInput(event : Event){**  **console.log("Event handled!")**  **}** |

If you want, you can log *event* here just to see what it looks like

1. Moving on, we will now access the contents of that text box and pass it on to the <p> tag using a new class property and string interpolation. First, add a new property to the class:

|  |
| --- |
| **title = 'skills2';**  **divColor = "Black";**  **approved = false;**  **userInput = "";**  **getTitle(){**  **return this.title;**  **}** |

1. Then in the handleInput() function, we must now identify the exact type of HTML element we are handling. First the input control is of type HTMLInputElement and we use Generics to identify the event.target as of this type. Second, you can’t just access the value directly, we need to wrap the entire event.target along with its generic type into parenthesis. Only then can you access its value (the content that the user types):

|  |
| --- |
| **handleInput(event : Event){**  **//console.log("Event handled!" + event)**  **this.userInput = (<HTMLInputElement>event.target).value ;**  **}** |

1. Now we can pass this value back to the browser via string interpolation:

|  |
| --- |
| **<input**  **type="text"**  **class="form-control"**  **maxlength="6"**  **(input) = "handleInput($event)"**  **/>**  **<p>{{userInput}}</p>**  **</div>** |

1. We could also pass data to the browser via innerHTML, see below:

|  |
| --- |
| **(input) = "handleInput($event)"**  **/>**  **<p>{{userInput}}</p>**  **<div [innerHTML]="userInput"></div>**  **</div>** |

Here we bind the innerHTML property of the <div> tag to a property in the TS Class file.

The next two points are optional. We already did something similar in Part 3 above.

1. I created a new method in the component TS code where I simply return the value in the userInput variable:

|  |
| --- |
| **this.userInput = (<HTMLInputElement>event.target).value ;**  **}**  **displayUserInput(){**  **return this.userInput;**  **}** |

1. Then just call this function from the HTML template:

|  |
| --- |
| **<p>{{userInput}}</p>**  **<div [innerHTML]="displayUserInput()"></div>**  **</div>** |

This just demonstrates that we can access data on the component via a property and string interpolation or via a function that returns data that can be converted into a string.

# part 06 – Two-Way Binding

So far we have seen event, property and data binding. Almost all HTML elements have both data and events associated with them. There are certain HTML elements where this is extremely important, an example is the *input* tag. Of course, Angular gives us the ability to combine property and event binding. This is commonly referred to as two-way binding. It works just like what we have now, but less code to write.

1. Before we change the template, this feature needs some support. In the app.module.ts file we would need to import the formsModule:

|  |
| --- |
| **import { BrowserModule } from '@angular/platform-browser';**  **import { FormsModule } from "@angular/forms";**  **import { AppComponent } from './app.component';**  **@NgModule({**  **declarations: [**  **AppComponent**  **],**  **imports: [**  **BrowserModule,**  **FormsModule**  **],** |

Also add the module to the imports array. We will not use this file anymore during this boot camp.

1. With the module imported, simply put some square brackets around the input event and replace the input key word with ng-model:

|  |
| --- |
| **class="form-control"**  **maxlength="6"**  **[(ngModel)] = "handleInput($event)"**  **/>**  **<p>{{userInput}}</p** |

1. Next simply point this directive to a property on the component class:

|  |
| --- |
| **maxlength="6"**  **[(ngModel)] = "userInput"**  **/>** |

With two-way binding, we hook into the input event of the input form control AND update the property userInput automatically

1. Test the changes and you will see that it works just like before. However the advantage of this structure is that the value of this property can be changed from anywhere within the code:

|  |
| --- |
| **divColor = "Black";**  **approved = false;**  **userInput = "Hello Skillsoft";**  **getTitle(){**  **return this.title;**  **}** |

If you did make this change, *Hello Skillsoft* will appear on the browser window until the user decides to change this value.

1. (Optional) To test this even further, duplicate the input box and set it up just like you did before, so:

|  |
| --- |
| **<p></p>**  **<input**  **type="text"**  **class="form-control"**  **maxlength="6"**  **[(ngModel)] = "userInput"**  **/>**  **<input**  **type="text"**  **class="form-control"**  **maxlength="6"**  **(input) = "handleInput($event)"**  **/>**  **<p>{{userInput}}</p>** |

Code from Part 5 #8

1. As the page refreshes, only the input control with two-way binding gets the value set in the TS code. The second input box is empty since it did not implement two-way binding. Also the box with two-way binding will not update the second box. This is because the property, userInput, is getting updated, NOT the box itself. On the other hand, the box without two-way binding will update everything!

