Introduction to Angular

Dec 2018

Ted Hagos

[tedhagos@gmail.com](mailto:tedhagos@gmail.com)

[@lovescaffeine](https://twitter.com/@lovescaffeine)

<https://dev.to/tedhagos>

1 Setup

We’ll need the following;

* **npm** (Node Package Manager). You need to install NodeJS in your system in order to get npm. You can download the pre-compiled binary installer from <https://nodejs.org>. You can also install NodeJS via nvm (NodeJS Virtual Manager), and there are some advantages to doing this, but the simplest is to download and use the binary
* **Visual Studio Code**. This is optional, you can (and should use) the editor which you are most comfortable with, but we’ll use VS Code here because it has a good eco-system for NodeJS, Angular and TypeScript projects
* **TypeScript**. We’ll install this using npm
* **Angular CLI**. We’ll also install this using npm

TODO include instructions to install nvm in Linux and macOS. Include curl and wget

TODO include instructions to install node (nvm?) in Windows

To install Visual Studio Code, get the pre-compiled binary installer from <https://code.visualstudio.com/download>. Get the the installer appropriate for your platform and install it in the same way you would install any other software in your OS.

**Launch Visual Studio**

* macOS – Launch a “Finder” window, go to the “Applications” folder (in left-hand sidebar), double click “Visual Studio Code”
* Windows – You can launch the IDE from the “Start” menu
* Linux Ubuntu/Mint – Click “Menu” type “Visual Studio Code”

You can launch VS Code from the command line.

1. Launch VS Code
2. Open the command palette CTRL + Shift + P (it’s cmd + Shift + P on macOS)
3. Type “shell”. Then, install “Shell command: install code command in PATH”

Apart from Visual Studio code, we’ll also need to install the TypeScript compiler and some other nice tools; you can install these tools from the command line, as shown in the following example;

npm install -g typescript 1

npm install chalk 2

npm install @types/node 3

|  |  |
| --- | --- |
| [1](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/typescript_tools.html#CO1-1) | installs the TypeScript compiler globally, so we can use tsc tool from from anywhere in the command line |
| [2](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/typescript_tools.html#CO1-2) | This is optional, we just want to color our console logs |
| [3](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/typescript_tools.html#CO1-3) | We’re going to run our examples in a node environment, so, it’s best if Typescript recognizes some Node.js syntax; You don’t need if you will run the code on a browser, you’ll only need this if the runtime environment is NodeJS |

1 JavaScript Review

Basic JS types are;

* Number (float, double and integers)
* String
* Boolean
* null
* undefined
* Objects (Arrays are objects)

It’s dynamically typed, so you can write things like this

var a = 10;

var b = “Hello”;

var c = `The quick brown

fox jumped over the head of the

lazy dog`;

JavaScript infers that the variable a is of a Number type because 10 is a number literal, and variable b is of String type because characters enclosed in double quotes (and single quotes). The last example defines variable c also as a String, those are not single quotes, they’re backticks. It’s a new addition to ES2015 which allows you to define Strings that can span multiple lines.

JS is also loosely or weakly typed, hence, you can do things like this

var foo = function() {

return "Hello";

}

console.log(foo);

console.log(typeof(foo));

foo = "World";

console.log(foo())

console.log(typeof(foo));

JS is object oriented, so quite naturally, it has objects. Objects look like this in JS;

var cat = { name: "Fluffy", color: "White"}

console.log(typeof(cat)); // object

Everything inside the pair of curly braces constitutes the definition of an object. This syntax is called JSON (JavaScript Object Notation).

You can also create objects in JS like this;

var obj = function() {

};

display(typeof(new obj())); // object

display(typeof(obj())); // function

Because of JS’s dynamic nature, you can also do these things

var cat = { name: "Fluffy", color: "White"}

console.log(typeof(cat)); // object

cat.age = 3;

console.log(`${cat.name} is already ${cat.age} years old`)

display(typeof(obj())); // function

This is probably one of the things that trips up new JS programmer’s who are coming from a different OOP background. If you’re coming from Java, C# or C++, your OOP inclinations are trying to find what is the equivalent of the “class” keyword in JS. There is none. JS doesn’t have class-based inheritance, what is has is “prototypal” inheritance. Which means, you basically inherit from other objects.

With the ES2015 came some new syntactic sugar of JS; now we can use the “class” keyword, like so

class Cat {

age = 3;

name = "Fluffy";

say() {

console.log("Meoww");

}

}

const cat = new Cat();

console.log(`${cat.name} is already ${cat.age} years old`);

Now this looks a bit more like class-based OO language, but don’t be fooled, it still uses prototypal inheritance.

An object in JS looks like this;

'use strict';

var cat = { name: "Fluffy", color: "White"}

cat.age = 3

cat.speak = function() {

display("Meooow")

}

display(cat.name)

display(cat.age)

cat.speak()

function display(message) {

// var out = document.getElementById('output');

// var temp = out.innerHTML;

// temp = `${temp} <br/> ${message}`

// out.innerHTML = temp

console.log(message)

}

# Hoisting

Listing 1-1. Variable hoisting

number = 0x10;

var mistypeVariable = 10;

console.log(`var number = ${number}`); // 16

console.log(`var mistypeVariable = ${mistypedVariable}`); // 10

This is one of the more dangerous behaviors of the JS runtime, it allows undeclared variables to exists (and be used) through a mechanism called *variable hoisting*.

