
Angular: Quick Notes

Version 0.1.0

Kapil Sharma



2018-10-27

Contents

1	About these notes	7
1.1	Sponsors	7
1.1.1	Ansh Systems	7
1.1.2	Jet Brains	8
1.2	About this book	8
1.2.1	Contributing	8
1.3	Supporting Videos	8
1.4	Supporting code and course project	9
1.5	License	9
1.6	About Author	9
2	Install & First Application	11
2.1	Node JS	11
2.2	Angular CLI	11
2.3	First Angular Application	12
2.3.1	Directory Sctucture	12
2.3.2	Changing output for First Application	12
2.4	How our application works?	14
3	Components; Angular's basic building blocks	17
3.1	Creating a component	17
3.1.1	Creating a component manually.	18
3.1.2	Creating a component using Angular CLI.	20
3.2	Component stylesheet	22
3.3	Data Binding	24
3.3.1	String interpolation	24
3.3.2	Property Binding	24
3.3.3	Event Binding	25
3.3.4	Two way binding	29

4	Course Project	33
4.1	Getting things done	33
4.1.1	Getting things done summary.	33
4.2	ToDo App	34
4.2.1	Home page or ToDo List Page	35
4.2.2	Add ToDo page	36
4.2.3	Category list and Add Category	36
4.3	Creating assignment project	36
4.3.1	Creating ToDo App	36
4.4	Using Bootstrap	37
4.5	Planning Components	38
4.6	Header component	38
4.7	ToDo Component	39
4.8	v0.1.0	43
5	Components Continued	45
5.1	Component structure	45
5.2	Custom Property Binding - Passing data from Parent to Child component.	46
5.3	Custom event binding - Passing data from child to parent component	47
5.3.1	New course project requirement: Adding Create component	48
5.3.2	Creating custom event	51
5.3.3	Catching custom event	52
5.4	Passing HTML to child component - ng-content	53
5.5	Local Reference	56
5.5.1	Local reference in Type Script file	58
6	Directives	61
6.1	Attribute directives	61
6.2	Structural directives.	61
6.3	Attribute directive example	61
6.3.1	ngClass	62
6.3.2	ngStyle	62
6.4	Structure directive example	63
6.4.1	ngFor	63
6.4.2	ngIf	64
6.4.3	Using else part with ngIf	64
7	Models, Services and Dependency Injection	67
7.1	Models	67

7.2	Services & Dependency Injection	69
7.3	ToDoService	70
7.3.1	How Dependency Injection is working?	73
7.3.2	Hierarchical Injector	73
7.3.3	How to get new instance?	74
7.4	Data service	76
7.4.1	Data Service for custom Event Binding.	79

1 About these notes

These notes are distributed with **Corporate Training: Angular** meetup scheduled by **Kapil Sharma** and **PHP Reboot** on October 27th, 2018.

Please remember, this is not complete book to teach you angular, but quick notes as most people make to remember a concept that they learned or understood earlier.

If you wish to learn Angular, **Corporate Training: Angular** meetup videos will be available on Kapil's [YouTube Channel](#)¹ after the meetup. Subscribe to YouTube channel for notification of all future meetup & training videos. In case of any doubt, you can get in touch with Kapil through [Twitter](#)² or [Linked In](#)³

1.1 Sponsors

Corporate Training: Angular meetup series by PHP Reboot is sponsored by [Ansh Systems](#)⁴ and [Jet Brains](#)⁵.

1.1.1 Ansh Systems

As Sponsors, **Ansh Systems** is providing arrangements and snacks for the meet-up. Ansh Systems, is having 200+ employees development center in Pune, India, working on technologies like PHP, Node JS, Dot Net, Angular, React, Vue, Android and iOS native development etc. This whole corporate training meet-up series are actually one of their many internal corporate trainings, provided to their new and existing employees. As sponsors, Ansh Systems also allowed PHP Reboot to publicly share their internal corporate training of Angular.

¹Kapil's YouTube Channel <http://bit.ly/kapilyoutube>

²Kapil's Twitter <https://twitter.com/kapilsharmainfo>

³Kapil's Linked In <https://www.linkedin.com/in/kapilsharmainfo>

⁴Ansh Systems: <http://www.ansh-systems.com>

⁵Jet Brains: <https://www.jetbrains.com>

1.1.2 Jet Brains

Jet Brains is expert in providing great IDE like PHP Storm, Web Storm, IntelliJ Idea etc. As sponsors, they are providing two Jet Brains license for any of their IDE to be raffled out during the meet-up. If you are the lucky winner of their IDE license during the meet-up, you may download any of their IDE with one year subscription

1.2 About this book

Right from his days as student, Kapil preferred to make notes for quick revision. Now a days, he is making his notes in markdown. This book is mostly Kapil's notes he made while learning Angular but updated a bit to be distributed as a EBook.

This book will be always work in progress. Once the training finishes, I will keep updating book updated with latest version and tips and tricks of Angular. You can check the version of book on title page and in the header of every single page in the pdf. You can go to Book's GitHub repository⁶ to check latest version. If you have older version, you can always download latest version from PDF Link⁷.

1.2.1 Contributing

Since this book is made through quick notes, there could be lot of type and grammar mistakes. If you find any such mistake, you are most welcome to fix them and send a PR. If you do not have time to send PR, feel free to create an issue and Kapil will fix it. Let's make this book as quick reference guide for Angular.

1.3 Supporting Videos

We will record all the meet-up videos and upload them on Kapil's YouTube channel. You will find all YouTube links updated in this section. However, I request you to subscribe my YouTube channel⁸ to get updated about newly uploaded videos. This will also motivate me to keep making free tutorial videos on new technologies and tools.

⁶<https://github.com/kapilsharma/AngularQuickNotes>

⁷<https://github.com/kapilsharma/AngularQuickNotes/blob/master/AngularQuickNotes.pdf>

⁸Kapil's YouTube Channel <http://bit.ly/kapilyoutube>

1.4 Supporting code and course project

1.5 License

This book is publicly available under Creative Commons Attribution Share Alike 4.0 International⁹ license. You can see the exact license by visiting links but in short:

- You are free to copy and share the e-book.
- You are free to use the book commercially.
- You are free to update the book and share provided you use similar license.

However, the restrictions is you should keep the license and give back proper attribution to original author. This can be done by linking back to GitHub Repository¹⁰.

1.6 About Author

Kapil Sharma is having 14+ years experience in Web Application Development and currently working as Technical Architect at Ansh Systems Pvt. Ltd. He often conduct internal trainings at Ansh Systems and contents of this training is actually internal corporate training at Ansh Systems.

If you have any doubt or question about this free course, you can get in touch with Kapil through his twitter handle @KapilSharmaInfo.

⁹<https://creativecommons.org/licenses/by-sa/4.0/>

¹⁰<https://github.com/kapilsharma/AngularQuickNotes>

2 Install & First Application

2.1 Node JS

As prerequisite, Angular need Node JS. Angular application does not depends on Node JS but while development, development tools of angular like CLI, Development Server, installing dependencies etc take advantage of tools provided by Node JS. Node JS can be downloaded from [Official Website](#)¹. It is recommended to install latest version, not LTS version.

It is recommended to install Node JS using [Node Version Manager \(NVM\)](#)². Angular works best with Latest version. However, if you are a Node JS developer, or depends on Node JS for other tasks, you might want to stay with LTS version. NVM, not only allow us to install multiple Node JS versions, but also it is very easy to switch versions of Node JS with NVM.

Intalling Node JS directly or through NVM is straight forward. Please google the latest information on installing Node JS.

2.2 Angular CLI

Angular CLI makes it very easy to work with Angular. We mostly develop Angular Applications in [Type Script](#)³, which is super set of Java Script. Don't worry about it, Type Script will covered during the Corporate Training Meetup. Type Script needs to be compiled to Java Script. Also, we have lot of dependencies for the project development and a local server to test the application. Angular CLI manages all that and makes developer's life very easy.

We can install Angular CLI globally with following command

```
1 npm install -g @angular/cli
```

¹<https://nodejs.org/en/>

²<https://github.com/creationix/nvm>

³Type Script: <https://www.typescriptlang.org>

2.3 First Angular Application

Angular CLI, installed above, can be used with `ng` command. Run following command

```
1 ng new HelloWorldApp
```

The command will create a new folder `HelloWorldApp` with angular project. To run local server, go to new project folder and run command

```
1 ng serve
```

After that, open `localhost:4200` in the browser to check default application.

2.3.1 Directory Sctucture

At root level, in project, you will find 3 folders

- `e2e`
 - This folder stands for `End to End` testing and contains test cases. We will check testing later.
- `node_modules`
 - This folder is created by npm (Node Package Manager) and contains all the dependencies of the project.
- `src`
 - This is the folder which contains source code of our project and while developing angular application, we will be mostly working within this folder.
 - This folder also contains few files and folder. We will be mostly using `app` folder and know about other files and folder as and when needed while we advance with angular learning. For now, just do not touch any other file/folder.

Other than these 3 folders, you will find few files as well at root level. They are mostly supporting files to configure our new Angular project.

2.3.2 Changing output for First Application

Let's delete everything in `app/app.component.html` and put simple code like

```
1 <h1>
2   Hello {{ title }}!
3 </h1>
```

The output will be as follow

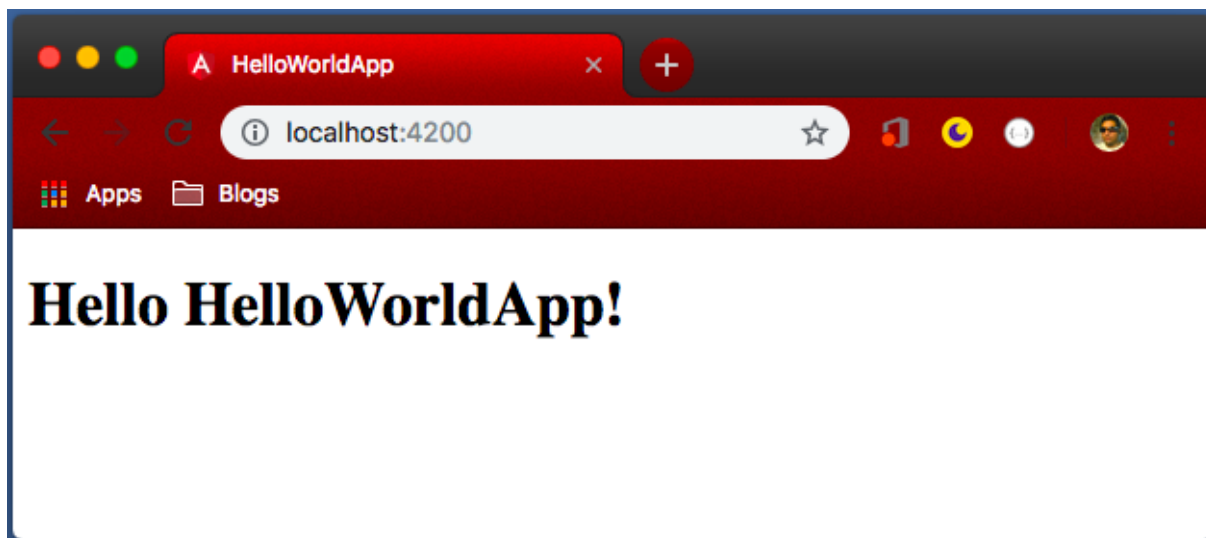


Figure 2.1: Hellow World App

We can see `{{ title }}` has been replaced by `HelloWorldApp`. This is done in `app.component.ts` file.

```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-root',
5   templateUrl: './app.component.html',
6   styleUrls: ['./app.component.css']
7 })
8 export class AppComponent {
9   title = 'HelloWorldApp';
10 }
```

Notice line 9 `title = 'HelloWorldApp'`; . This is actually where we are assigning value to the variable `title`, which is later used in `app.component.html` file. This is actually called **String Interpolation**. Thus, if there is a variable in Type Script class, we can display it in HTML with String Interpolation like `{{ variableName }}`. We will learn more about it in “Data Binding” section of next chapter “Understanding component; Basic building block of Angular”.

2.4 How our application works?

We saw we can update “app.component.html” to update design and “app.component.ts” to change the variable, but how are they working?

To understand it in better way, let’s check the source code of application in browser (Right click and select “view source” in chrome). It is like

```
1 <!doctype html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>HelloWorldApp</title>
6   <base href="/">
7
8   <meta name="viewport" content="width=device-width, initial-scale=1">
9   <link rel="icon" type="image/x-icon" href="favicon.ico">
10 </head>
11 <body>
12   <app-root></app-root>
13 <script type="text/javascript" src="runtime.js"></script><script type="
    text/javascript" src="polyfills.js"></script><script type="text/
    javascript" src="styles.js"></script><script type="text/javascript"
    src="vendor.js"></script><script type="text/javascript" src="main.js
    "></script></body>
14 </html>
```

What!! Our code from “app.component.html” (<H1> tag) or “app.component.ts” (title variable) is not there. How it is working then?

We write angular code in Type Script, which is super set of Java Script but the browser do not understands Type Script. Thus, to display the page, angular needs to compile our application into Java Script.

Notice Line 13, here we are including few Java Script files

- runtime.js
- polyfills.js
- styles.js
- vendor.js
- main.js

We didn’t wrote any of these JS file. They are actually dependencies of Angular application and in this simple Hello World Application, main.js contains our actual source code. We can also notice line 12

`<app-root></app-root>`, which is updated by generated JS files to display out contents. However, let's not try to understand generated Java Script; Angular CLI makes sure that generated JS files are optimized but not very convenient to read by humans. Instead, let's concentrate on how Angular CLI read our application to generate these JS files.

When we run `ng serve` (or `ng build`, as we will see later), Angular CLI first looks `angular.json` file to check the configurations. These configurations are well set and mostly we need not to touch them. From it, it identify main Type Script file is `src/main.ts` as follow:

```
1 import { enableProdMode } from '@angular/core';
2 import { platformBrowserDynamic } from '@angular/platform-browser-
  dynamic';
3
4 import { AppModule } from './app/app.module';
5 import { environment } from './environments/environment';
6
7 if (environment.production) {
8   enableProdMode();
9 }
10
11 platformBrowserDynamic().bootstrapModule(AppModule)
12   .catch(err => console.error(err));
```

Here, first few lines are `import`. It is a Type Script feature in include other files in the program. After them, it sets environment (development, testing, staging, production, or as you wish). Then at line 11, `bootstrapModule` inform Angular CLI that `AppModule` is the first module to be initialized and actually is the starting point of our application. All files of `XyzModule` are stored in folder `src/xyz`. Thus, files of `AppModule` are stored in folder `src/app`.