# part 07 – Selector as Attribute and as Class

1. It is possible to use the *selector* defined in the TS file as an attribute in the HTML template. Since I do not yet have another component, I will demonstrate this feature using app-root. First in the app.component.ts file, add square brackets around app-root:

|  |
| --- |
| **@Component({**  **selector: '[app-root]',**  **templateUrl: './app.component.html',**  **styleUrls: ['./app.component.css']**  **}** |

1. Then in the index.html (we don’t have a component yet) comment out the original app-root code and add a pair of <div> tags:

|  |
| --- |
| **<body>**  **<!-- <app-root></app-root> -->**  **<div></div>**  **</body>** |

1. Now just add the app-root selector as an attribute of this <div> tag:

|  |
| --- |
| **<body>**  **<!-- <app-root></app-root> -->**  **<div app-root></div>**  **</body>** |

The page should work exactly like it did before, however it is NOT recommended that you do this in production.

1. In a similar way, we can change the selector to behave like a class. Back in the TS file, remove the square brackets and add a period in front of app-root:

|  |
| --- |
| **@Component({**  **selector: '.app-root',**  **templateUrl: './app.component.html',**  **styleUrls: ['./app.component.css']**  **})** |

1. Then in the HTML, use app-rot as a class:

|  |
| --- |
| **<body>**  **<!-- <app-root></app-root> -->**  **<div class = "app-root"></div>**  **</body>** |

The page should work exactly like it did before, however it is NOT recommended that you do this in production.

**Note:before continuing on, please revert the code back to what it was before we started # 6. If you are not sure, just unzip the src folder for day1part6.zip file and use those files instead.**

# part 08 – Introduction to Directives

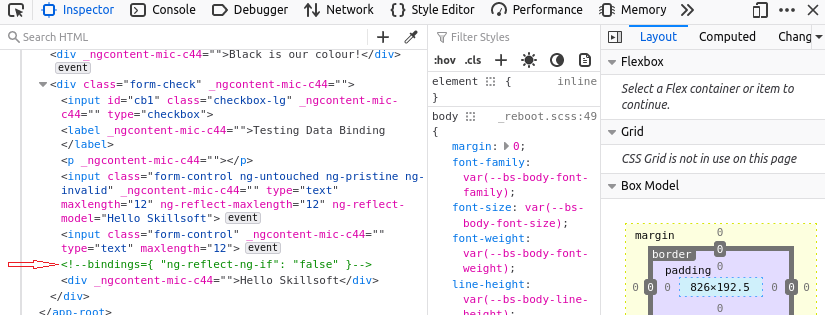
A directive is simply an instruction to the DOM. It tells the DOM to do something like add a new pair of <div> tags or change the look and feel of some HTML element. Components are directives. We used a directive in part 6 above, ng-model. There are structural directives like \*ngIF and \*ngFor. However mostly you will come across attribute directives where the directive is used like an attribute.

Some of the examples in this section does not make sense from a user perspective but is being shown to highlight the mechanical aspect of Angular features such as Directives.

1. At the moment we have two input controls and two places where we display the same content. Lets add a structural directive to the <p> tag on or around line 24:

|  |
| --- |
| **/>**  **<p \*ngIf="approved">{{userInput}}</p>**  **<div [innerHTML]="displayUserInput()"></div>** |

Since anything \*ngIf points to must evaluate to true/false I am just using the approved property we added in Part 4 #4

1. Once the page refreshes, one of the <p> tag and its content disappears. You can verify this by looking at the derived code. There are two input controls but only one output, the <div>  
     
   
2. Of course, there is an *else* part to this directive. However, I will leave that until we cover *local references* and *ng-template*. In the meantime, remember that the property called approved is being toggled by the *click* event of the content in our <div> tag on line 2. This is in the template. So, to see the <p> tag appear and disappear, click on this <div> tag repeatedly.
3. The other structural directive \*ngFor works well with repetitive data such as arrays and other collections. Add any array to the class with three or four elements:

|  |
| --- |
| **divColor = "Black";**  **approved = false;**  **userInput = "Hello Skillsoft";**  **messages : string[] = ["Hello", "from", "Skillsoft"];**  **getTitle(){** |

Notice the declaration is different, this is TypeScript recommended way to use variables. Change the other declarations to this format.