Listing 1-2. Using strict [[1]](#footnote-1)mode

‘use strict’

number = 0x10;

var mistypeVariable = 10;

console.log(`var number = ${number}`); // will throw error

console.log(`var mistypeVariable = ${mistypeVariable}`); // will throw error

To remedy this, we can use strict mode in the code. The preferred way of declaring and using variables is as follows;

# Scope level variables

The *var* keyword declares and defines a variable globally or locally to an entire function regardless of block scope, as shown in listing 1-3.

Listing 1-3. Using the var keyword

function foo() {

var x = 1;

if (true) {

var x = 2; // same variable!

console.log(x); // 2

}

console.log(x); // 2

}

foo()

Now, let’s see the effect when we use the *let* keyword, instead of var. See Listing 1-4.

Listing 1-4. Effects of the let keyword

function foo() {

let x = 1;

if (true) {

let x = 2; // different variable!

console.log(x); // 2

}

console.log(x); // 1

}

foo()

## Exercises

1. Create a function that will reverse an input String
2. Create a function that will check for palindromes
3. Find the sum of all even numbers (and odd numbers) under 1000

# Further References

If you need some refresher in JS or you’d like to dive a bit deeper, here are some suggestions for reading materials.

* Definitive JavaScript, by David Flanagan <https://amzn.to/definitivejavascript> - A book by David Flanagan, this was one of the earliest (most complete) reference book on JS
* You don’t know JS (series) by Kyle Simpson <https://bit.ly/jskylesimpson> (some are free)
* JavaScript Design Patterns by Adi Osmani <https://bit.ly/jsdesignpatterns_aosmani>
* Programming JavaScript Applications by Eric Elliot <https://amzn.to/2SJKLea>

2 A bit of TypeScript

Variable

Functions

Classes

Decorators

Modules

Is it possible to write Angular apps without using TypeScript? Yes, it is (you can read about it here <http://bit.ly/angularwithouttypescript>). But there are some downsides to that, namely;

* Typescript simplifies (and hides) a lot of the complexity tha goes into making an Angular component e.g. TypeScript’s `class` keyword is actually just doing an lot of IIFEs to prevent scope pollution (it’s not all the `class` does, but you can certainly see why it’s simpler to write the class keyword instead of an IIFE)
* Angular is already big, anything that makes it simpler and more concise is welcome
* Most of the literature about using Angular is with TypeScript (not Vanilla JS or any other thing that transpiles to Vanilla JS). The books about Angular apps features TypeScript, the tutorials on YouTube, Udemy, Pluralsight etc., features TypeScript as the language, so you’d be better off using the same language that most people in the community uses
* Angular itself, was written using TypeScript. There are plenty more reasons, but these 4 should be enough to choose TypeScript

# What’s the big deal with TypeScript

Dynamic typing

This

Fat arrow

ES2015

Listing 2-1. Hello TypeScript

var mistypeVariable = 10;

class Start {

name: string;

constructor(message: string) {

this.name = message

}

public main() {

console.log(`Hello ${this.name}`);

}

}

new Start("World").main()

console.log(mistypeVariable)

Listing 2-2. Class basics

class Person {

private \_lastname:string;

private \_firstname:string;

private \_email:string;

constructor(lname:string, fname:string, email:string) {

this.\_lastname = lname;

this.\_firstname = fname;

this.\_email = email;

}

get fullName():string {

return `${this.\_firstname} ${this.\_lastname}`;

}

set lastname(newname:string) {

console.log(`changing ${this.\_lastname} to ${newname}`)

this.\_lastname = newname

}

set firstname(newname:string) {

console.log(`changing ${this.\_firstname} to ${newname}`)

this.\_firstname = newname

}

}

var john = new Person('Doe','John','jdoe@gmail.com')

console.log(`full name: ${john.fullName}`)

john.lastname = "Grisham";

john.firstname = "Jane";

console.log(`full name: ${john.fullName}`)

|  |  |
| --- | --- |
| **TypeScript compiler command** | **Description** |
| tsc --init | Creates a tsconfig.json in the current directory |
| tsc --project . --watch | Compiles all typescript files found in the current directory. It performs an auto-compilation whenever a typescript source file changes in the current dir |
| tsc --target es2015 <file.ts> --watch | Compiles one specific typescript source file and targets the es2015 (ES6) JS version. It watches any changes to the specified typescript source file |
| tsc <file.ts> | Compiles one specific typescript source file. If you use the tsc compiler like this (specifying a source file to compile), the compiler will ignore tsconfig.json |

More information <https://www.typescriptlang.org/docs/handbook/tsconfig-json.html>

3 Introduction to Angular

# What you need to know;

* JavaScript
* HTML
* CSS

What’s “nice to have”;

* Prior background in building web apps with other technologies e.g. PHP, Java, ASP.net
* Knowledge of OOP via Java, C#, C++, PHP etc

Some of questions that can get us started are;

* What’s an Angular component
* Where do we put the HTML for the UI
* When should we use data-binding?
* Why do we need a service
* How do we build a component

Angular is;

* A JS framework (some even say, it’s a platform)
* It’s used for building client-side applications (resides on the browser)
* It uses HTML, CSS and JavaScript

There are other JS frameworks, why Angular?