To load App module, Angular CLI will first check `src/app/app.module.ts` file, which, by default, looks as follow

```
1 import { BrowserModule } from '@angular/platform-browser';
2 import { NgModule } from '@angular/core';
3
4 import { AppComponent } from './app.component';
5
6 @NgModule({
7   declarations: [
8     AppComponent
9   ],
10  imports: [
11    BrowserModule
```

```
12 ],
13 providers: [],
14 bootstrap: [AppComponent]
15 })
16 export class AppModule { }
```

Here we get first important lesson about angular, Type Script part of this file contains import statements (first four line) and an empty class (last line).

Line 6-15 looks like doc-block, or metadata about the class. In Angular, it is called decorator. Module Decorator, to be exact as it starts with `@NgModule`, which is imported from angular core library in line 2. It takes a Java Script object as an argument. This Java Script object contains few few important properties, most important here is `bootstrap` (line 14). This tells Angular CLI which component should be loaded first. For now, our application contains just one module and one component but as we start making complex applications, we will have multiple modules and each module will contain several components.

Now angular knows it need to start with AppComponent, it will further check `app.component.ts` file.

```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-root',
5   templateUrl: './app.component.html',
6   styleUrls: ['./app.component.css']
7 })
8 export class AppComponent {
9   title = 'HelloWorldApp';
10 }
```

Here, we can find Component Decorator. Most important to note is `selector`, which is `app-root`. If you remember, we saw `<app-root></app-root>` on line 11 of generated HTML. It tell angular, it need to replace `app-root` tag with `app component` and code of `app.component.html` will replace `app-root` on browser through generated java script.

This is just high level view of how angular applications work. We are not ready to learn more about components, which are basic building blocks of an angular application.

3 Components; Angular's basic building blocks

Components are the core of an Angular application. Every angular application have at least one component. Every component generally have 4 files:

- TypeScript file (name.component.ts)
 - This is a Type Script class with `@Component` decorator. It defines data and logic for the component
- HTML file (name.component.html)
 - Although having this it is not mandatory as we can define HTML template in component decorator as well, but generally component HTML have multiple lines, which we mostly prefer to define in separate HTML file.
- Styles file (name.component.css)
 - It is style sheet for component's HTML. Like HTML, can can define CSS also in component decorator, but in most cases, to separate different type of code in different files, we mostly define styles in separate css file.
- Tests (name.component.spec.ts)
 - This file contains test cases for the component. We will discuss how to write unit tests for angular application but for most of the training, we will concentrate to understand different angular features. Thus, until we discuss about unit tests, we will not use this file in any component.

3.1 Creating a component

There are two ways to create new components:

1. Manually
2. Using Angular CLI

Let's check both of them one by one.

3.1.1 Creating a component manually.

We already discussed that a component generally have 3 files; TypeScript class, HTML Template and CSS styles.

In our HelloWorldApp, let's create a new component to display “Manual component”.

As we discussed in [Section 2.4 How our application works?](#), a module bootstraps only one component and we already bootstrapped “AppComponent”. All other components, now should be called or loaded from AppComponent.

Separate folder for each component: A complex angular application contains many components. Thus, as good a good coding practice, we should create separate folder for each component and they should follow hierarchical order. For example, if app component is going to load “Manual-Component”, it must be in “app/manual” folder.

For this example, we are going to name our new component as `ManualComponent`. Following coding best practices of angular, let's create folder `manual` in `app` folder. Then we need to create 3 files

src/app/manual/manual.component.ts

```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-manual',
5   templateUrl: './manual.component.html',
6   styleUrls: ['./manual.component.css']
7 })
8 export class ManualComponent { }
```

As we discussed, Type Script is super set of Java Script. Like Java Script, if we include a TS file into another TS file, included TS file needs to export something. Thus, on line 8, we are exporting a class named `ManualComponent`. However, we need not to code anything right now so this class is empty.

From angular's point of view, to make this file as component, we need to define component decorator. Component is something provided by Angular, not Type Script. Thus, in first line, we are importing Component from angular core library. Once imported, we can use `@Component()` decorator on “ManualComponent” class to tell angular this is a component.

A Component decorator needs three things (as JS object)

- selector
 - It tells angular how other components will refer this component. In other words, it defines the tag which we can use in HTML. To avoid confusion with other HTML tags, as a coding

best practice, we prefix our custom tags with something, mostly with `app-`. Thus, our selector is `app-manual`. However, you are free to choose any name, provided it is unique.

- `templateUrl`
 - It defines the template (HTML) for the component. As discussed earlier, in case we just have one or two line HTML, we can also define HTML directly here. In that case, we define `template`, instead of `templateUrl`. However, either “`template`” or “`templateUrl`” is mandatory.
- `styleUrls`
 - Please note, we have only one template but can have multiple style files for a component. Thus, it is `styleUrls` (Notice “s” at the end). Also, unlike `template`, it is an array, not string.
 - Like `templateUrl`, we can also replace `styleUrls` with `styles`. Any one of them is mandatory.

Lets also define `manual.component.html`, which is ideally just one line of code

```
1 <p>Loaded from Manual Component</p>
```

Similarly, `manual.component.css` is also very simple style sheet

```
1 p {  
2   color: red;  
3 }
```

We defined the component but we are still not using it anywhere. Lets use our new tag in `app.component.html`

```
1 <h1>  
2   Hello {{ title }}!  
3 </h1>  
4 <app-manual></app-manual>
```

Although it seems we are done, we have one final step remaining. Angular still do not know about our new component and will throw an error as soon as it comes across `app-manual` tag in `app component's` `html`. To tell angular about our new component, we must update `app.module.ts` file.

```
1 import { BrowserModule } from '@angular/platform-browser';  
2 import { NgModule } from '@angular/core';  
3  
4 import { AppComponent } from './app.component';  
5 import { ManualComponent } from './manual/manual.component';  
6  
7 @NgModule({  
8   declarations: [  

```

```
9     AppComponent,  
10     ManualComponent  
11   ],  
12   imports: [  
13     BrowserModule  
14   ],  
15   providers: [],  
16   bootstrap: [AppComponent]  
17 })  
18 export class AppModule { }
```

Here, we added line 5, where we are importing our new component. Still, just importing a Type Script file will not work. In line 10, we are actually declaring our new component. Declaration means, angular will not immediately load this new component but will read its component decorator and will know about new tag. As soon as we use new tag in any component (app.component.html in our case), angular will load (create instance of ManualComponent) our component there.

With this, our application now loads our new component

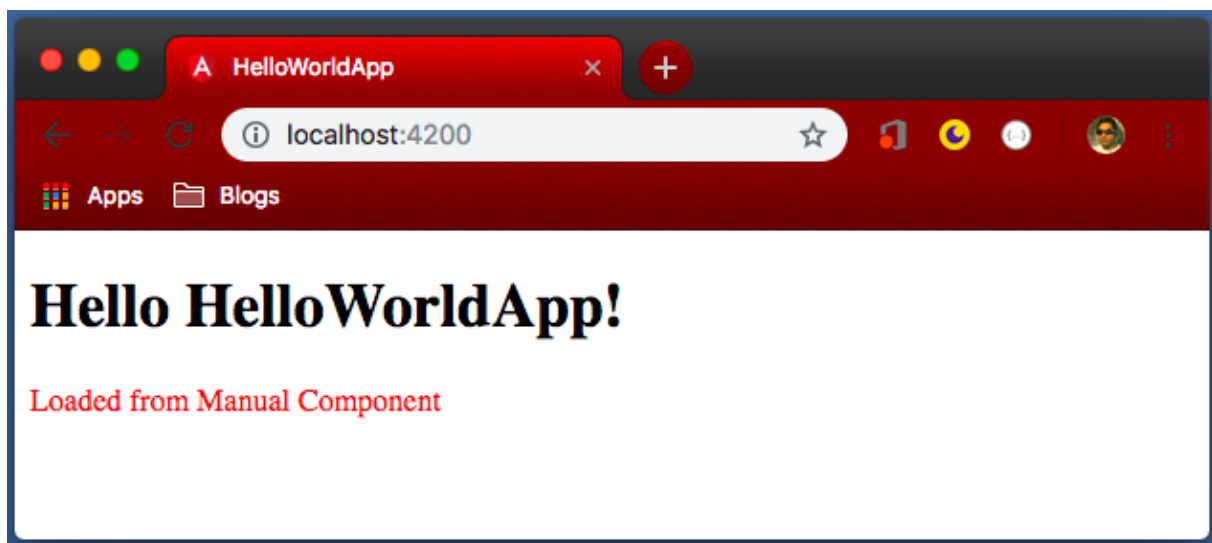


Figure 3.1: Manual Component

3.1.2 Creating a component using Angular CLI.

It is good to understand how angular components works. However, once we understand it, instead of creating components manually, we can use angular CLI to actually create our components. This will fast track component creation, as single command can create all the required files as well as make necessary changes in module.ts file.

Angular CLI starts with `ng` command. We already saw `ng new` and `ng serve` commands. They both were Angular CLI commands.

Command to generate new component is `ng generate component <name>`. Angular CLI also comes with lot of short-cut command. Short cut command to generate component is `ng g c <name>`. Let's run `ng generate component cli` command to create new component "CliComponent" through Angular CLI.

```
1 ng generate component cli
2 CREATE src/app/cli/cli.component.css (0 bytes)
3 CREATE src/app/cli/cli.component.html (22 bytes)
4 CREATE src/app/cli/cli.component.spec.ts (607 bytes)
5 CREATE src/app/cli/cli.component.ts (257 bytes)
6 UPDATE src/app/app.module.ts (466 bytes)
```

As we can see in output, it generated 4 files and updated `app.module.ts` file. Let's also update "`cli.component.html`" and "`cli.component.css`" to match our manual component style.

cli.component.html

```
1 <p>Loaded from CLI Component</p>
```

cli.component.css

```
1 p {
2   color: blue;
3 }
```

We also need to add our selector "`app-cli`" (auto generated, check in `ts` file) in "`app.component.html`"

app.component.css

```
1 <h1>
2   Hello {{ title }}!
3 </h1>
4 <app-manual></app-manual>
5 <app-cli></app-cli>
```

Now our generated output is as follow

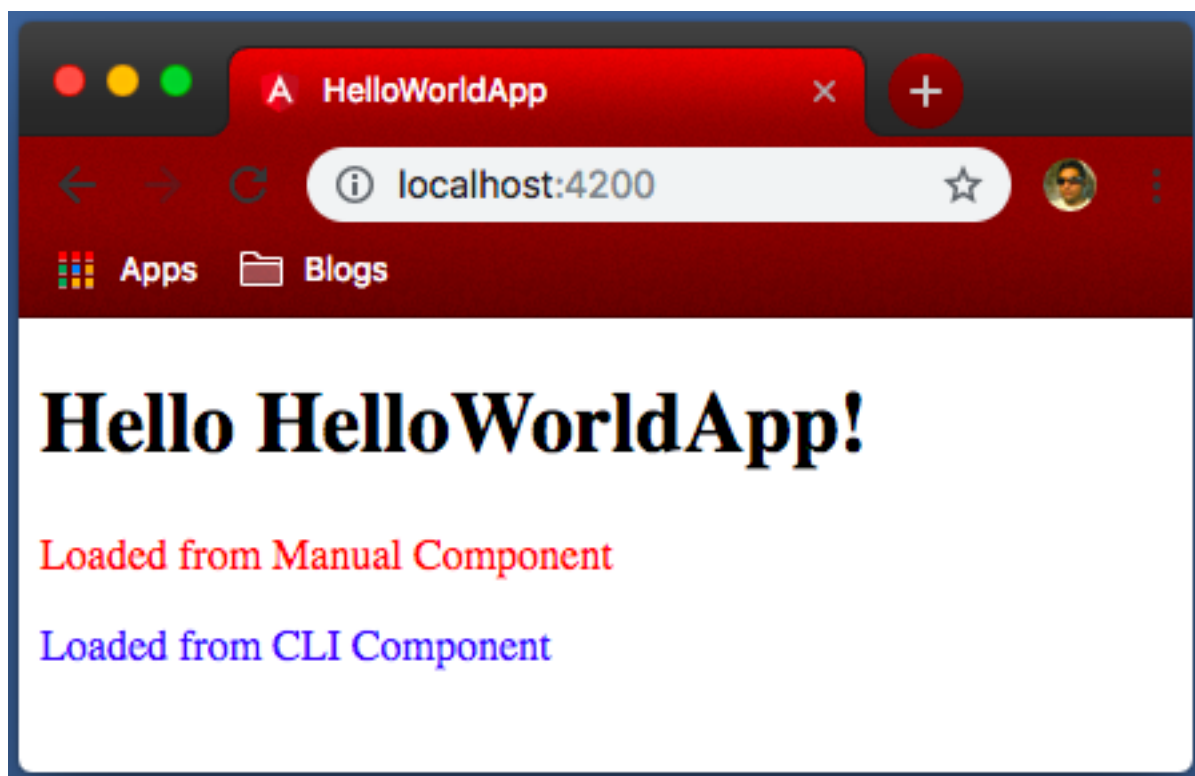


Figure 3.2: Cli Component

3.2 Component stylesheet

In above image, we can see manual component is in red and cli component is in blue, exactly as we defined in our css.

However, if you were checking closely, you might be wondering that both components have `<p>` tag, then how angular manage to define separate styles for single tag. To understand this, lets use chrome's dev tools to check generated HTML code as shown in following image.