1. The messages array has three elements. We can use this number in the directive like this:

|  |
| --- |
| **<p \*ngIf="approved">{{userInput}}</p>**  **<div [innerHTML]="displayUserInput()"></div>**  **<div \*ngFor="let message of messages">**    **</div>**  **</div>** |

We add a new pair of <div> tags, but we will end up with three pairs of <div> tags. Each pair of tags will contain one of the elements of the array.

1. Simply use string interpolation:

|  |
| --- |
| **<div \*ngFor="let message of messages">**  **{{message}}**  **</div>** |

1. Another useful directive, though not structural is ngStyle. We simply apply it according to the syntax below:

|  |
| --- |
| **<div [innerHTML]="displayUserInput()"></div>**  **<div**  **\*ngFor="let message of messages"**  **[ngStyle]="{color:'blue'}"**  **>**  **{{message}}**  **</div>** |

I moved the attributes into different lines for readability. The highlighted line here is the equivalent of the HTML/CSS: style="color:'blue'"

1. The above will turn all output within the \*ngFor code block blue. The power of this directive of course comes with conditions. See below:

|  |
| --- |
| **<div  \*ngFor="let message of messages"  [ngStyle]="{color: message.length > 4 ? 'blue':'green'}"  >** |

According to the code, if the length of the output word is greater than four characters, the entire word will be blue, otherwise it will be green.

1. Another useful directive, though not structural is ngClass. We simply apply it according to the syntax below:

|  |
| --- |
| **<span [ngClass]="{standOut:approved==false}">{{ getTitle() }} app is running!</span>**  **<div (click)="makeBlue()">{{ divColor }} is our colour!</div>**  **<div class="form-check">** |

So the class called standout will be applied to the entire <span> tag as long as *approved* is false. Remember *approved* is a property in the class we added in Part 4 #4.

1. You can use any CSS you want, but this is the class I coded for this point. It is placed in the app.component.css file:

|  |
| --- |
| **.standOut{**  **font-size: larger;**  **font-weight: bold;**  **}** |

# part 09 – Introduction to Components

When the app was built using the Angular CLI, it came with property binding. The name of the app was passed to the template via double curly braces.

In Part 3 we saw event binding. We created a function called makeBlue() and we called it from the HTML template by hooking into the *click* event of the <div> tag.

In Part 4 we hooked into the *property* of an HTML element (checked) and attached it to a property in the TS code (approved). This is *property binding*.

We saw data binding in Part 5. We added a text box, then with *event binding*, we passed data into the TS code. We stored that user input in a property and passed it back to the HTML template where it was displayed between <p> tags.

In Part 6 we say two-way binding. The content the user gave us was passed back to the DOM but that same DOM element was initialized from the TS code.

With components, we should be able to perform all these tasks, with the addition of cross communication. We should be able to pass data from parent to child to grand child and back up the chain. Communication between grandchildren is possible but is an advanced topic. Spoiler alert, it can be done with services.

Before continuing, use the CLI to create a new component:  
ng g c second-child --skip-tests

1. In the first example we will pass a value from the parent to the child component. This is sometimes referred to as *custom property binding*. Angular allows this via a special decorator called *@Input*. We will pass the value of the messageToChildren property of the parent component to the FirstChildComponent.

In first-child.component.ts file, create a property to hold the Parent’s message:

|  |
| --- |
| **export class FirstChildComponent implements OnInit {**  **messageFromParent : string = "";**  **constructor() { }** |

1. Then import the Input package from @angular/core:

|  |
| --- |
| **import { Component, OnInit, Input } from '@angular/core';**  **@Component({**  **selector: 'app-first-child',** |

1. Now decorate the property from #1 with @Input:

|  |
| --- |
| **export class FirstChildComponent implements OnInit {**  **@Input() messageFromParent : string = "";**  **constructor() { }** |

By doing this, messageFromParent is now setup to accept values passed to it from a parent component. In our case the parent component is app.component.ts.