Why the move from AngularJS to Angular?

**Angular versions**

Angular versions 1.0 to 1.7 is known as AngularJS

Angular versions 2.x to the current version is known simply as Angular. The two code-bases are incompatible. There is no upgrade path from AngularJS to Angular.

# Anatomy of an Angular app

An app is made up of components and services that provides functionalities across those components.



Fig 1-1. Angular App

An Angular component is made up of templates, classes and meta-data.



Fig 1-2. Angular component

* **Template** – this is HTML part, the UI part. It’s the stuff that users can see. It’s an HTML fragment
* **Class** – This is the code associated with the view; properties are data elements that you may use for the UI. The methods are what you will use if you want to react to some events e.g. when a user clicks a button
* **Meta-data** – provides additional information about the component. It tells the Angular framework that the class (this class, where the meta-data is co-located) should be treated as an Angular component

In order for all the components to come together, we pull glue them together using Angular modules.



Fig 1-3. Angular modules

An application needs at least, one root angular module. It can have more modules, but it needs to have at least one.

Starting an Angular app, isn’t like using a library like JQuery or React where you “import” or include the JS library in the HTML using the <script> tag — this was how you build AngularJS before, but not anymore.

4 Tools and Project Setup

Language - typescript

Editor – Use whatever you’re comfortable with, but we’ll use Visual Studio Code

Environment -

A basic Angular app

Installing an app

Running an app

About modules

# Language and editor

Angular uses JS, so, you can use a variety of languages e.g. CoffeeScript, TypeScript, VanillaJS, Dart, Kotlin or any other languages that transpiles to JS. In this course, we’ll use TypeScript — why? Because that’s what many Angular programmers use. It’s also used in a quite a lot of literature about Angular.

For the editor, you can use whatever you’re comfortable with; but I’ll use Visual Studio Code.

# Environment

You’ll need npm.

To setup a new Angular app, we’ll need to do the following;

1. Create an application folder
2. Add package definition and configuration files
3. Install the packages
4. Create the app’s Angular module
5. Create the *main.ts* file
6. Create the host Web Page (index.html)

* Manually perform these steps – not a good idea, but it can be done
* Use the CLI – this is the recommended tool
* Starter files -

Example 2-1. Quick start with the CLI

npm install -g @angular/cli

ng new hello-world

npm install

# Modules

* TypeScript modules
* ES2015 modules
* Angular modules

5 Components

Components are the building blocks of Angular apps. An app is essentially just a collection of components that are arranged either side-by-side or nested in one another. A component has 3 major parts, 1) a class 2) metadata and 3) a template. Think of a component as as something that contains application logic and that it controls a region of the user interface using HTML *templates*.



Fig 5-1. A closer look at components

To continue our investigation of components, let’s look at how components are structured and their make-up. Create a new Angular app, as shown in example 5-1; don’t include Angular routing (so, press “N”) and choose “CSS” (press enter while CSS is selected).

Example 5-1. Create a new application

$ ng new ch5person

? Would you like to add Angular routing? No

? Which stylesheet format would you like to use

> CSS

SCSS

SASS

LESS

STYLUS

The CLI tool will generate and scaffold a basic Angular app. Change directory to “ch5person” and open it in VS Code, as shown in Example 5-2.

Example 5-2. Open the project in VS Code

cd ch5person

code .

The project contains many files, the most important ones (for now) are shown in figure 5-1 below.



Figure 5-1. Important files in an Angular project.

|  |  |
| --- | --- |
| [1](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/delme_2.html#CO1-1) | **index.html**.This is the main html page of the app |
| [2](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/delme_2.html#CO1-2) | **main.ts**. This is the starting point |
| [3](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/delme_2.html#CO1-3) | **app folder**. This is the folder of the default component that the CLI tool generated for us; you can add more. This app folder contains all the files for an Angular component |
| [4](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/delme_2.html#CO1-4) | **app/app.component.ts**. A TypeScript file which contains the definition of our component |
| [5](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/delme_2.html#CO1-5) | **app/app.module.ts**. This is the module file, this is where we tell Angular which Component are we loading first; wev’e got to start somewhere, think of this as something similar to C’s int main() or Java’s public static void main. It’s our entry point |

One of the differences of AngularJS with Angular is the tooling support. The CLI makes Angular more than a framework, it’s almost a platform. If you’ve worked with front-end projects before, you may have used tools like bower, grunt or gulp and the various web servers like http-server; and you may have spent quite a deal of time setting up a build-script for all these. Angular comes with some tooling in the form of Angular-CLI. It’s a bunch of command-line interface (CLI) tools that you can use to create new apps, new components, run tests and even use as a development server.