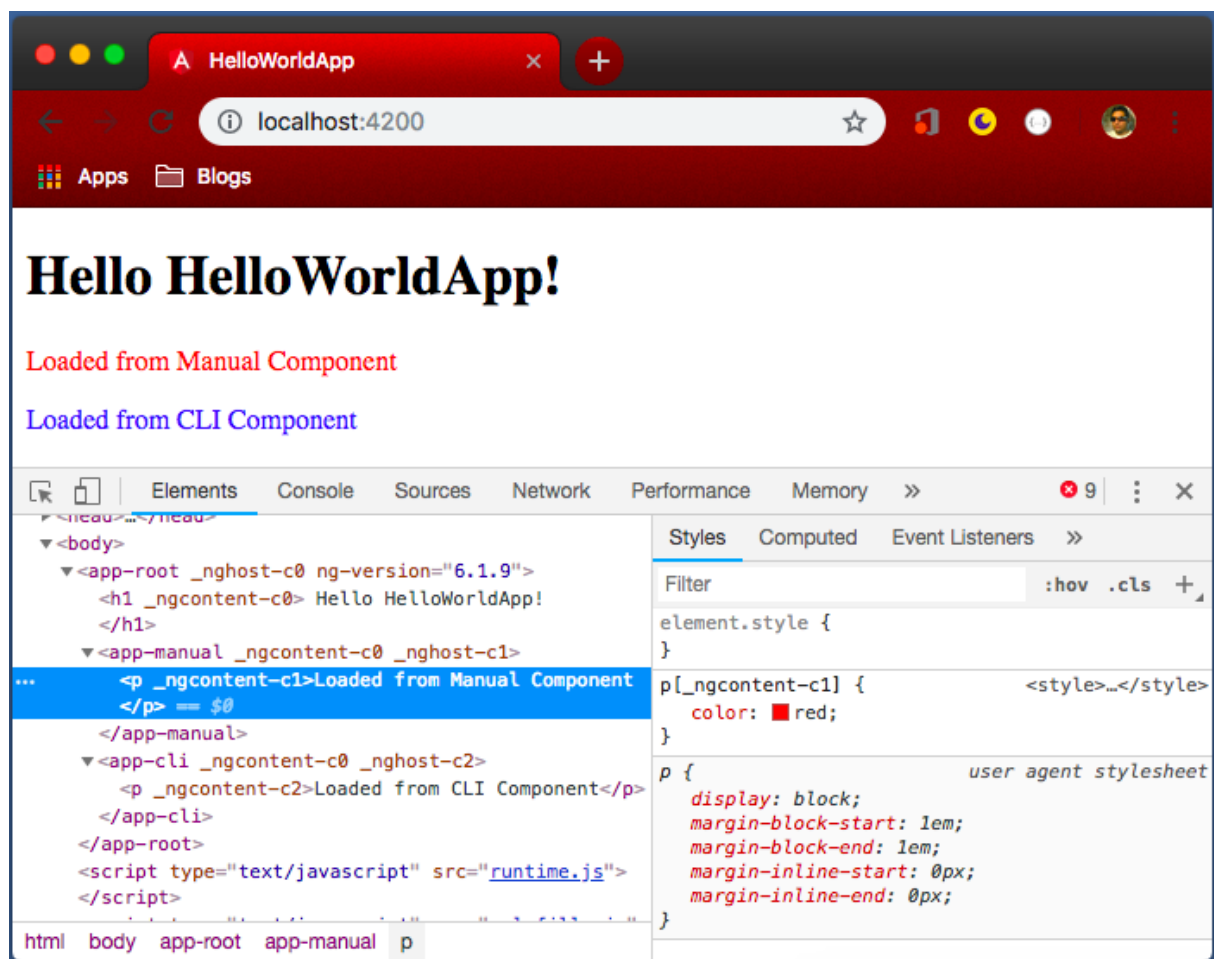


Figure 3.3: Dev tools

As you might notice, `app-root` tag have a special attribute `_ngghost-c0` and all the elements within it, including `app-manual` and `app-cli`, have attribute `_ngcontent-c0`. Here, `c0` is the name angular provided to `app-root`.

Similarly, `app-manual` and `app-cli` have attributes `_ngghost-c1` and `_ngghost-c2` and all element within them (only “p” tag) have attributes `_ngcontent-c1` and `_ngcontent-c2` respectively. Thus, angular provided them name as `c1` and `c2`.

This way, angular may identify different components. In the same image, “p” tag under `app-manual` is selected and its styles are loaded on right hand side. Please note, angular have not generated generic style for `p` tag but for `p[_ngcontent-c1]` tag. In this way, angular makes sure the style we defined for an individual components, affects only that component. We will later learn how to define generic style that may be applied to all elements, regardless of component.

3.3 Data Binding

Data binding is a way of communication between Type Script (Business Logic) and HTML Template (Presentation). There are four types of data binding in Angular.

	Data Binding	Syntax	Direction
1	String Interpolation	{{ variableName }}	TS -> HTML
2	Property Binding	[property]="data"	TS -> HTML
3	Event Binding	(event)="expression"	HTML -> TS
4	Two-way Binding	[(ngModel)]="variableName"	HTML <-> TS

3.3.1 String interpolation

We already seen the example of string interpolation. We devined variable `title` in `app.component.ts` of our HelloWorldApp. Later, we displayed it in `app.component.html` using string interpolation `{{ title }}`.

As we can print variable in TS file to HTML file, in string interpolation, data flows from Type Script to HTML.

3.3.2 Property Binding

Like String Interpolation, data flows from Type Script to HTML for Property Binding as well. However, here property means some HTML property like `img's src` propoerty, `button's disabled` property, etc. Thus, if we want to show value of some variable on HMTL page, use string interpolation. However, if you want to control property (or attribute) of some HTML element, Property binding is used.

For both "String Interpolation" and "Property Binding" data flows from Type Script to HTML.

String Interpolation: Display valus of TS variable on DOM

Property Binding: Assign value of TS variable to HTML property

For example, let's set a new variable `name` in our `app.component.ts`.

app.component.ts

```
1 import { Component } from '@angular/core';
2
```



```
3 @Component({
4   selector: 'app-root',
5   templateUrl: './app.component.html',
6   styleUrls: ['./app.component.css']
7 })
8 export class AppComponent {
9   title = 'HelloWorldApp';
10  name = 'Kapil';
11 }
```

Now with `name` set, lets display it in HTML, but with property binding.

```
1 <h1>
2   Hello {{ title }}!
3 </h1>
4 <div>
5   <span>Hello </span>
6   <span [innerHTML]="name"></span>
7 </div>
8
9 <app-manual></app-manual>
10 <app-cli></app-cli>
```

On line 5, we could although use String Interpolation like `{{ name }}`, we used Property Binding to show the concept.

In above example, both String Interpolation and Property Binding could be used, we can select anyone based on our requirements.

To understand properties and events of HTML, MDN (Mozilla Developer Network) Web Docs^a is one of the best resources. Google is also developer's best friend. As an Angular developer, you need to manipulate HTML DOM a lot, thus understanding DOM, HTML elements and their properties and events is necessary. It is like vocabulary for Frontend developer. However, if you do not understand them right now, don't worry, understanding them is easy and with time and experience, you will be comfortable. Just keep increasing your vocabulary.

^a<https://developer.mozilla.org/en-US/>

3.3.3 Event Binding

DOM regularly fire events based on user activity/input. Some example of events are:

- User moves mouse.

- Mouse enters/leaves an element (hover).
- Mouse (left/right/wheel) clicked.
- Key is pressed/down/up

As a frontend developer, we need to react on these events and this is where event binding is useful. As you can assume, user activity happens on HTML (contents visible to user) but code to react on these events is business logic, which is written in Type Script. Thus, event binding sends data from HTML to Type Script.

For example of Event Binding, let's add a input box on our app component. Whenever user enters something there, default name "Kapil" should change to the name entered by user.

app.component.html

```
1 <h1>
2   Hello {{ title }}!
3 </h1>
4 Enter your name <input type="text" (input)="onChange($event)" />
5 <div>
6   <span>Hello </span>
7   <span [innerHTML]="name"></span>
8 </div>
9
10 <app-manual></app-manual>
11 <app-cli></app-cli>
```

As you can notice on line 4, there is a input of type text. However, it also have `(input)="onChange($event)"`. This is event binding. One event on `input type=text`¹ tag is `input`².

We are binding input event. While binding event, we need to tell Angular what to do when this event is fired. We want to call a function `onChange` and we also want to pass event data. Event data can be passed with special reserved variable `$event`. Angular creates that variable whenever an event happens. Now, we can handle this event on Type Script as follow.

app.component.ts

```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-root',
5   templateUrl: './app.component.html',
6   styleUrls: ['./app.component.css']
7 })
8 export class AppComponent {
9   title = 'Angular';
10 }
```

¹<https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input/text>

²<https://developer.mozilla.org/en-US/docs/Web/Events/input>

```
7  })
8  export class AppComponent {
9      title = 'HelloWorldApp';
10     name = 'Kapil';
11
12     onChange(event: KeyboardEvent) {
13         this.name = (<HTMLInputElement>event.target).value;
14     }
15 }
```

We added `onChange` function in Type Script between line 12-14.

Few notes about Type Script and Type Script functions:

- Type Script allow to define types of variable, that is, unlike Java Script, it can act as “strictly typed language”. That’s how it got its name.
- To define the type of a variable, syntax is `variableName: type`.
- In Type Script, we can define functions by writing function name, followed by () for optional parameters and {} for function body.
- Type Script can also define return type.
- Complete Syntax:

```
1  functionName(parameterOne: Type, parameterTwo: type): returnType {
2      // function body;
3  }
```

- If we need to refer any variable of other function defined in class, we can do this using `this` keyword. This should be clear if you worked with any object oriented programming language. Unfortunately, OOPs is out of scope for this course. (PS: I’m planning OOP video as well. Keep checking my YouTube channel^a; Sorry for self-promotion but couldn’t resist ;)

^a<http://bit.ly/kapilyoutube>

`onChange` function is one line function, which is crystal clear, we are assigning value of our input tag to variable name. However, looking at syntax, you may have following questions:

- How can I know event is of type “KeyboardEvent”?
- What is HTMLInputElement?
- What is (event.target)?
- From where should I learn all these alien looking syntax?

Valid questions, specially if you didn’t have much experience with Java Script. You will learn all of them with time and experience but I have few suggestions here, which could help you to self identify thing.

Let's replace `onChange` function as follow and check the output in console (select developer tools in chrome and open `console` tab)

```
1 onChange(event: Event) {  
2     console.log(event.constructor.name);  
3     console.log(event.target.constructor.name);  
4     console.log((<HTMLInputElement>event.target).value);  
5     this.name = (<HTMLInputElement>event.target).value;  
6 }
```

Here, we added 3 `console.log` statements. It is java script syntax to log some value through JS. These logs can be seen in console of browser. Console outout should be like (Assuming you pressed "K"):

```
1 InputEvent  
2 HTMLInputElement  
3 K
```

If you have an object in java script, you can get it's class name but adding "`.constructor.name`" to the object. Thus, first log will tell us name of class of `event`, which is `InputEvent`, you can use it to write type of parameter.

Now (missing in code) you can do `console.log(event)` to inspect different properties of "event" object. You will notice, it have a property "target" which is again on object. When you extract "target" you will find data entered in text box against `value` property.

Still, `this.name = event.target.value`; will not work. You will get error "value is not property of InputElement". Here, second console statement `console.log(event.target.constructor.name)`; will help you. It will tell us `event.target` is an object of `HTMLInputElement` class but our error mentioned "InputElement". Thus, we specifically need to tell Angular that `event.target` is an object of `HTMLInputElement`.

We can cast (change) object by syntax `<castTo>object`. Thus, `<HTMLInputElement>event.target` will convert `target` to object of `HTMLInputElement`. Please note () for this whole syntax as we want to call `value` on casted object. Now `(<HTMLInputElement>event.target).value` must not looked like alien code and you should be able to solve similar problem in future.

Once you solve the problem, don't forget to remove `console.log` statements.

Now with these changes, our application looks as follow:

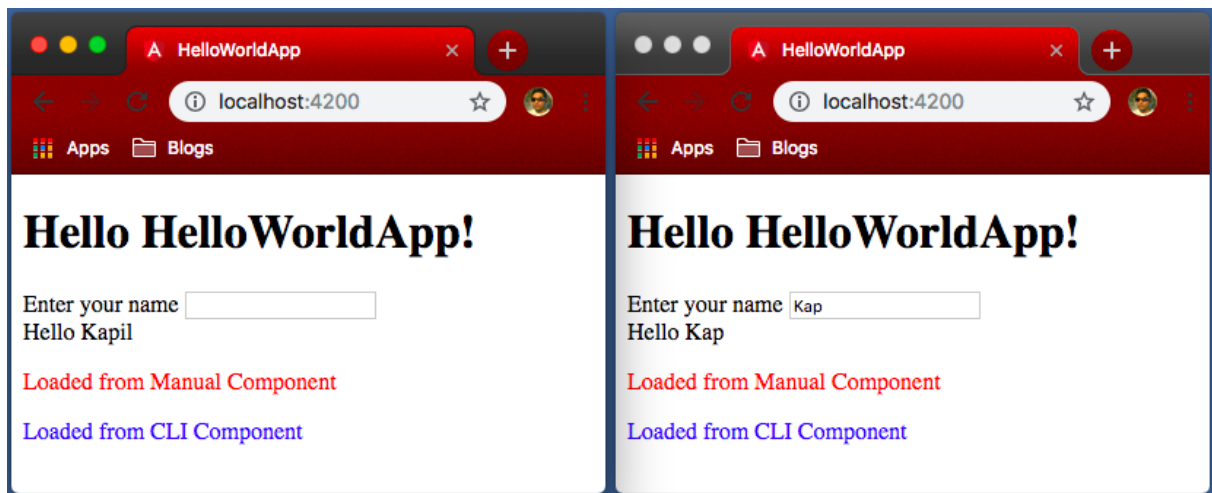


Figure 3.4: Event Binding

Here, left image shows default value of name and right image shows changes when we enter the name.

3.3.4 Two way binding

As the name suggest, two way binding synchronize data both Type Script to HTML and vice versa. Syntax for two way binding is `[(ngModel)]="variableName"`.