1. The first child component has its own selector, app-first-child. Use it to display the child component:

|  |
| --- |
| **<span [ngClass]="{standOut:approved==false}">{{ getTitle() }} app is running!</span>**  **<app-first-child></app-first-child>**  **<div (click)="makeBlue()">{{ divColor }} is our colour!</div>**  **<div class="form-check">** |

Refresh the browser and the first-child’s content should appear in the browser

1. The next task is to bind the parent’s message (property) to the app-first-child custom element. This can be done from the parent, when we use the selector of the child component:

|  |
| --- |
| **<span>{{ title }} app is running!</span>**  **<app-first-child [messageFromParent]="messageToChildren"></app-first-child>**  **<app-second-child></app-second-child>** |

Refresh the browser and the parent’s message should appear in the first-child component

1. Starting with the parent component (app.component.ts), the property messageToChildren is used in the parent’s HTML template. In the parent template, when the child template is invoked, a property on the child messageFromParent is bound to the parent’s property. The value of messageToChildren is passed into the child template.

|  |
| --- |
| <app-first-child [messageFromParent]="messageToChildren"></app-first-child> |

1. It can easily be a method in the parent component. Comment out the property and add a method with the same name:

|  |
| --- |
| **export class AppComponent {**  **title : string = 'skills2';**  **compStatus : string = "App Component - Top Component";**  **//messageToChildren : string = "Hello from app component";**  **messageToChildren(){**  **return "Hello from a method in the parent app!"**  **}**  **}** |

1. Then simply change the property in the parent’s template to a method call:

|  |
| --- |
| **<span class = "topComponent">{{ title }} app is running!</span>**  **<app-first-child class = "firstChild" [messageFromParent]="messageToChildren()"></app-first-child>**  **<app-second-child class = "secondChild"></app-second-child>** |

# part 10 – Components Child to Parent Communication (Static Data)

Child to Parent communication is more complicated. It involves raising an event on the child which then is used by the parent to transfer data.

1. Import Output and EventEmitter in the child component class:

|  |
| --- |
| **eimport { Component, OnInit, Input, Output, EventEmitter } from '@angular/core';**  **@Component({**  **selector: 'app-first-child',** |

1. In the first-child component class, declare a new property and decorate it with **@Output()**. Now, **eventEmitter** is a property and an event emitter at the same time:

|  |
| --- |
| **export class FirstChildComponent implements OnInit {**  **@Input() messageFromParent : string = "";**  **@Output() eventEmitter = new EventEmitter<string>();**  **constructor() { }** |

1. Create a property that will hold the string message:

|  |
| --- |
| **export class FirstChildComponent implements OnInit {**  **@Input() messageFromParent : string = "";**  **messageFromFirstChild : string = "Message from first-child";**  **@Output() eventEmitter = new EventEmitter<string>();**  **constructor() { }** |

1. Next add a method that will raise an event and emit a value, the message:

|  |
| --- |
| **ngOnInit(): void {**  **}**  **generateChildMessage() {**  **this.eventEmitter.emit(this.messageFromFirstChild);**  **}** |

Remember that the property messageFromFirstChild holds the message

1. Also, add a button on the child to activate the method in #4:

|  |
| --- |
| **<button**  **type="button"**  **(click)="generateChildMessage()">**  **Send message to parent**  **</button>** |

1. In the parent component, so in app.component.ts file, add a function to receive the message:

|  |
| --- |
| **messageToChildren(){**  **return "Hello from a method in the parent app!"**  **}**  **messageFromChildren(msg: string) {**  **console.log(msg);**  **}** |

1. Then in the parent’s HTML template, so in app.component.html, configure the custom element, so <app-first-child>. Connect the components by binding the *event emitter* in the child to a *method* in the parent that handles messages from the child:

|  |
| --- |
| **<span class = "topComponent">{{ title }} app is running!</span>**  **<app-first-child**  **class = "firstChild" [messageFromParent]="messageToChildren()"**  **(eventEmitter) = "messageFromChildren($event)"**  **>**  **</app-first-child>**  **<app-second-child class = "secondChild"></app-second-child>** |

Remember eventEmitter is an @Output property of the child. However, it points to a method (messageFromChildren) in the parent class in the app.component.ts file.

# Appendix A – Install Angular 16 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 16 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.