We’ve already used the CLI to create an app, now, let’s use it to run a dev server. You can run a dev server by using the following command.

ng serve -o

The -o flag means we want to launch a browser and view our app in it. This effectively runs a webpack dev server at port 4200. The browser opens a tab with <http://localhost:4200>. Figure 5-2 shows our app with the default component.

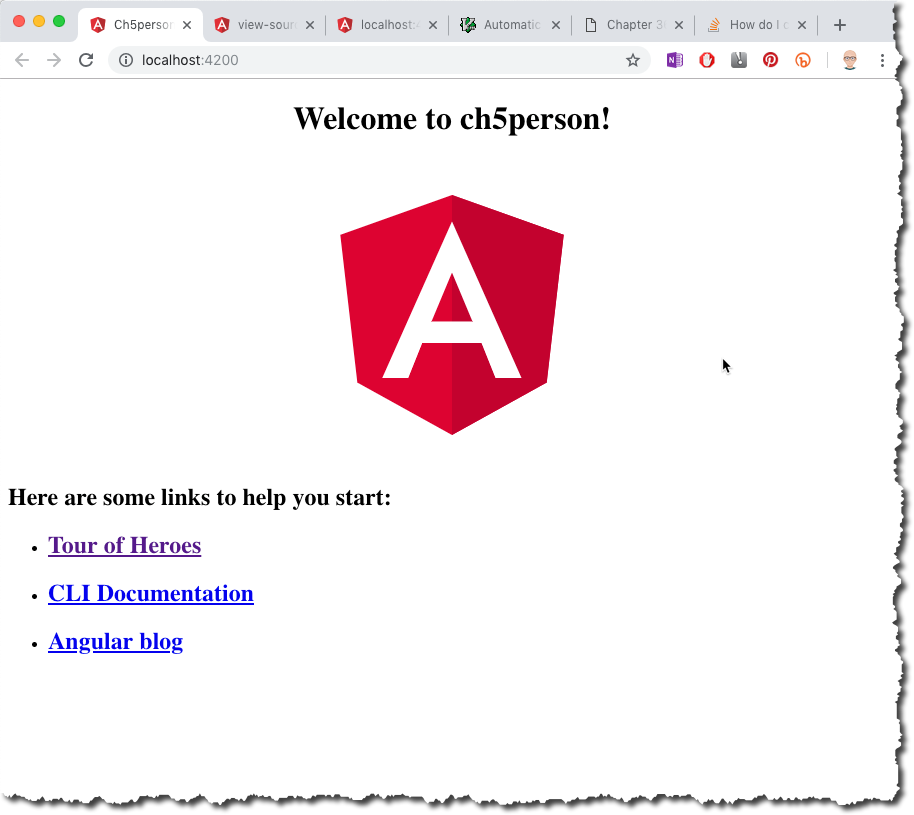


Figure 5-2. Default screen for a newly created app

You may also run the application using npm, like so

npm start

This command will look for the package.json file on the current directory (presumably the project root directory) and look at the *scripts* definition.

Listing 5-1 shows us an excerpt from package.json

Listing 5-1.package.json

{

"name": "ch5person",

"version": "0.0.0",

"scripts": { 1

"ng": "ng",

"start": "ng serve", 2

"build": "ng build",

"test": "ng test",

"lint": "ng lint",

"e2e": "ng e2e"

},

|  |  |
| --- | --- |
| [1](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/package_json.html#CO1-1) | The *scripts* object defines the various arguments we can use with the npm command |
| [2](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/package_json.html#CO1-2) | As you can see, the *start* argument invokes the ng serve command. You might want to edit this line so that it reads ng serve -o; that way, when you invoke npm start, it automatically launches a browser instance |

## Let’s make some changes

An Angular component contains application logic (the class), metadata and the HTML template

Listing 5-2. Modified AppComponent

import { Component } from "@angular/core"; 1

@Component({ 2

selector: 'app-root', 3

template: ` 4

<h1>{{ title }}</h1> 5

`

})

export class AppComponent { 6

title = "Hello World" 7

}

|  |  |
| --- | --- |
| [1](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/appcomponent.html#CO1-1) | We need the Component function from the core Angular libraries, so, let’s import it |
| [2](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/appcomponent.html#CO1-2) | This is the metadata part of our component. We used the @Component decorator to tell Angular that the class *AppComponent* (see 6 below) is actually a Component. The decorator takes a single argument (a decorator is a function, remember), it takes a JSON object where we can define some of the important properties of the component |
| [3](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/appcomponent.html#CO1-3) | This line tells Angular which portion of the UI (html) this component is supposed to control; in this case, it’s app-root. You will find this html element inside *index.html* (located in the root folder of the project) |
| [4](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/appcomponent.html#CO1-4) | The template property is where we tell Angular what gets rendered on the HTML page, noticeably, it looks a lot like HTML; and that’s good, because everything we know about HTML tags and elements are valid here. Notice that we’re not using single quotes to fence our template, those are back-ticks. If you’re using an American style keyboard, that key is found to the left of “1” and above the “TAB” key. It’s an ES2015 syntax which allows us to define a string that spans multiple lines |
| [5](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/appcomponent.html#CO1-5) | The {{ }} moustache or handlebars syntax is Angular’s way of binding data to an HTML element, it’s called interpolation syntax or template expression. During runtime, the expression inside the template will be substituted by the value of the property (with the same name) in the class AppComponent. In this way, the AppComponent is acting like a *model* for the app |
| [6](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/appcomponent.html#CO1-6) | This is the class definition. It’s a TypeScript class. We’re exporting it so that it can be used by other (Angular) components |
| [7](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/appcomponent.html#CO1-7) | A property inside the class AppComponent |

Assuming that ng serve is still running, you should see the effects of our changes.