Let's see example of Two-Way binding. In our Hello world app, let's also include a text box for age and change text to "Hello Nane, you are x years old".

app.component.html

```

1 <h1>
2   Hello {{ title }}!
3 </h1>
4 <p>Enter your name <input type="text" (input)="onChange($event)" /></p>
5 <p>Enter your age <input type="text" [(ngModel)]="age" /></p>
6 <div>
7   <span>Hello </span>
8   <span [innerHTML]="name"></span>.
9   <span>You are {{ age }} years old.</span>
10 </div>
11
12 <app-manual></app-manual>
13 <app-cli></app-cli>

```

On line 5, we added a new test and input box. On input box, we have [(ngModel)]=age. Here, as we will see, age is a variable in TypeScript that we are also displaying on Line 9.

app.component.ts

```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-root',
5   templateUrl: './app.component.html',
6   styleUrls: ['./app.component.css']
7 })
8 export class AppComponent {
9   title = 'HelloWorldApp';
10  name = 'Kapil';
11  age = 0;
12
13  onChange(event: KeyboardEvent) {
14    this.name = (<HTMLInputElement>event.target).value;
15  }
16 }
```

Here, on line 11, we introduced new variable, and initialized it by “0”. When we reload our form, we can note “0” is prepopulated in our new input field. But wait, if you try to reload now, you will get an error.

ngModel is a functionality not available in angular core but it is provided by Forms module of angular. Thus, before we can use two way binding, we must import forms module.

app.module.ts

```
1 import { BrowserModule } from '@angular/platform-browser';
2 import { NgModule } from '@angular/core';
3 import { FormsModule } from '@angular/forms';
4
5 import { AppComponent } from './app.component';
6 import { ManualComponent } from './manual/manual.component';
7 import { CliComponent } from './cli/cli.component';
8
9 @NgModule({
10   declarations: [
11     AppComponent,
12     ManualComponent,
13     CliComponent
14   ],
```

```
15   imports: [  
16     BrowserModule,  
17     FormsModule  
18   ],  
19   providers: [],  
20   bootstrap: [AppComponent]  
21 })  
22 export class AppModule { }
```

Here, we are importing FormsModule from angular forms on line 3. As we discussed, import is type script feature, and angular will not know about it until we add it under “imports” array. We did that in line 17 above. Now our page will look like follow with age prepopulated and it will also change the text as soon as we change in age text box.

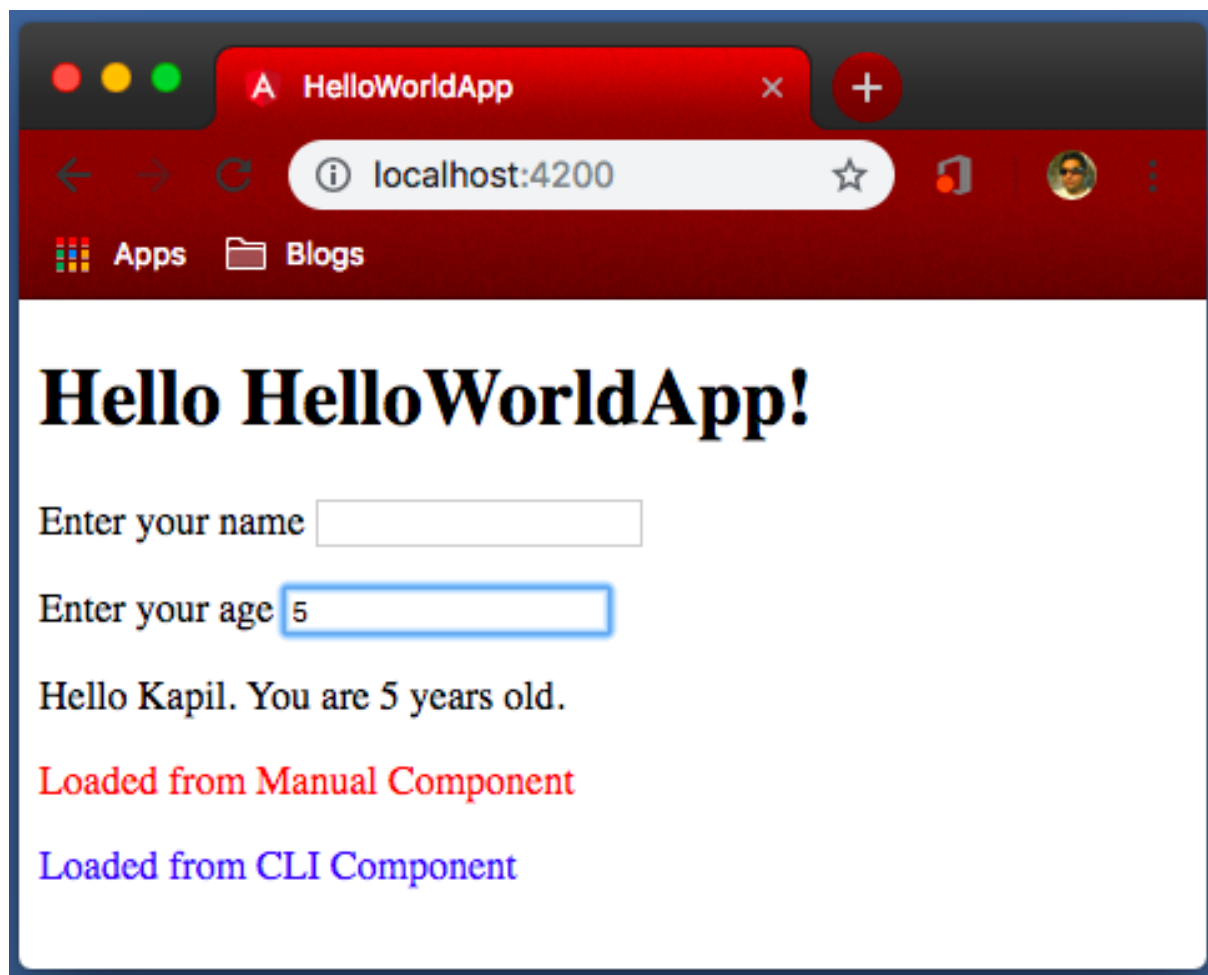


Figure 3.5: Two Way Binding

We are progressing well but now it is time to go little more advanced and Hello world application is not sufficient for it. You might have noticed that we were able to demonstrate the concept of databinding in hello world app, it do not makes lot of sense.

Thus, let's create a new application, that we will develop while we learn angular. Before we continue remaining topics of components, lets discuss requirements of our course project. Later, whatever new topics we learn, we will immediately apply to our course project.

4 Course Project

Making a practical application is always helpful while learning some new programming language or a framework. The application we make while learning need not to be a client project. We can be our own client and think about an application, that we can use. Thus, to maximize our learning, we will be making an application, that we may use in our daily life.

4.1 Getting things done

Have you ever said “I could not do a thing because I was out of time”? Most of us must have said that; some might say many times a day and others once in a while.

David Allen has written a best seller book named Getting things done, the art of stress-free productivity¹. In business world, this book is considered as bible of productivity. In India, it can be purchased from Amazon² or Flipkart³ or other sites or book stores like crossword etc.

However, we need to make an application, for practice and our own use. If you do not want to read whole book, a good summary is given at Four minute books⁴. Let's discuss a quick summary for our application here as well.

4.1.1 Getting things done summary.

1. Capture, Collect what has your attention
 - Use an in-basket, note pad, or voice recorder to capture 100% of everything that has your attention. Little, big, personal and professional—all your to-do's, project, things to handle or finish.
 - For our application, we will place all the tasks as “ToDo” in “input” category, which is also our default category.
2. Clarify, Process what it mean

¹<https://gettingthingsdone.com>

²<https://www.amazon.in/Getting-Things-Done-Stress-free-Productivity/dp/0349410151>

³<https://www.flipkart.com/getting-things-done/p/itm3zrk9cpvyhu4>

⁴<https://fourminutebooks.com/getting-things-done-summary/>

- Take everything that you capture and ask: Is it actionable? If no, then trash it, incubate it, or file it as reference. If yes, decide the very next action required. If it will take less than two minutes, do it now. If not, delegate it if you can; or put it on a list to do when you can.
 - This is thinking step so our application will not have it. We can just show all the items in a particular category.
3. Organize, Put it where it belongs
 - Put the action remainders on the right list. For example create lists for the appropriate categories—call to make, errands to run, emails to send etc.
 - In our application, we must provide a way to add/edit/delete categories, which will act as lists.
 4. Reflect, Review frequently
 - Look over your lists often as necessary to determine what to do next. Do a weekly review to clean up, update your lists, and clear your mind.
 - This is again a thinking step. We will provide support to list ToDo based on different criteria like by category, by status, by due date (Not in course project but assignment), etc.
 5. Engage, Simply do
 - Use your system to take appropriate actions with confidence.
 - This is action or actually doing the task. Our application can't help there but just to remind to do things.

4.2 ToDo App

We are making out ToDo App to achieve above steps through a web application.

4.2.1 Home page or ToDo List Page

ToDo App			ToDo List	Add ToDo	Category List	Add Category
Angular session 1			Angular Learning			
<input checked="" type="checkbox"/>	Angular session 1 Assignment	Angular Learning	Edit	Delete		
<input checked="" type="checkbox"/>	Angular session 2	Angular Learning	Edit	Delete		
<input checked="" type="checkbox"/>	Angular session 2 Assignment	Angular Learning	Edit	Delete		
<input checked="" type="checkbox"/>	Angular session 3	Angular Assignment	Edit	Delete		
<input checked="" type="checkbox"/>	Angular session 3 Assignment	Angular Learning	Edit	Delete		
<input checked="" type="checkbox"/>	Angular session 4	Angular Assignment	Edit	Delete		
<input checked="" type="checkbox"/>	Angular session 4 Assignment	Angular Learning	Edit	Delete		
<input type="checkbox"/>	Schedule Angular meetup 2	PHPReboot	Edit	Delete		
<input type="checkbox"/>	Commit course project code of Angular meetup 1	PHPReboot	Edit	Delete		
<input type="checkbox"/>	Update Angular Quick Notes ebook	Ebooks	Edit	Delete		

Figure 4.1: ToDo List

Above image is wireframe, actual page might look little different in design.

Our app have a top bar with Text logo on the left and 4 links on the right. Links are:

- ToDo List
 - This is the landing (this) page. Current page must be highlighted in menu as well. We do this by changing color of current page link to green.
- Add ToDo
 - This is the form to add new ToDo list.
- Category List
 - Like ToDo list, this page will show list of available categories.
- Add category
 - This page will contain a form to add new categories.

After Menu bar at top, which is common for all the pages, we have ToDo list on landing page. This page contains a table with 4 columns:

- Checkbox to show done/not done task
- ToDo - Task to be done.
- Category under which task is listed.

- Actions, For now, we have two action for each task, edit and delete. Delete will simply delete the ToDo. Edit will open the task on Add Task page but with fields pre-populated.

4.2.2 Add ToDo page

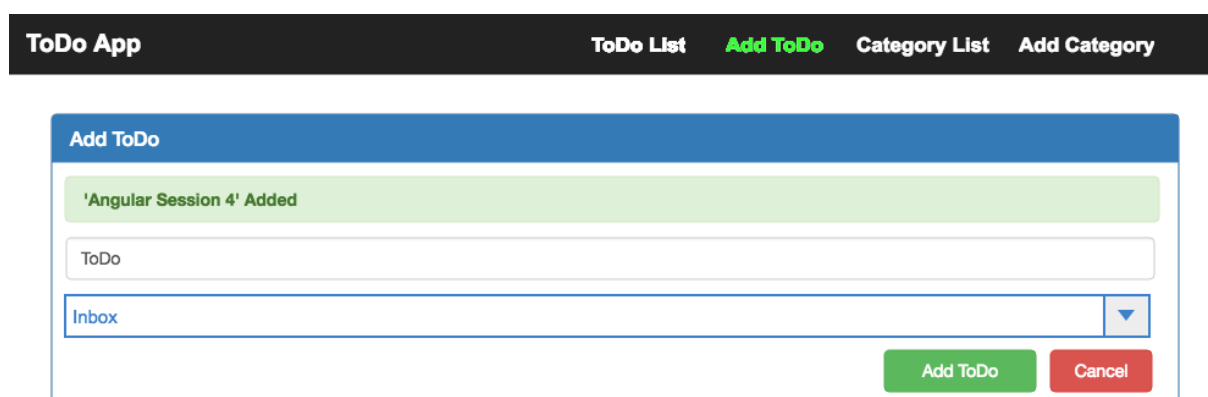


Figure 4.2: Add ToDo

Add ToDo page simply contains a form with a text field to enter ToDo, Drop down to select category and “Add ToDo” and “Cancel” buttons.

4.2.3 Category list and Add Category

Category list and add category pages are similar to ToDo list and Add ToDo pages but they will be dealing with categories. We will complete ToDo pages and category pages will be there as assignment.

4.3 Creating assignment project

We have already seen how to create a new Angular application, how to create components and data binding. Before learning anything new, let's use these concepts to start building our application.

4.3.1 Creating ToDo App

Let's create a new project `ToDoApp`, we already know the command:

```
1 ng new ToDoApp
```

This will create a new angular project. Open your project in your favorite editor/IDE. In this course, I'll be using PHP Storm, which is similar to Web Storm, with added support for PHP. PHP Storm is an IDE developed by Jet Brains, who is also sponsor of PHP Reboot community and provide free license of any Jet Brains IDEs to be given as swags during meetups. In Corporate training of Angular on October 27th, 2018, two licenses will be raffled out. Web Storm, PHP Storm or most of other Jet Brains IDEs have excellent support for JavaScript frameworks including Angular.

However, if you need any free alternative, Visual Studio Code is supposed to have excellent support for Angular and is favorite editor of many Angular developers. You may also choose any other IDE/Editor, which you feel most comfortable with.

4.4 Using Bootstrap

Bootstrap is an open source CSS framework, which makes it very easy to work with CSS. We will be using Bootstrap in our ToDo App but you are free to use any CSS framework or making your own custom CSS from scratch. For ToDo App, we will be using Bootstrap 4, which is recently released and is the latest version of bootstrap.

We will install bootstrap through npm. To do that, run following command

```
1 npm install --save bootstrap@4
```

“npm” will warn about installing JQuery and other JS library. However, We need bootstrap only for CSS, not JQuery, so we can avoid that warning.

Both Angular and JQuery are JS frameworks and cross each other's way. Thus, even though it is technically possible, we generally avoid using JQuery with Angular.

Any package installed through npm goes in “node_modules” folder. This folder contains many npm packages, which helps during development so every library available in “node_modules” is not used in final angular application. We need to specifically tell angular to include bootstrap in our application.

This is where “angular.json” file at root is useful. In “angular.json” file, add our bootstrap's css file “node_modules/bootstrap/dist/css/bootstrap.min.css” under “styles” array.

```
25 "styles": [  
26     "node_modules/bootstrap/dist/css/bootstrap.min.css",  
27     "src/styles.css"  
28 ],
```

Please note, there are multiple “styles” array in the file. We need to add bootstrap in styles array under “build”.