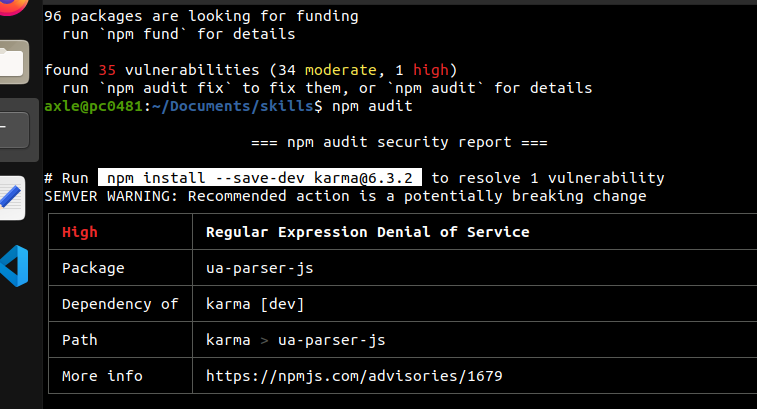
Structural directives are meant to create and destroy DOM elements and usually start with the \* character such as *\*ngIf*

|  |
| --- |
| **<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 E – 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": {** |

# Appendix F – Components Child to Parent Communication (Dynamic Data)

Child to Parent communication is more complicated. It involves raising an event on the child which then is used by the parent to transfer data. Now we pass data after a DOM event has been raised and where the DOM element itself is providing the data.

In this section we will insert a pair of Radio buttons and have the value of the clicked button sent to the parent component.

1. First on the first-child component, add a pair of <div> tags and inside of those, add two radio buttons with the same name:

|  |
| --- |
| **<p>{{messageFromParent}}</p>**  **<div>**  **Male <input type="radio"**  **name="gender" value="Male"/>**  **Female <input type="radio"**  **name="gender" value="Female"/>**  **</div>**  **<button** |

1. Two-way bind the each of these buttons to a *property* on the TS code, soon to be implemented:

|  |
| --- |
| **<div>**  **Male <input type="radio"**  **name="gender" value="Male"**  **[(ngModel)]="selectedRBValue" />**  **Female <input type="radio"**  **name="gender" value="Female"**  **[(ngModel)]="selectedRBValue" />**  **</div>** |

1. At the same time, hook into the onclick event and point it to a *method* in the class (soon to be implemented):

|  |
| --- |
| **<div>**  **Male**  **<input type="radio"**  **name="gender" value="Male"**  **[(ngModel)]="selectedRBValue"**  **(click)="generateGenderMessage($event)" />**  **Female**  **<input type="radio"**  **name="gender" value="Female"**  **[(ngModel)]="selectedRBValue"**  **(click)="generateGenderMessage($event)" />**  **</div>** |

The reason for passing the $event is so we can access the *value* of each input control in the TS code.

1. Over in the class, add that property we bounded to in the template:

|  |
| --- |
| **messageFromFirstChild : string = "Message from first-child";**  **@Output() eventEmitter = new EventEmitter<string>();**  **isChecked: boolean = false;**  **selectedRBValue : string = "Female";**  **constructor() { }** |

I initialized this property with the value of *Female*.

1. Next add a method that will raise an event and emit a value, the value of the radio button:

|  |
| --- |
| **generateGenderMessage(event : Event) {**  **}** |

1. In the method from #5, store the value of the clicked RB in the property we set up in #3 above:

|  |
| --- |
| **generateGenderMessage(event : Event) {**  **this.selectedRBValue = (<HTMLInputElement>event.target).value ;**  **}** |

Again, we know that it is an HTML Input Element that is being triggered. We pass the value of the RB to the property we set in #4

1. Then use the eventEmitter to emit the property’s value back to the parent:

|  |
| --- |
| **generateGenderMessage(event : Event) {**  **this.selectedRBValue = (<HTMLInputElement>event.target).value ;**  **this.eventEmitter.emit(this.selectedRBValue);**  **}** |

Remember eventEmitter is an @Output property of the child. However, it points to a method in the parent class in the app.component.ts file messageFromChildren

1. Remember, we did set the initial value of selectedRBValue to be *female*. Therefore, we should fire the eventEmitter just before the child component is drawn on the browser window. This will pick up that default value of our new property, *female*:

|  |
| --- |
| **constructor() { }**  **ngOnInit(): void {**  **this.eventEmitter.emit(this.selectedRBValue);**  **}**  **generateChildMessage() {** |

There are other lifecycle hooks we can use to emit the RB’s value, but for now ngOnInit() will work just fine. So on load, *Female* should show in the console window.