## What happened

The compilation process is as follows;

1. When we kicked off with **ng serve**, the CLI compiled the project (**ng build**)
2. The compiled application (html + js + icons + map files etc) was put in the **dist** folder (located at the root folder of the project, same level as **src**). You should inspect that, but t he resulting files are not fun to read at all.

This is another major difference between AngularJS and Angular. The latter compiles the whole project outside the browser while the former compiles the project within the browser. So, in an AngularJS project, the source files (the ones you write at design time) are the same files during runtime. In an Angular project, the runtime files are not the same as the source files. This is why the map files are included in the *dist* folder, so you have enough debugging info if you want to use the debugging tools of the browser.

So, we’re not going to follow the runtime files, that’s of little interest to us at the moment. What we need to know is how are the Angular artefacts (files) connected architecturally.

At a high level, this is what happens;

1. The Angular compiler looks at main.ts (at the root folder), see Listing 5-3. From there, we know that were supposed to load AppModule, so, let’s go go AppModule
2. Every Angular application needs at least one module. This is where we tell Angular what components (or other modules) we’re using and which component is our entry point. See Listing 5-4. In our example, AppComponent is our entry point
3. You’ve already seen AppComponent (see Listing 5-2 above), and you also already know what it does. That brings us now up to speed.

Listing 5-3. main.ts

import { enableProdMode } from '@angular/core';

import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';

import { environment } from './environments/environment';

if (environment.production) {

enableProdMode();

}

platformBrowserDynamic().bootstrapModule(AppModule) 1

.catch(err => console.error(err));

|  |  |
| --- | --- |
| [1](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO1-1) | Tells the Angular compiler which module to start with. Think of this as the public static void main of the app. We know where to get the AppModule because we imported it from ‘./app/app.module’ |

Listing 5-4. AppModule

import { BrowserModule } from '@angular/platform-browser'; 1

import { NgModule } from '@angular/core'; 2

import { AppComponent } from './app.component'; 3

@NgModule({ 4

declarations: [ 5

AppComponent

],

imports: [

BrowserModule

],

providers: [],

bootstrap: [AppComponent] 6

})

export class AppModule { } 7

|  |  |
| --- | --- |
| [1](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO1-1) | Let’s import the BrowserModule, because our app will run in a Browser. Angular was architected to run on multiple platforms like mobile (using NativeScript) or the desktop (using Electron) |
| [2](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO1-2) | We need the NgModule function from the core libs |
| [3](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO1-3) | We’ll need to reference AppComponent in the metadata section, so, we need to import it |
| [4](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO1-4) | This file is supposed to be a module, so, we need to tell Angular that this is a module. We do that by decorating the class (inside this file) with the @NgModule decorator |
| [5](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO1-5) | This part of metadata simply tells Angular what components are app is using. You can have many components in an app, that’s why it’s specified as an array |
| [6](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO1-6) | Tells Angular which component is the entry point |
| [7](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO1-7) | This is our class definition, we don’t need to do anything in it though |

Chapter 6 Data Binding

Data-binding usually means that a UI element, like an **HTML <input>** element is automatically bound to a data-source, in our case, thet Angular component.

In order to do data-binding in Angular, we need to do the following steps;

1. In the Component (listing 6-1), we need to import the FormsModule from angular/forms. In earlier versions of Angular, the FormsModule was part of angular/core, so, it didn’t need a separate import; but that’s not the case now
2. We also have to make changes to app.module.ts file in order to reflect the importation of the FormsModule
3. In the HTML template (listing 6-2), use either the square bracket syntax or the moustache syntax to do one-way binding
4. In the HTML template (listing 6-2), use the banana-in-a-box syntax — [()] — to do two-way binding. Two-binding means you can make changes on the data either via the UI input element or on the Component, and the change is propagated automatically. The data can be changed in a bi-directional manner

Listing 6-1. src/app/app.component.ts

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

import { FormsModule } from "@angular/forms" 1

@Component({

selector: 'app-root',

templateUrl: './app.component.html', 2

styleUrls: ['./app.component.css'] 3

})

export class AppComponent {

appTitle = "Angular Two way binding"

story = { 4

name: "The sound of drums"

}

changeName() { 5

this.story.name = "Blink"

}

}

|  |  |
| --- | --- |
| [1](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO1-1) | We’re going to use ngModel in our template file, it belongs to the FormsModule, so, we need to import it |
| [2](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO1-2) | We’re not using an inline template anymore. If you want to use an external HTML template, use the templateUrl property instead of just template |
| [3](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO1-3) | We can also specify an external css file. You can use more than one css, hence, the array syntax |
| [4](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO1-4) | We’re defining a story property in the AppComponent class; it’s a JSON |
| [5](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO1-5) | We’re defining a method inside our AppComponent class. We’ll call this function as a response to a click event |

Listing 6-2 shows the annotated HTML for our component.