With this, now bootstrap css is added in our project, which you can check under page source and start using bootstrap for styling.

4.5 Planning Components

It is always important to plan out our components. We can ideally code every thing on single component but this is obviously not a good idea. Component help us to separate out concerns and write reusable code that can be reused at other places. We will see example of reuse of component in our course project. Let’s divide our components as follow:

- AppComponent
 - HeaderComponent
 - TodoComponent
 - * add
 - * list
 - CategoryComponent
 - * add
 - * list

This gives us basic following layout with bootstrap

app.component.html

```
1 <!-- Header -->
2 <div class="container">
3   <!-- ToDo -->
4   <!-- Category -->
5 </div>
```

Here, commented “Header”, “Add ToDo”, and “ToDoList” represents selector of respective componente. Let’s start filling place holders by creating required components.

4.6 Header component

Let’s create now component through Angular CLI.

```
1 ng g c common/Header --no-spec
```

Here, “-no-spec” option tell Angular CLI that we do not want to create spec.ts file. We can start by making our navigation in new component

app/common/header/header.component.html

```
1 <nav class="navbar navbar-dark bg-dark">
2   <a class="navbar-brand" href="#">ToDo App</a>
3   <div class="navbar-nav d-flex flex-row">
4     <a class="nav-item nav-link active mx-3" href="#">ToDo List</a>
5     <a class="nav-item nav-link mx-3" href="#">Add ToDo</a>
6     <a class="nav-item nav-link mx-3" href="#">Category List</a>
7     <a class="nav-item nav-link mx-3" href="#">Add Category</a>
8   </div>
9 </nav>
```

Now let's load new component in our app component

app.component.html

```
1 <app-header></app-header>
2 <div class="container">
3   <!-- ToDo -->
4   <!-- Category -->
5 </div>
```

That's all, our header is ready. Only issue, right now, links are not working, but we will soon make them working.

4.7 ToDo Component

Let's create new component ToDoComponent

```
1 ng g c ToDo --no-spec
```

We first need to update app.component.html to include new ToDo component

app.component.html

```
1 <app-header></app-header>
2 <div class="container">
3   <app-to-do></app-to-do>
4   <!-- Category -->
5 </div>
```

Now, we need to update “to-do.component.ts” to get our data. For now, let us hardcode data in TypeScript file. As we progress in the course, we will see how to handle data in better way.

app/to-do/to-do.component.ts

```
1 import { Component, OnInit } from '@angular/core';
2
3 @Component({
4   selector: 'app-to-do',
5   templateUrl: './to-do.component.html',
6   styleUrls: ['./to-do.component.css']
7 })
8 export class ToDoComponent implements OnInit {
9
10   todos = [
11     {
12       'name': 'Angular Session 1',
13       'done': true,
14       'category': 'Angular'
15     },
16     {
17       'name': 'Angular Session 1 Assignment',
18       'done': true,
19       'category': 'Angular'
20     },
21     {
22       'name': 'Angular Session 2',
23       'done': true,
24       'category': 'Angular'
25     },
26     {
27       'name': 'Angular Session 2 Assignment',
28       'done': true,
29       'category': 'Angular'
30     },
31     {
32       'name': 'Angular Session 2',
33       'done': true,
34       'category': 'Angular'
35     },
36     {
37       'name': 'Angular Session 2 Assignment',
38       'done': true,
```



```
39     'category': 'Angular'
40   },
41   {
42     'name': 'Schedule Angular meetup 2',
43     'done': false,
44     'category': 'PHPReboot'
45   },
46   {
47     'name': 'Update Angular quick notes ebook',
48     'done': false,
49     'category': 'ebook'
50   }
51 ];
52
53 constructor() { }
54
55 ngOnInit() { }
56
57 }
```

This is the default TS class generated by CLI, we just added a variable “todos” which is a Java Script array of Objects. Please note, Type Script is super set of Java Script thus, JS code mostly works in TS.

app/to-do/to-do.component.html

```
1 <div class="row">
2   <div class="col">
3     <table class="table">
4       <thead>
5         <tr>
6           <th scope="col">#</th>
7           <th scope="col">ToDo</th>
8           <th scope="col">Category</th>
9           <th scope="col">Actions</th>
10        </tr>
11      </thead>
12      <tbody>
13        <tr *ngFor="let todo of todos">
14          <td><input type="checkbox" class="form-control" [checked]="
15            todo.done"></td>
16          <td>{{ todo.name }}</td>
17          <td>{{ todo.category }}</td>
18          <td>
```

```
18         <button class="btn btn-info mr-2">Edit</button>
19         <button class="btn btn-danger">Delete</button>
20     </td>
21 </tr>
22 </tbody>
23 </table>
24 </div>
25 </div>
```

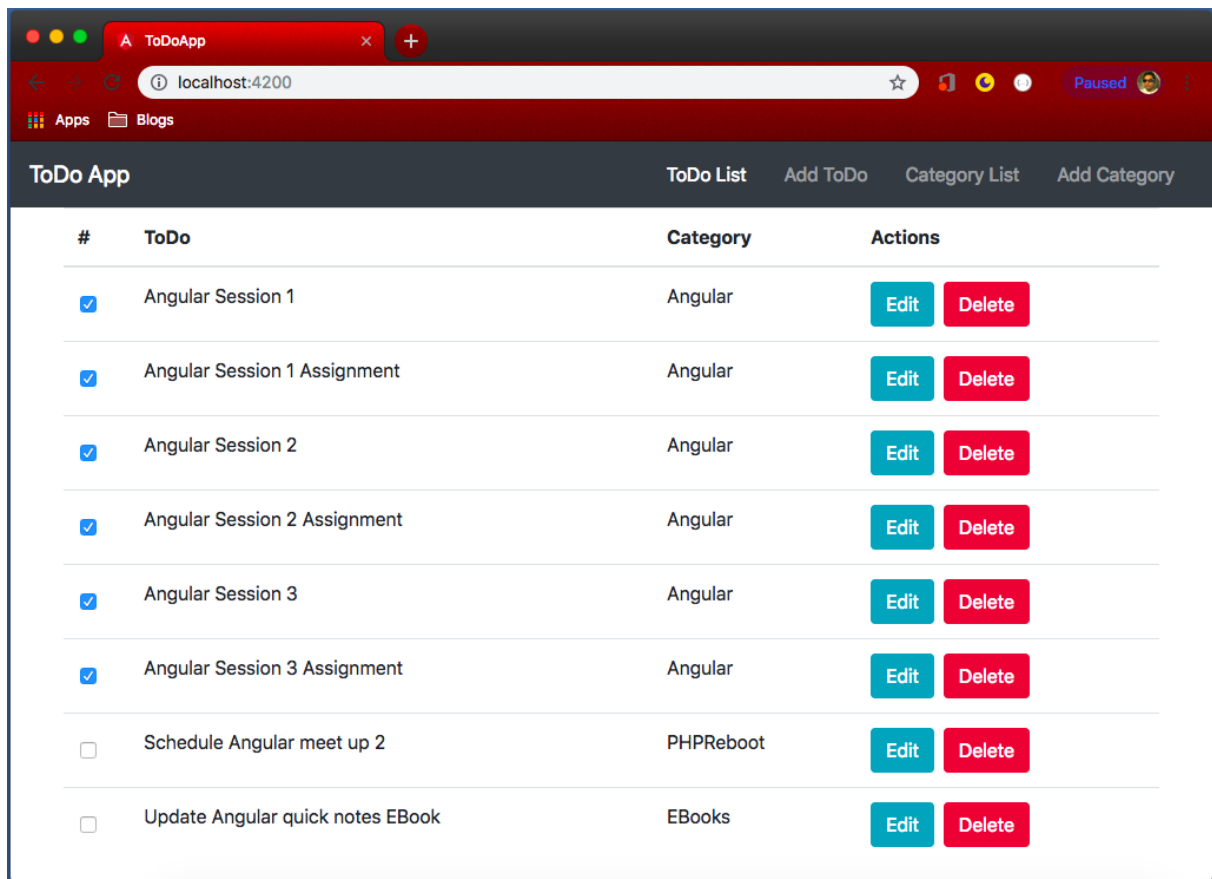
In HTML file, line 1-12 and 22-25 are simple HTML and Bootstrap. Between line 13-21, we want to print table rows.

Line 13 “tr” is simple HTML element to print table row. However, there is a special code `*ngFor="let todo of todos"`. This is called directive. We will discuss more about directives after components. Actually, component is also a directive with template. In short, directives are the instructions in the DOM, just like components instruct angular to change a “tag”, directives are attributes which instruct angular to change DOM.

Here, we are using ‘*ngFor’ directive, which is used to repeat an element for a given instance of time. It is much like “foreach” loop of many programming languages. `let todo of todos` will take one element of todos at a time and save it as `todo`. This will be repeated for every element of the array.

Within “tr” tag, we have simple “td” tags and “todo” is representing individual JS object within todos array. Thus, we can use object properties in JS syntax like “todo.name”, “todo.category” etc.

With this code, our application now looks as follow.

**Figure 4.3:** ToDo Component

4.8 v0.1.0

Code till here is available in branch `v0.1.0`.

5 Components Continued

Now we have our basic course project ready, we can continue working on that while learning new things. We have two goals here:

- Learn and apply new features of angular.
- Learn and apply best practices.

Before we learn a new topic, let's discuss about component structure and make our application better.

5.1 Component structure

Now our application can show hardcoded to-do list. However, we also have categories and both to-do and category have list and create component. As we discussed under “Planning Components” section of last chapter, we should have components divided as follow:

- AppComponent
 - HeaderComponent
 - ToDoComponent
 - * add
 - * list
 - CategoryComponent
 - * add
 - * list

Thus, we should have ListComponent under ToDo component. Let's create a new “ListComponent”.

```
1 ng generate component to-do/list --no-spec
```

Now, since we need to load out ToDo list from ListComponent, let's move code of “to-do.component.html” to “list.component.html”. With this, we now need to load ListComponent in ToDoComponent.

app/to-do/to-do.component.html

```
1 <app-list></app-list>
```

“to-do.component.html” now have just one line of code. However, if we check our page now, there is no task in our list. Reason is obvious, we are looping on “toDos” variable but “list.component.ts” have no such variable.

To make it work again, we need to move “toDos” variable in ListComponent. However, this is not good. ToDoComponent, as the name suggest, is the right place to keep our “toDos” array and perform all operations on it. ListComponent, as name suggest, is just responsible to show the list, not manage it.

Thus, we need a way to pass our “toDos” array from ToDoComponent (parent component) to ListComponent (child component). Time to learn a new feature of Angular.

5.2 Custom Property Binding - Passing data from Parent to Child component.

We already learned about property binding. By property binding, we assign some value to HTML elements.

What is component?

If you see the components in angular, we are creating new HTML elements, for example `<app-list>`. Normal HTML elements contain properties like `checked` property in `<input type="checkbox">`. This is meta data to the a HTML element, which tell if checkbox is selected or not.

Similarly, on our custom HTML elements (components), we can have some meta-data, which we need to pass through HTML.

When one component call other component, calling component is the parent component and called component is the child component.

In our ListComponent, we want some data to be passed by parent component (ToDoComponent). This can be done through property binding. However, since we designed ListComponent, it do not have any default property which could be bind. Let's edit our ListComponent to inform angular that we are expecting some data from parent component.

app/to-do/list/list.component.ts

```
1 import {Component, Input, OnInit} from '@angular/core';
2
3 @Component({
4   selector: 'app-list',
5   templateUrl: './list.component.html',
6   styleUrls: ['./list.component.css']
7 })
```

```
8 export class ListComponent implements OnInit {  
9  
10   @Input() todos;  
11  
12   constructor() { }  
13  
14   ngOnInit() {  
15   }  
16  
17 }
```

We made two changes. On line 10, we added a variable `todos`. Important to note, we also added `@Input()` before it. It is another decorator, which tell angular that this variable will be injected and should be available for property binding.

Input class/decorator is defined by Angular, Type Script have no idea about this class. Thus, as second change in the class, we imported `Input` from `@angular/core` in line 1.

Now we are ready to pass `todos` from `ToDoComponent` to `ListComponent`.

app/to-do/to-do.component.html

```
1 <app-list [todos]="todos"></app-list>
```

Here, we added new property binding `[todos]="todos"`. It could be confusing due to same variable name in both components. Property `[todos]` is the variable in child component (`list.component.ts`) and value `"todos"` is the variable in parent component (`to-do.component.ts`).

Variable names must be defined at class level so above variable names are correct. However, please play with names to understand concept in better way. One way, use property binding as `[todosInListComponent]="todosInToDoComponent"`. Now change the name in respective TS classes to keep it working.

Code till here is committed in branch `v0.1.1`

5.3 Custom event binding - Passing data from child to parent component

In last section, we learned to use custom property binding to pass data from parent to child component. Similarly, we can use custom event binding to pass data from child to parent component.

What is event?

In Java Script, we can catch some event happening on a HTML element. For example, a button element fires “click” event when clicked.

Similarly, child component can fire some custom event, that parent element may catch. We obviously need to define those events.

5.3.1 New course project requirement: Adding Create component

Before we understand custom event binding, let’s discuss new requirement for our course project. Right now, we are just listing ToDo list. Now we need ability to add new ToDo in the list. For this, we need new CreateComponent. Let’s create one.

```
1 ng generate component to-do/create --no-spec
```

In this new form, we want to do two-way binding so first let’s add FormsModule in “app.module.ts”.

app.module.ts

```
1 import { BrowserModule } from '@angular/platform-browser';
2 import { NgModule } from '@angular/core';
3 import { FormsModule } from '@angular/forms';
4
5 import { AppComponent } from './app.component';
6 import { HeaderComponent } from './common/header/header.component';
7 import { ToDoComponent } from './to-do/to-do.component';
8 import { ListComponent } from './to-do/list/list.component';
9 import { CreateComponent } from './to-do/create/create.component';
10
11 @NgModule({
12   declarations: [
13     AppComponent,
14     HeaderComponent,
15     ToDoComponent,
16     ListComponent,
17     CreateComponent
18   ],
19   imports: [
20     BrowserModule,
21     FormsModule
22   ],
23   providers: [],
24   bootstrap: [AppComponent]
25 })
```



```
26 export class AppModule { }
```

We imported FormsModule in line 3 and added in imports array in line 21.

app/to-do/create/create.component.html

```
1 <div class="row">
2   <div class="col">
3     <section class="card card-primary">
4       <div class="card-header">
5         Add new ToDo
6       </div>
7       <div class="card-body">
8         <div class="form-group">
9           <label for="to-do">To Do</label>
10          <input type="text" class="form-control" id="to-do" [(ngModel)]="name" />
11        </div>
12        <div class="form-group">
13          <label for="category">Category</label>
14          <select class="form-control" id="category" [(ngModel)]="category">
15            <option value="inbox">Inbox</option>
16            <option value="angular">Angular</option>
17            <option value="phpreboot">PHPReboot</option>
18            <option value="ebooks">EBooks</option>
19          </select>
20        </div>
21        <button class="btn btn-primary" (click)="onAddToDo()">Add ToDo</button>
22      </div>
23    </section>
24  </div>
25 </div>
```

This is a simple form designed using bootstrap 4. It have a text field, a select field and a button.