Listing 6-2. src/app/app.component.html

<div>

<h2>{{ appTitle }}</h2> 1

<div>

Two-way binding

<input [(ngModel)]="story.name"> 2

</div>

<p>{{story.name}}</p>

<div>

One-way binding

<input [value]="story.name"> 3

</div>

<button (click)="changeName()">Change the name</button> 4

</div>

|  |  |
| --- | --- |
| [1](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch_html.html#CO2-1) | You’ve seen this already. The interpolation using the template syntax is a way to bind a UI element (like the innerHTML of an H2 in this case) to class property |
| [2](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch_html.html#CO2-2) | Two way binding means we can update the property of a class dynamically. In this case, we updated the story.name property using a user input from the text field |
| [3](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch_html.html#CO2-3) | Like interpolation, the bracket syntax is another way to do one-way binding |
| [4](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch_html.html#CO2-4) | And this is how to bind a DOM event to class method |

Listing 6-3. src/app.module.ts

import { BrowserModule } from '@angular/platform-browser';

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

import { AppComponent } from './app.component';

import { FormsModule } from '@angular/forms'i; 1

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

FormsModule 2

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

|  |  |
| --- | --- |
| [1](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch_app_module.html#CO3-1) | We have to declare the FormsModule in our app’s root component |
| [2](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch_app_module.html#CO3-2) | We also have to make it a part of the imports section |

directives removed from AngularJS (John Papa, less built in directives)

## Event Binding

Event binding has been simplified in Angular, we no longer have to use (and memorize) directives like ng-click, ng-mouseover, ng-mouse, ng-mousemove, ng-keydown etc. In Angular, we can use the actual names of the DOM event. For example, if we need a button element to respond to a click event, we simply need to bind the DOM click event to the button using the syntax as shown in listing 6-3 (below.

Listing 6-3. Button click

<button (click)="changeName()">

Change name

</button>

Simply surround the name of the event (click, in our case) with a pair of parentheses, and assign a function to it. The function must exist in the Component (shown in Listing 6-4)

Listing 6-4. Somewhere in app.component.ts

@Component({

...

})

export class ThisComponent {

...

changeName() {

console.log("changeName is called");

this.name = this.temp;

}

}

Chapter 7 \*ngIf and \*ngFor

\*ngIf and \*ngFor are structural directives (built-in directives). They’re responsible for HTML layout, they shape or reshape the document’s (DOM) structure by removing or adding HTML elements.

Structual directives are easy to recognize because they are prepended by an asterisk (star sign). For example;

<div \*ngIf ="hero">{{hero.name}}</div>

The \*ngIf directive can be used to hide or show the innerHTML of a DOM element, as shown in the code snippet above. In the expression \*ngIf ="hero", if “hero” is truthy, then the contents of the div will be visible. Presumably, “hero” is property somewhere in the associated component (app.component.ts). If “hero” is falsy, then contents of the div won’t be visible.

The \*ngFor, like the \*ngIf, is another build-in directive (they’re from the BrowserModule, which we’ve imported almost for every project, so, there’s no need for additional imports anymore).

The \*ngFor can be used to iterate through array items in the Component, listings 7-1 and 7-2 shows their usage.

Listing 7-1. app.component.html

<div class="table" \*ngIf='books && books.length'> 1

<table>

<thead>

<tr>

<th>Title</th>

<th>Author</th>

<th>Rating</th>

</tr>

</thead>

<tbody>

<tr \*ngFor='let book of books'> 2

<td>{{ book.title }}</td>

<td>{{ book.author }}</td>

<td>{{ book.rating }}</td>

</tr>

</tbody>

</table>

</div>

|  |  |
| --- | --- |
| [1](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO1-1) | We’re checking if the associated Component has a property named "books" and that it contains something (books.length should be truthy if the array isn’t empty) |
| [2](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO1-2) | books is the name of the array in the Component and 'book' is a programmer defined iterator which we will use to reference the individual elements of the JSON object |

Listing 7-2. app.component.ts

import { Component } from '@angular/core';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'Books';

books: any[] = [ 1

{

title: "Don Quixote",

author: "Cervantes",

rating: 5

},

{

title: "War and Peace",

author: "Leo Tolstoy",

rating: 4.5

},

{

title: "Hound of Baskervilles",

author: "Arthur Conan Doyle",

rating: 4

},

{

title: "Scarlet Letter",

author: "Nathaniel Hawthorne",

rating: 4.5

}

]

}

|  |  |
| --- | --- |
| [1](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO2-1) | 'books' is a property of the Component. It’s an array of JSON objects |

Chapter 8 Pipes

Pipes are used to transform data. The typical usage looks something like this

<div>

{{ author.name | uppercase }}

</div>

In the example above, “**uppercase**” is the pipe-name (a built-in pipe directive) and “**author.name**” is a property found in the associated **app.component.ts**; The pipe character is used to separate the data (author.name) and the pipe-name (uppercase). What’s happening here is that the “author.name” is acting as an input (left hand side of the pipe char) to the pipe-name (uppercase). The pipe is transforming the input and in turn outputs a new value (uppercase version of author.name).

We can also create our very own custom pipes. To create a custom pipe, we need the following;

1. The pipe definition. This is just a decorated TypeScript class where we write the program logic for the pipe
2. Our usual Component
3. Our usual HTML; and
4. Our usual app.module.ts

Listings 8-1, 8-2, 8-3 and 8-4 illustrates how to create and use custom pipes

Listing 8-1. Pipe definition, app.pipes.ts

import { PipeResolver } from '@angular/compiler'; 1

import { Pipe, PipeTransform } from '@angular/core'; 2

@Pipe({ 3

name: 'digitToWords' 4

})

export class AnyNameWouldDo implements PipeTransform { 5

transform(num:number) { 6

let retval:string = "";

switch(num) {

case 0:

retval = "nada";

break;

case 1:

retval = "uno";

break;

case 2:

retval = "dos";

break;

case 3:

retval = "tres";

break;

case 4:

retval = "quatuor";

break;

case 5:

retval = "cinco";

break;

case 6:

retval = "seiz";

break;

default:

retval = "unknown"

}

return retval; 7

}

}

|  |  |
| --- | --- |
| [1](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO1-1) | We need the PipeResolver module, so we have to import it |
| [2](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO1-2) | We also need Pipe and PipeTransform modules |
| [3](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO1-3) | When we wanted to create a Component, we used the @Component decorator. We’ll, we want to create a pipe now, so we use teh @Pipe decorator |
| [4](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO1-4) | The Pipe decorator only needs the name property. In this case, we named it "digitToWords". This is the exact pipe name we will use in the HTML part of our solution |
| [5](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO1-5) | Now we come to the actual class for the Pipe. A Pipe class needs to implement the PipeTransforminterface |
| [6](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO1-6) | The PipeTransform interface only has a single method (which we need to override). We have to override the transform() method of the PipeTransform interface. This is the heart of what makes a pipe work. The transform method takes variable arguments, but in this case we’re only accepting one argument. The argument of the transform method is the input of the pipe. If the usage of the pipe is {{ 4 | digitToWords }}, 4 is the input to transform |
| [7](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO1-7) | The return value of transform (retval) is what will be the output of the pipe expression |

Listing 8-2. app.component.ts, our data source

import { Component} from '@angular/core';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'Pipe Demo';

numbers = [0,1,2,3,4,5,6,7]; 1

}

|  |  |
| --- | --- |
| [1](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO2-1) | I simply need an array to iterate through, so let’s define a simple array of Integers |

Listing 8-3. app.module.ts

import { BrowserModule } from '@angular/platform-browser';

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

import { AppComponent } from './app.component';

import { AnyNameWouldDo } from './app.pipes'; 1

@NgModule({

declarations: [

AppComponent,

AnyNameWouldDo 2

],

imports: [

BrowserModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

|  |  |
| --- | --- |
| [1](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO3-1) | We have to import the definition file for our Pipe |
| [2](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO3-2) | We also have to declare in the declarations section |

Listing 8-4. app.component.html

<h1>{{ title }}</h1>

<ul>

<li \*ngFor="let n of numbers">

{{n}} - {{ n | digitToWords | uppercase }} 1

</li>

</ul>

|  |  |
| --- | --- |
| [1](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO4-1) | n is the input for our pipe. It’s the num parameter to the transform(num) method in our pipe definition (Listing 8-1, bullet 6). |

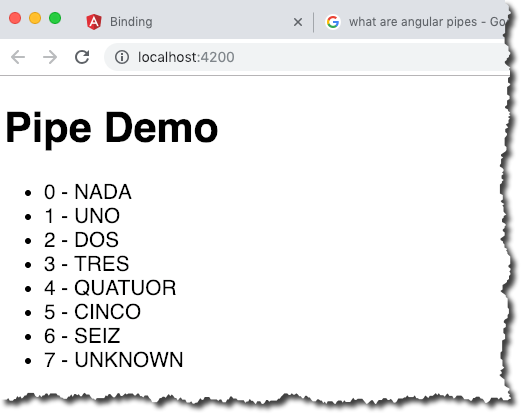


Figure 8-1. Output of the Pipe exercise

Chapter 9 Lifecycle hooks

A component goes through some lifecycle events (which are managed by Angular). Here they are;

1. It gets created
2. Renders it
3. Creates and renders its children (if it has any)
4. Process changes e.g. when its data-bound properties changed
5. then destroys it before removing its template from the DOM

Angular provides from lifecycle hooks we can use so we can tap into these events as they happen.

Chapter10 Services

Basic steps in creating a service

1. Create the service class
2. Decorate it with @Injectable
3. Inject it in the component where you want to use it

These are all very familiar steps, you’ve done a similar pattern of work with components.

There are 2 ways to provide the service (you can inject it at the root level). This way, it will be available from any component in the application and it will also be available to other services. The second way is to provide it to specific components only.