****Please note, we are not using HTML form. Do not wrap our form in <form> tag. If you do this, it will not work. Just avoid using**

tag for now, we will learn more about it in “Forms” chapter.**

First things to notice in our HTML file, we used two-way data binding in line 10 and 14. You can correctly assume we are having variable “name” and “category” in our TS class.

Another thing to notice is event binding on button. If you remember event binding, we are calling “onAddToDo” method. Please note, this is not a custom event but a inbuilt event for a button.

You can follow any naming convention. In this book, we are starting any event listener function with **on**. Thus, **onAddToDo** means event for adding ToDo. As a good coding practice, decide any naming conventions with your team and stick to it in whole project.

Now you can probably assume what our TS class should look like.

app/to-do/create/create.component.ts

```
1 import { Component, OnInit } from '@angular/core';
2
3 @Component({
4   selector: 'app-create',
5   templateUrl: './create.component.html',
6   styleUrls: ['./create.component.css']
7 })
8 export class CreateComponent implements OnInit {
9
10   name: string;
11   category: string;
12
13   constructor() { }
14
15   ngOnInit() {
16   }
17
18   onAddToDo() {
19     console.log(this.name);
20     console.log(this.category);
21   }
22 }
```

As expected, we created two variables “name” and “category”, used for two way binding and a function “onAddToDo”, which is simply logging the variables. If you run project now, it will log the variables when button “Add ToDo” button is clicked.

However, we wish to sent these variable to parent Component (ToDoComponent) so that it can add it in “toDos” variable.

5.3.2 Creating custom event

Before we can fire custom event, we need to create one. Let's do it, obviously in CreateComponent Class.

app/to-do/create/create.component.ts

```
1  import { Component, EventEmitter, OnInit, Output } from '@angular/core'
2
3  @Component({
4    selector: 'app-create',
5    templateUrl: './create.component.html',
6    styleUrls: ['./create.component.css']
7  })
8  export class CreateComponent implements OnInit {
9
10     name: string;
11     category: string;
12
13     @Output() todoAdded = new EventEmitter<{
14       name: string,
15       category: string
16     }>();
17
18     constructor() { }
19
20     ngOnInit() {
21     }
22
23     onAddToDo() {
24       this.todoAdded.emit({
25         name: this.name,
26         category: this.category
27       });
28     }
29   }
```

Custom event need two things

- Create an event (Tell angular about possible event that can be fired)
- Fire (emit) the event whenever required.

We create an event or tell angular about possible event from component with `@Output` decorator.

We obviously imported Output class from Angular/core in line 1. Between line 13-16, we are actually creating the event. Any possible event is actually an instance of EventEmitter class, which we also import in line 1.

TypeScript syntax to create EventEmitter (or any other class) is `new EventEmitter()`. However, you may notice a JS object between `<>` before `()`. This is the place where we are telling angular what data to expect in the event. We are telling that our event object will contain two properties; name and category and both will be of type string (TS data type).

Once we defined possible event, we can actually fire (emit) the event. We are doing it in “onAddToDo” method. We do this by calling `emit` method of EventEmitter and obviously need to pass a JS object containing data we promised while creating EventEmitter object.

5.3.3 Catching custom event

Once a custom event is fired, binding to it is similar to event binding.

app/to-do/to-do.component.html

```
1 <app-create (todoAdded)="onToDoAdded($event)"></app-create>
2 <app-list [todos]="todos"></app-list>
```

We added `(todoAdded)="onToDoAdded($event)"` which is just event binding, only difference, “todoAdded” is not a builtin event but we created it. On this event, we are “onToDoAdded” function of TS file.

app/to-do/to-do.component.ts

```
53 onToDoAdded(todo: {
54   name: string,
55   category: string
56 }) {
57   this.todos.push({
58     name: todo.name,
59     done: false,
60     category: todo.category
61   });
62 }
```

At line 53, add above code in to-do.component.ts file. We knew what data our event contains. Thus, as parameter, we told angular that variable `todo` is a JS object which contains name and category property.

On line 57, we simply JavaScript Array's push method to add new data to the existing toDos array.

Since the same array is passed to ListComponent, our list will be automatically updated.

Code till here is committed in branch [v0.1.2](#)

5.4 Passing HTML to child component - ng-content

Any HTML between component selectors are ignored by Angular. This is obvious as we design contents of our component tags with-in component's HTML file. There is no way angular might know where to place HTML with-in component tags.

However, if needed, we can tell angular where to render HTML with-in component tags. We can do this with "ng-content" tag.

Let's assume heading of our "Create" and "List" component is dynamic, which should be rendered by parent component (ToDoComponent). Thus, we first need to define required HTML in selector within "to-do.component.html"

app/to-do/to-do.component.html

```
1 <app-create (todoAdded)="onToDoAdded($event)">Add new ToDo</app-create>
2 <app-list [todos]="todos">ToDo List</app-list>
```

We defined title of component within component tags. However, it will not take effect. For this to take effect, we need to update html file of respective components.

app/to-do/create/create.component.html

```
1 <div class="row">
2   <div class="col">
3     <section class="card card-primary">
4       <div class="card-header">
5         <ng-content></ng-content>
6       </div>
7       <div class="card-body">
8         <div class="form-group">
9           <label for="to-do">To Do</label>
10          <input type="text" class="form-control" id="to-do" [(ngModel)]="name" />
11        </div>
12        <div class="form-group">
13          <label for="category">Category</label>
```

```

14         <select class="form-control" id="category" [(ngModel)]="
           category" >
15             <option value="inbox">Inbox</option>
16             <option value="angular">Angular</option>
17             <option value="phpreboot">PHPReboot</option>
18             <option value="ebooks">EBooks</option>
19         </select>
20     </div>
21     <button class="btn btn-primary" (click)="onAddToDo()">Add ToDo<
       /button>
22 </div>
23 </section>
24 </div>
25 </div>

```

Note, in line 5, where we want to display HTML with component's selector, we added "ng-content" tag. Let's do the same for list component.

app/to-do/list/list.component.html

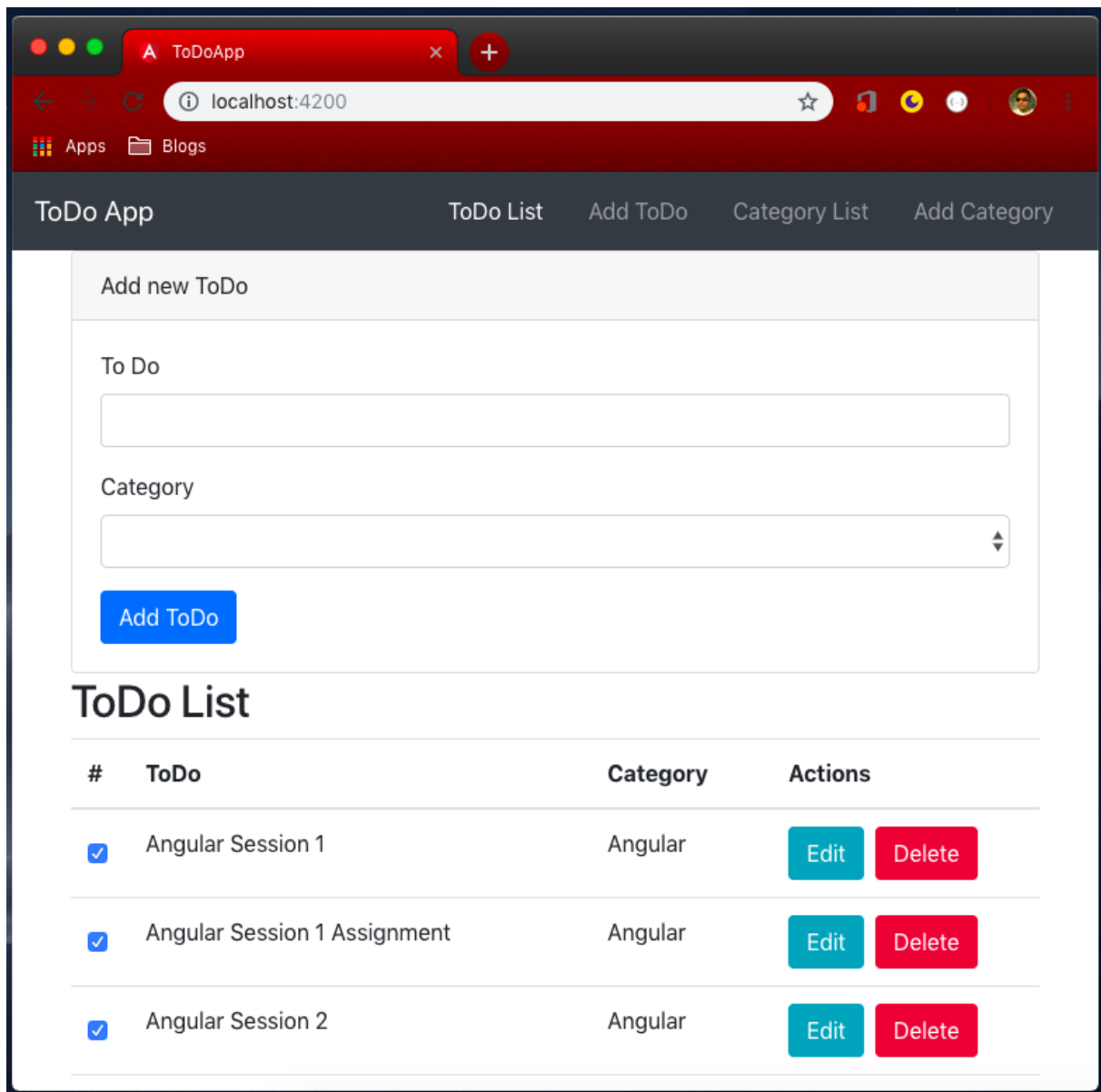
```

1 <div class="row">
2   <div class="col">
3     <h2>
4       <ng-content></ng-content>
5     </h2>
6     <table class="table">
7       <thead>
8         <tr>
9           <th scope="col">#</th>
10          <th scope="col">ToDo</th>
11          <th scope="col">Category</th>
12          <th scope="col">Actions</th>
13        </tr>
14      </thead>
15      <tbody>
16        <tr *ngFor="let todo of todos">
17          <td><input type="checkbox" class="form-control" [checked]="todo
            .done"></td>
18          <td>{{ todo.name }}</td>
19          <td>{{ todo.category }}</td>
20          <td>
21            <button class="btn btn-info mr-2">Edit</button>
22            <button class="btn btn-danger">Delete</button>

```

```
23         </td>
24     </tr>
25 </tbody>
26 </table>
27 </div>
28 </div>
```

Since our list was not having title earlier, we added line 3-5 to add new header and the HTML of component's selector. Our application now looks as follows

**Figure 5.1:** ng-content

Code till here is committed in branch `v0.1.3`

5.5 Local Reference

In our `create.component.html` file, we used two-way binding for name and category variables. We do not want our HTML to be edited from typescript but just to read value of our inout in TS file. Thus, two

way binding is not correct here.

We just want variable in one way; from HTML to Type Script. In Data binding section, we learned HTML->TS flow happens through event binding, that we are already doing. However, we are not sending data from HTML->TS through event binding but through two way binding. We can actually do this using Local reference. Let's first check the code.

app/to-do/create/create.component.html

```
1 <div class="row">
2   <div class="col">
3     <section class="card card-primary">
4       <div class="card-header">
5         <ng-content></ng-content>
6       </div>
7       <div class="card-body">
8         <div class="form-group">
9           <label for="to-do">To Do</label>
10          <input type="text" class="form-control" id="to-do" #name />
11        </div>
12        <div class="form-group">
13          <label for="category">Category</label>
14          <select class="form-control" id="category" #category >
15            <option value="inbox">Inbox</option>
16            <option value="angular">Angular</option>
17            <option value="phpreboot">PHPReboot</option>
18            <option value="ebooks">EBooks</option>
19          </select>
20        </div>
21        <button class="btn btn-primary" (click)="onAddToDo(name,
22          category)">Add ToDo</button>
23      </div>
24    </div>
25  </div>
```

Now we removed two way binding from line 10 and 14. We actually replaced them with “#name” and “#category” respectively. We actually assigned them a name or **Local Reference**.

Local Reference is a name given to some element, that we can later use in same file. We are using this local reference in line 21 `(click)="onAddToDo(name, category)"`, where we are passing values as parameters of event. We obviously need to update our TS file to accept these new parameters.

app/to-do/create/create.component.ts

```
1  import { Component, EventEmitter, OnInit, Output } from '@angular/core'
2
3  @Component({
4    selector: 'app-create',
5    templateUrl: './create.component.html',
6    styleUrls: ['./create.component.css']
7  })
8  export class CreateComponent implements OnInit {
9
10     @Output() toDoAdded = new EventEmitter<{
11       name: string,
12       category: string
13     }>();
14
15     constructor() { }
16
17     ngOnInit() {
18   }
19
20     onAddToDo(name: HTMLInputElement, category: HTMLSelectElement) {
21       this.toDoAdded.emit({
22         name: (<HTMLInputElement>name).value,
23         category: (<HTMLSelectElement>category).value
24       });
25   }
26 }
```

we removed class variables “name” and “category” and updated onAddToDo function on line 20. Worried how to identify parameter type as HTMLInputElement and HTMLSelectElement? Just go back to Event Binding section again to revise how can we identify them, in case we do not remember them.

Code till here is committed in branch [v0.1.4](#)

5.5.1 Local reference in Type Script file

We saw how to use Local Reference in HTML file. We can also use Local Reference in Type Script. Let's check the example.

app/to-do/create/create.component.ts

```
1 import {Component, ElementRef, EventEmitter, OnInit, Output, ViewChild}
   from '@angular/core';
2
3 @Component({
4   selector: 'app-create',
5   templateUrl: './create.component.html',
6   styleUrls: ['./create.component.css']
7 })
8 export class CreateComponent implements OnInit {
9   @ViewChild('name') nameInTS: ElementRef;
10  @ViewChild('category') categoryInTS: ElementRef;
11
12  @Output() todoAdded = new EventEmitter<{
13    name: string,
14    category: string
15  }>();
16
17  constructor() { }
18
19  ngOnInit() {
20  }
21
22  onAddToDo() {
23    this.todoAdded.emit({
24      name: (<HTMLInputElement>this.nameInTS.nativeElement).value,
25      category: (<HTMLInputElement>this.categoryInTS.nativeElement).
        value
26    });
27  }
28 }
```

We first need to create two variables to hold the value of local reference. We did this in line 9 and 10. The syntax starts with `@ViewChild('nameOfLocalReferenceInHTMLFile')`. It tells angular we want to assign local reference to current variable of TS file. After that we declare variable. These variables are always are of type “ElementRef”, however, you can confirm or identify it with trick we discussed during event binding.

Since we are using ViewChild and ElementRef, we obviously imported them in line 1.

In our onAddToDo function, we no longer needs parameters so we removed them from here and in HTML file as well. However, with this, we needed to change the way of assigning value.

We moved away from two way binding to local reference to remove local variable and now we added

them again. However, now they are only one way binding (HTML->TS) and we can not change them in HTML accidentally. Thus, angular provide us different tools to be used in different situations, which tool to use depends on actual project condition.

Code till here is committed in branch [v0.1.5](#)

6 Directives

Directives are the most basic building block of an Angular application.

Oops! Didn't we said same about components? Yes, we were right and we are right. Component is just an example of directive. Component is a directive with template.

Directives are used to extend the power of HTML attributes and shape(reshape) DOM's structure.

There are three type of directives:

- Component directive (or simply component) that we already discussed in last chapter.
- Attribute directives
- Structural directive

Let's first see some of the directives provided by angular.

6.1 Attribute directives

Attribute directives, as name suggest, looks like normal HTML attribute. They only change (not add/remove) elements they are on but change their appearance or behavior.

6.2 Structural directives.

Structural directives are responsible for HTML layout. They shape or reshape the DOM's structure, typically by adding, removing, or manipulating elements. They are easy to recognize as asterisk (*) precedes the directive attribute name. For example `*ngIf`, `*ngFor`.

6.3 Attribute directive example

Let's see some attribute directives provided by Angular.

6.3.1 ngClass

It can apply CSS classes to a HTML element. Example:

app/to-do/list/list.component.css

```
1 .bold {  
2   font-weight: bold;  
3 }
```

app/to-do/list/list.component.html

```
16 <tr *ngFor="let todo of todos;">  
17   <td><input type="checkbox" class="form-control" [checked]="todo  
    .done"></td>  
18   <td [ngClass]="todo.done ? '' : 'bold'">{{ todo.name }}</td>  
19   <td>{{ todo.category }}</td>  
20   <td>  
21     <button class="btn btn-info mr-2">Edit</button>  
22     <button class="btn btn-danger">Delete</button>  
23   </td>  
24 </tr>
```

As you may check in line 18, we used `ngClass="bold"` and defined “bold” class in CSS, only for Todos which are not done. Now our incomplete Todos are shown in bold text.

`ngClass` also support arrays `[ngClass]="['bold', 'red']"` and objects `[ngClass]="{bold: true, red: false}"` syntax, which makes it easy to control through Component class.

Code till here is committed in branch `v6.3.1`

6.3.2 ngStyle

Like `ngClass`, we can also use `ngStyle` to write styles without using CSS (If your company policy allows ;)

```
1 @Component({  
2   selector: 'app-style-example',  
3   template:  
4     <p  
5       [ngStyle]="{  
6         'color': 'red',  
7         'font-weight': 'bold',  
8         'borderBottom': borderBottomStyle
```

```
9         }">
10         <ng-content></ng-content>
11     </p>
12
13 })
14 export class StyleExampleComponent {
15     borderStyle = '1px solid red';
16 }
```

6.4 Structure directive example

Let's first check some structural directives provided by Angular.

6.4.1 ngFor

We actually already seen the example of ngFor.

app/to-do/list/list.component.html

```
16     <tr *ngFor="let toDo of toDos;">
17         <td><input type="checkbox" class="form-control" [checked]="toDo
18             .done"></td>
19         <td [ngClass]="toDo.done ? 'bold' : 'normal'">{{ toDo.name }}</td>
20         <td>{{ toDo.category }}</td>
21         <td>
22             <button class="btn btn-info mr-2">Edit</button>
23             <button class="btn btn-danger">Delete</button>
24         </td>
25     </tr>
```

If you working in any programming language, it must be very easy for you to identify what line 16 is doing. It is repeating whole repeating “tr” tag. Like “foreach” loop in many languages, it create local variable “toDo”, which is single element of “toDos” array. This loop is repeated for every single element of “toDos”.

Now we understand what is “ngFor”. It add “tr” for every “toDos” element. Note, with structure directive, angular is adding an element in the DOM. **But how Angular do it? Is asterisk (*) there just representing structure directive?**

Well, Angular use `template1` tag to define template and through JS they are added/removed from the DOM. While creating template, it use internal directive without asterisk. Thus, event though asterisk make it easy to identify type of directive, it is actually used by angular to render the templates.

6.4.2 ngIf

As the name suggest, ngIf add the element to the DOM if condition is true. For example, let's assume we want to show delete button in our Todo List only if task in completed. We can do it as follow:

app/to-do/list/list.component.html

```
16     <tr *ngFor="let todo of todos;">
17       <td><input type="checkbox" class="form-control" [checked]="todo
18         .done"></td>
19       <td [ngClass]="todo.done ? 'bold' : 'normal'">{{ todo.name }}</td>
20       <td>{{ todo.category }}</td>
21       <td>
22         <button class="btn btn-info mr-2">Edit</button>
23         <button class="btn btn-danger" *ngIf="todo.done">Delete</
24       </td>
25     </tr>
```

As you may notice in line 22, `*ngIf="todo.done"` is doing the trick. With this, our delete button will be added only if task is completed. Also note, if task is not done, button will not be hidden, it simply won't be present (added).

Code till here is committed in branch `v6.4.2`.

6.4.3 Using else part with ngIf

Since ngIf is directive, we need to define else condition in same directive. This can be done by using "ng-template".

Let's assume, if Todo is not complete, we want to show text "Finish Todo". We can do it as follow

app/to-do/list/list.component.html

```
16     <tr *ngFor="let todo of todos;">
17       <td><input type="checkbox" class="form-control" [checked]="todo.
18         done"></td>
```

¹<https://developer.mozilla.org/en-US/docs/Web/HTML/Element/template>


```
18     <td [ngClass]="todo.done ? '' : 'bold'">{{ todo.name }}</td>
19     <td>{{ todo.category }}</td>
20     <td>
21         <button class="btn btn-info mr-2">Edit</button>
22         <button class="btn btn-danger" *ngIf="todo.done; else
           incompleteToDo">Delete</button>
23         <ng-template #incompleteToDo>
24             <span>Finish ToDo</span>
25         </ng-template>
26     </td>
27 </tr>
```

Code till here is committed in branch **v6.4.3**. However, `ngIf` & `template` added here will be removed in next version as we need delete button for every `ToDo`.

7 Models, Services and Dependency Injection

Directives (mainly Components) are helping us shape initial draft of our application and it is working very well.

However, if you are a developer, who values better code architecture like separations of concerns, SOLID Principle, mainly Single Responsibility Principle, you might have noticed something does not seem right.

The architecture we made is fine for a To Do app but in case we need to design some really big application with multiple modules, things are not as loosely coupled as they should be.

In case you have not noticed, let's find some issue in our current To Do application (v6.4.3).

Problem 1: There is a big "todos" array in TodoComponent. Thinking in terms of responsibilities, TodoComponent is responsible for To Do. In MVC terms, it is TodoController, it needs to control the whole flow but by defining the todos array, it is also taking responsibility of the Model (data layer). Agreed we still didn't learn how to communicate with the server to load data, still, TodoController does not seem the right place for defining todos.

7.1 Models

To solve this problem, let's create a To Do model. Model is a simple data class, which holds the data. To be precise, To Do will hold the data of one To Do. Again, since we can have multiple models in the application, it makes sense to make a separate folder for models. Let's create a folder "app/models".

app/models/ToDo.ts

```
1 export class ToDo {
2   name: string;
3   done: boolean;
4   category: string;
5
6   constructor(name: string, done: boolean, category: string) {
7     this.name = name;
8     this.done = done;
```

```
9     this.category = category;
10  }
11 }
```

Models are representation of data (say one row in DB table) in object form.

ToDo class doesn't need much explanation. It is simple TypeScript class having 3 variables. We are setting these variables through constructor.

We also need to do necessary changes in ToDo Component to use our new model.

app/to-do/to-do.component.ts

```
1  import { Component, OnInit } from '@angular/core';
2
3  import { ToDo } from '../models/ToDo';
4
5  @Component({
6    selector: 'app-to-do',
7    templateUrl: './to-do.component.html',
8    styleUrls: ['./to-do.component.css']
9  })
10 export class ToDoComponent implements OnInit {
11
12   toDos: Array<ToDo> = [];
13
14   onToDoAdded(toDo: {
15     name: string,
16     category: string
17   }) {
18     this.toDos.push({
19       name: toDo.name,
20       done: false,
21       category: toDo.category
22     });
23   }
24
25   constructor() {
26     this.toDos.push(new ToDo('Angular Session 1', true, 'Angular'));
27     this.toDos.push(new ToDo('Angular Session 1 Assignment', true, 'Angular'));
28     this.toDos.push(new ToDo('Angular Session 2', true, 'Angular'));
29     this.toDos.push(new ToDo('Angular Session 2 Assignment', true, 'Angular'));
```

```
30     this.todos.push(new Todo('Angular Session 3', true, 'Angular'));
31     this.todos.push(new Todo('Angular Session 3 Assignment', true, '
    Angular'));
32     this.todos.push(new Todo('Schedule Angular meet up 2', false, '
    PHPReboot'));
33     this.todos.push(new Todo('Update Angular quick notes EBook', false,
    'EBooks'));
34 }
35
36 ngOnInit() { }
37 }
```

Here, in line 12, we removed all the `ToDo` assignment in variable `todos`. Here, after colon(`:`), we are defining that `todos` variable will contain an array of `ToDo` objects. `Array<ToDo>` is TypeScript syntax to define that.

We import `ToDo` model in line 3 and pushing new `ToDo` Model in the constructor Line 26-33.

We need not to make any change in `ToDoComponent` HTML & `ListComponent` TS & HTML file. With this, our application is behaving as earlier, but with some better decoupling and separation of concern.

Code till here is committed in branch `v7.1.0`.

7.2 Services & Dependency Injection

A service is a TypeScript class with a narrow and well-defined purpose. As a general coding best practice, it does something specific (small, single responsibility) but does it very well.

Main purpose of service is to provide reusable code that we can reuse in multiple components (DRY Principle). Angular distinguishes components from services to increase modularity and reusability. By separating a component's view-related functionality from other kinds of processing, you can make your component classes lean and efficient.

We just saw Models and like services, it is also a TypeScript class, so what is the difference between Service & Model or any other TS class? Major difference is dependency injection.

We already saw `ToDo` model class. With this, we are making new object of `ToDo` model in the constructor of our `ToDoComponent`. This is called a tight coupling, now our `ToDoComponent` is tightly coupled with `ToDo` model class. This also makes it difficult to test, which we will be discussing later in a chapter dedicated to testing. However, we need an array of `ToDo` models, how can we get rid of this tight coupling.

Whenever you use “new” keyword, always think you are making a tight coupling. Is there any other way around?

Dependency Injection

Dependency injection is a programming concept, which makes our code loosely coupled. It says, instead of creating an instance where you need, pass that instance from a central place.

Our components are also the classes that means, someone is creating their instances, who? Well, Angular do that for us. It create instance of our components. It can also create instance of other Type Script classes, provided we tell it how to do that. Services is just that way, Angular can create instance of services in an optimized way and may provide us same instance, where ever we need it.

Let’s understand it with an example.

7.3 ToDoService

Using ToDo Model, we removed big array from our ToDoComponent class. However, now we are creating instance of ToDo models in constructor. A “new” keyword in ToDoComponent raise few questions.

- Components are responsible to correctly show data. Is our ToDoComponent trying to do more that it should?
- ToDoComponent is creating instance of ToDo model, it is really responsible for it?
- Is ToDo component following Single Responsibility Principle?

Unfortunately, answer to all of above questions is “No”. This suggest ToDoComponent needs refactoring. To do that, let’s first create ToDoService, which should hold business logic to manage Todos.

```
1 ng g s services/ToDo --no-spec
```

This will create a service ToDoService in file “app/services/to-do.services.ts”. Now we want to first manage our Todos in ToDoService, let’s update some code.

app/services/to-do.service.ts

```
1 import { Injectable } from '@angular/core';
2
3 import { ToDo } from '../models/ToDo';
4
5 @Injectable({
6   providedIn: 'root'
7 })
```

```
8 export class ToDoService {
9
10   todos: Array<ToDo> = [];
11
12   constructor() {
13     // Once we learn about Http service, we will load this data from
14     // server.
15     this.todos.push(new ToDo('Angular Session 1', true, 'Angular'));
16     this.todos.push(new ToDo('Angular Session 1 Assignment', true, 'Angular'));
17     this.todos.push(new ToDo('Angular Session 2', true, 'Angular'));
18     this.todos.push(new ToDo('Angular Session 2 Assignment', true, 'Angular'));
19     this.todos.push(new ToDo('Angular Session 3', true, 'Angular'));
20     this.todos.push(new ToDo('Angular Session 3 Assignment', true, 'Angular'));
21     this.todos.push(new ToDo('Schedule Angular meet up 2', false, 'PHPReboot'));
22     this.todos.push(new ToDo('Update Angular quick notes EBook', false, 'EBooks'));
23   }
24
25   addToDo(name: string, category: string) {
26     this.todos.push(new ToDo(name, false, category));
27   }
28 }
```

Injectable (line 1 and 5-7) is the default code generated by CLI, we will learn about it shortly.

Line 3 is importing ToDo model, which we obviously need to hold Todos. Most of the other code here is taken from ToDoComponent. Line 10 is simply defining a class variable to hold an array of ToDo models. Constructor (Line 12-22) is exactly copied from ToDoComponent. “addToDo” method (Line 24-26) is basically the copy of “onToDoAdded” method of ToDoComponent but we simplified the parameters.

Now we have the required service, let’s update ToDoComponent to use it.

app/to-do/to-do.component.ts

```
1 import { Component, OnInit } from '@angular/core';
2
3 import { ToDo } from '../models/ToDo';
4 import { ToDoService } from '../services/to-do.service';
5
6 @Component({
```

```
7   selector: 'app-to-do',
8   templateUrl: './to-do.component.html',
9   styleUrls: ['./to-do.component.css']
10 })
11 export class ToDoComponent implements OnInit {
12
13   todos: Array<ToDo> = [];
14
15   onToDoAdded(toDo: {
16     name: string,
17     category: string
18   }) {
19     this.todoService.addToDo(toDo.name, toDo.category);
20   }
21
22   constructor(private todoService: ToDoService) {
23     this.todos = this.todoService.todos;
24   }
25
26   ngOnInit() { }
27 }
```

Now our component class is getting much simpler. Most important change is our constructor. You could notice **private** `todoService: ToDoService` as parameter. This is Type Script's short cut to define private class variable. Example

```
1 export class PrivateClassVariableExampel {
2   private classVariable: SomeClass;
3
4   constructor(classVariable: SomeClass) {
5     this.classVariable = classVariable;
6   }
7 }
```

This code is exactly equivalent to

```
1 export class PrivateClassVariableExampel {
2   constructor(private classVariable: SomeClass) {}
3 }
```


Type Script (and ES6) have many of such shortcuts, which we will see when needed. Once this book is complete, I'll be adding a separate chapters on ES6 and Type Script. This could be helpful for someone not familiar with ES6 or Type Script.

We still need to manage `todos` variable in `ToDoComponent` as we need to pass it to `ListComponent`. Thus, in line 23, we are making it as a copy of `todos` variable of `ToDoService`. However, by this, we are not using services effectively and we will soon get rid of it.

Another important thing to note in constructor, we said we need a parameter of type `ToDoService`, we never created an instance of our service. This is Dependency Injection. We will learn more about it once we go through all the changes.

Our “`onToDoAdded`” method (line 15) is also simplified. Now it is just calling “`addToDo`” method of `ToDoService`.

With these changes, our application is working exactly as it was working earlier but we are now using `ToDoService`.

7.3.1 How Dependency Injection is working?

In last example, we never created an object of `ToDoService`, yet it is available as constructor's parameter. Obviously someone or something is creating these objects. It is Angular, which is doing this job for us. However, we need to tell angular the strategy to create objects. In above example, Angular will create only one instance of `ToDoService` and pass same instance anywhere we was through Dependency Injection (parameter type). Thus, our `ToDoService` is virtually acting as Singleton class with only one instance (Virtually because you can create new instance with “`new`” keyword). We actually told Angular to do this through `providedIn: 'root'` passed in parameter object of `Injectable` decorator.

There could be cases where we wish to create multiple instance of a service. We can tell this to angular.

7.3.2 Hierarchical Injector

Dependency Injection in angular needs to things:

- Provider: tells angular how and when to create a new instance and where to pass existing one.
- Type-hinting: tells angular which class's object is needed.

Here, providers are important to control hierarchical injector behavior. Angular, by default, do not create instance of service during injecting. It just pass existing instance from current or parent component. Thus, object created in one component, will be available in all child components.

7.3.3 How to get new instance?

Well, its good to have single instance of service in most of the cases as we will soon learn how to use services as data services. It also make sense as service have reusable code, therefore making different instance of a service doesn't make much sense. However, if for some requirements, we need different instance of same service in different components, it can be done through providers.

In our example, we passed `providedIn: 'root'` in Injectable decorator. It means, object will be created at root of application and all child components (all components are children of root) should get same instance. In other words, with `providedIn: 'root'` we said we need same instance of the class in whole application.

Angular applications are divided in modules. We will learn more about modules later but for now, every application have one or more module. Our ToDoApplication have one module AppModule that we are configuring in "app/app.module.ts" file. Every module is a collection of multiple components. These components are also hierarchical. In our application, till now, AppComponent loads ToDoComponent and therefore, AppComponent is parent component of ToDoComponent or ToDoComponent is child component of AppComponent. Similarly, ToDoComponent is parent component of CreateComponent and ListComponent.

If we want a service to be available only for the components of a particular module, we will not add `providedIn: 'root'` in Injectable decorator of service but we will add ToDoService in provider array of app.module.ts.

app/services/to-do.service.ts

```
5  @Injectable()
6  export class ToDoService {
```

app/app.module.ts

```
1  import { BrowserModule } from '@angular/platform-browser';
2  import { NgModule } from '@angular/core';
3  import { FormsModule } from '@angular/forms';
4
5  import { AppComponent } from './app.component';
6  import { HeaderComponent } from './common/header/header.component';
7  import { ToDoComponent } from './to-do/to-do.component';
8  import { ListComponent } from './to-do/list/list.component';
9  import { CreateComponent } from './to-do/create/create.component';
10 import { ToDoService } from './services/to-do.service';
11
12 @NgModule({
```

```
13   declarations: [  
14     AppComponent,  
15     HeaderComponent,  
16     TodoComponent,  
17     ListComponent,  
18     CreateComponent  
19   ],  
20   imports: [  
21     BrowserModule,  
22     FormsModule  
23   ],  
24   providers: [  
25     ToDoService  
26   ],  
27   bootstrap: [AppComponent]  
28 })  
29 export class AppModule { }
```

Here, we removed `providedIn: 'root'` from service's injectable decorator but added in provider's array of AppModule's NgModule decorator. Now `ToDoService` is available only in app module but since our whole application is designed in single module, it will not make any difference.

Now let's assume we want our `ToDoService` only in `ToDoComponent` and its child components. To do this, let's revert changes in "app.module.ts" file and add providers in `ToDoComponent`. We can do it in providers array in component decorator.

app/to-do/to-do.component.ts

```
1  import { Component, OnInit } from '@angular/core';  
2  
3  import { ToDo } from '../models/ToDo';  
4  import { ToDoService } from '../services/to-do.service';  
5  
6  @Component({  
7    selector: 'app-to-do',  
8    templateUrl: './to-do.component.html',  
9    styleUrls: ['./to-do.component.css'],  
10   providers: [ToDoService]  
11 })  
12 export class ToDoComponent implements OnInit {  
13  
14   toDos: Array<ToDo> = [];  
15
```

```
16   onToDoAdded(toDo: {
17     name: string,
18     category: string
19   }) {
20     this.todoService.addToDo(toDo.name, toDo.category);
21   }
22
23   constructor(private todoService: ToDoService) {
24     this.todos = this.todoService.todos;
25   }
26
27   ngOnInit() { }
28 }
```

As we can see in line 10, we added provider's array in Component decorator.

A provider simply tell Angular to create a new instance of service for this Component and its child components. Actual instance will be passed through dependency injection (type hinting). As we will see in some time, we also need ToDoService in ListComponent. For this, we will make similar constructor in ListComponent. However, if we do not add provider in ListComponent, new instance of ToDoService will not be created. Angular will check it's parent component (ToDoComponent) for the instance of ToDoService and pass the one created for ToDoComponent.

Code till here is committed in branch v7.3.3

7.4 Data service

Service can also be used as data service. Why it is needed? In our list component, we need to show todos. For this, we are using custom property binding to get todos variable from parent component (ToDoComponent). This is fine for now as we need to pass data from immediate parent but in case there is a long hierarchy or we need to share data between two sibling components (Components having same parent), there will be a long chain of @Input() and @Output(). Thus, we can also use services to share data between two components distantly related. Let's see it with an example.

First, since our ToDoService holds the list of todos, we need not to pass that data from "to-do.component.html" to "list.component.ts". Let's first remove this.

app/to-do/to-do.component.html

```
1 <app-create (todoAdded)="onToDoAdded($event)">Add new ToDo</app-create>
2 <app-list>ToDo List</app-list>
```

We removed custom property binding from line 2. Since we are no having custom property binding, toDos variable inListComponent may also be removed.

app/to-do/list/list.component.ts

```
1 import {Component, Input, OnInit} from '@angular/core';
2 import {ToDoService} from '../services/to-do.service';
3
4 @Component({
5   selector: 'app-list',
6   templateUrl: './list.component.html',
7   styleUrls: ['./list.component.css']
8 })
9 export class ListComponent implements OnInit {
10
11   constructor(public toDoService: ToDoService) {
12   }
13
14   ngOnInit() {
15   }
16 }
```

We removed variable “toDos” but we still need to do list from toDoService. Thus, our constructor now have ToDoService injected. Notice **public**, it will make the variable as class variable, which can be used in HTML.

Last, we need to use toDoService variable instead of toDos in HTML

app/to-do/list/list.component.html

```
1 <div class="row">
2   <div class="col">
3     <h2>
4       <ng-content></ng-content>
5     </h2>
6     <table class="table">
7       <thead>
8         <tr>
9           <th scope="col">#</th>
10          <th scope="col">ToDo</th>
11          <th scope="col">Category</th>
12          <th scope="col">Actions</th>
13        </tr>
14      </thead>
```

```
15     <tbody>
16     <tr *ngFor="let todo of todoService.todos;">
17         <td><input type="checkbox" class="form-control" [checked]="todo
18             .done"></td>
19         <td [ngClass]="todo.done ? 'bold' : ''">{{ todo.name }}</td>
20         <td>{{ todo.category }}</td>
21         <td>
22             <button class="btn btn-info mr-2">Edit</button>
23             <button class="btn btn-danger">Delete</button>
24         </td>
25     </tr>
26 </tbody>
27 </table>
28 </div>
```

We just change “todos” to “todoService.todos” in line 16.

Last but probably most important, we want to tell angular to create new instance of ToDoService for ToDoComponent and all its child components. We also do not need “todos” class variable any more.

app/to-do/to-do.component.ts

```
1  import { Component, OnInit } from '@angular/core';
2
3  import { Todo } from '../models/Todo';
4  import { ToDoService } from '../services/to-do.service';
5
6  @Component({
7      selector: 'app-to-do',
8      templateUrl: './to-do.component.html',
9      styleUrls: ['./to-do.component.css'],
10     providers: [ToDoService]
11 })
12 export class ToDoComponent implements OnInit {
13
14     onToDoAdded(todo: {
15         name: string,
16         category: string
17     }) {
18         this.todoService.addToDo(todo.name, todo.category);
19     }
20
21     constructor(private todoService: ToDoService) { }
```

```
22
23   ngOnInit() { }
24 }
```

With these changes, our application is still working as earlier but now it is little light weighted as we are using ToDoService as Data Service and get rid of custom property binding.

Code till here is committed in branch v7.4.0

7.4.1 Data Service for custom Event Binding.

Like custom property binding, we can also use Data Service for custom event binding. In our code, CreateComponent is emitting new event on click of button. Let's first update it.

app/to-do/create/create.component.ts

```
1  import {Component, ElementRef, EventEmitter, OnInit, Output, ViewChild}
    from '@angular/core';
2  import {ToDoService} from '../services/to-do.service';
3
4  @Component({
5    selector: 'app-create',
6    templateUrl: './create.component.html',
7    styleUrls: ['./create.component.css']
8  })
9  export class CreateComponent implements OnInit {
10    @ViewChild('name') nameInTS: ElementRef;
11    @ViewChild('category') categoryInTS: ElementRef;
12
13    // @Output() toDoAdded = new EventEmitter<{
14    //   name: string,
15    //   category: string
16    // }>();
17    //
18    // constructor() { }
19    constructor(private toDoService: ToDoService) { }
20
21    ngOnInit() {
22    }
23
24    onAddToDo() {
25      this.toDoService.addToDo(
26        (<HTMLInputElement>this.nameInTS.nativeElement).value,
```

```

27     (<HTMLInputElement>this.categoryInTS.nativeElement).value
28   );
29   // this.todoAdded.emit({
30   //   name: (<HTMLInputElement>this.nameInTS.nativeElement).value,
31   //   category: (<HTMLInputElement>this.categoryInTS.nativeElement).
32   //     value
33   // });
34 }

```

Keeping commented code is not good. We kept it there just so that we can easily understand changes. Uncommented code will be removed in next version.

First change, we no longer need “todoAdded” event emitter on line 13, so it is deleted (commented). However, we need ToDoService, we asked Angular to inject it in the constructor (line 19).

Further, onAddToDo method now do not need to emit event (Line 28-32) but it need to call addToDo method of ToDoService (Line 25-28).

Since CreateComponent is no longer emitting event, it can be removed in HTML as well

app/to-do/to-do.component.html

```

1 <app-create>Add new ToDo</app-create>
2 <app-list>ToDo List</app-list>

```

With this, we now do not need “onToDoAdded” method. Actually there is no code left in our ToDoComponent.

app/to-do/to-do.component.ts

```

1 import { Component, OnInit } from '@angular/core';
2
3 import { ToDo } from '../models/ToDo';
4 import { ToDoService } from '../services/to-do.service';
5
6 @Component({
7   selector: 'app-to-do',
8   templateUrl: './to-do.component.html',
9   styleUrls: ['./to-do.component.css'],
10  providers: [ToDoService]
11 })
12 export class ToDoComponent implements OnInit {
13
14   // onToDoAdded(toDo: {

```



```
15    //    name: string,  
16    //    category: string  
17    // }) {  
18    //    this.todoService.addToDo(todo.name, todo.category);  
19    // }  
20  
21    // constructor(private todoService: TodoService) { }  
22    constructor() {}  
23  
24    ngOnInit() { }  
25 }
```

Still, `ToDoComponent` is doing one very important thing, it has a `providers` array. It is instructing Angular to create one single instance of `ToDoService` for this (`ToDoComponent`) and all child components. This will make sure both `CreateComponent` and `ListComponent` are working on a single instance of `ToDoService`.

Code till here is committed in branch `v7.4.1`