If you want the service to be available at the root level, you need to write the decorator of the service this way

@Injectable({

providedIn: 'root'

})

If you don’t need the service to be available at the root level, you can simply use the @Injectable decorator without any argument, like this

@Injectable()

But now, you’ll need to inject it to the specific component, like this

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css'],

providers: [AppService]

})

See the “providers” property in the JSON argument? Had you injected the service at the root level, you wouldn’t have to write that.

Listings 10-1, 10-2 and 10-3 shows how to create and use services.

Listing 10-1. app.service.ts

import { Injectable } from "@angular/core"; 1

@Injectable({ 2

providedIn: 'root'

})

export class AppService {

getFoo(): any[] { 3

return [

{ lastname : "Gosling",

firstname: "James"

},

{ lastname : "Hevery",

firstname: "Misko"

},

{ lastname : "Dahl",

firstname: "Ryan"

}

]

}

}

|  |  |
| --- | --- |
| [1](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO1-1) | We need the Injectable module, so let’s import it |
| [2](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO1-2) | When we want to create a component, we decorate a class with @Component. When we wanted a pipe, we decorated the class with @Pipe. Now we want a service, but we won’t decorate it @Service, that’s not how to do it. We decorate a service class with @Injectable instead. We want this service to be available to all components; that way, we create the instance of the service only once (a Singleton), so, we use the "providedIn: 'root'" property |
| [3](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO1-3) | The service has a method named getFoo() which returns an array of JSON objects |

Listing 8-2. app.component.ts

import { Component } from '@angular/core';

import {AppService} from './app.service';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css'],

providers: []

})

export class AppComponent {

title = 'Services 2';

names: any[] = null;

constructor(private fooservice:AppService){ 1

}

ngOnInit() {

this.names = this.fooservice.getFoo(); 2

}

}

|  |  |
| --- | --- |
| [1](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO2-1) | To make the service available to our component, we inject that dependency on the constructor |
| [2](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/scratch.html#CO2-2) | Now, we call the service |

Listing 8-3 shows how to use the component in the associated HTML

Listing 8-3. app.component.html

<h1>{{ title }}</h1>

<ul>

<li \*ngFor="let name of names">

{{name.lastname}}

</li>

</ul>

Chapter 11 HTTP

In order to get resources or data over HTTP, we need to use the HttpClient module of Angular. You can import it from @angular/common/http.

The tricky part about working with the HttpClient is that, a call like this

HttpClient.get(someURL)

Won’t return the actual data you’re expecting. For example, if “someURL” in our code snippet above is an RESTful enpoint that’s supposed to give us an array of JSON object (let’s say), that’s not what you’ll get. What you’ll get is an Observable object.

Think of an Observable object like a stream (it’s actually a stream). Before you can get any data from it, you have to subscribe to it and wait as it emits data. Only then can you get access to the data payload of the Observable object.

Listings 11-1, 11-2, 11-3 and 11-4 shows how to work with HttpClient.

Listing 11-1. app.service.ts

import { Injectable } from "@angular/core";

import { Observable } from "rxjs";

import { tap } from "rxjs/operators";

import { HttpClient } from "@angular/common/http"; 1

@Injectable({ 2

providedIn: 'root'

})

export class BookService {

private bookurl = 'api/books.json'; 3

constructor(private http: HttpClient) {} 4

getBooks():Observable<any> { 5

return this.http.get(this.bookurl).pipe( 6

tap(data => console.log(JSON.stringify(data)))

);

}

}

|  |  |
| --- | --- |
| [1](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/http.html#CO1-1) | We need the HttpClient, so, let’s import it |
| [2](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/http.html#CO1-2) | We’d like this class to be a Service |
| [3](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/http.html#CO1-3) | we won’t make a separate app server to serve the "books.json" file. Instead, we’ll make the file available for download as <http://oururl:4200/api/books.json>. You need to do three things here, (1) you need to make changes to "angular.json" (located in the root folder of the project), see Listing 11-2 (2) you need to create an "/api" folder under the "/src" folder of the project and (3) create the file books.json and put in the folder "/src/api/books.json", see Listing 11-3 |
| [4](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/http.html#CO1-4) | Let’s inject the HttpClient dependency into the constructor or our service |
| [5](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/http.html#CO1-5) | Our service has a method called getBooks(). It returns an Observable |
| [6](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/http.html#CO1-6) | The call http.get(this.bookurl) returns an Observable object. I’m using pipe and tap just to peek into the data payload without really consuming it. The pipe and tap are optional in this call; we could have simply written this line as return this.http.get(this.bookurl);, and we’d be fine |

In order for our local web server to serve the books.json file, we need to make some changes on the "angular.json" file (located on the root folder of the project). Listing 11-2 shows the annotated snippet of angular.json, and where to make the additional entry

Listing 11-2. angular.json

...

"architect": {

"build": {

"builder": "@angular-devkit/build-angular:browser",

"options": {

"outputPath": "dist/http",

"index": "src/index.html",

"main": "src/main.ts",

"polyfills": "src/polyfills.ts",

"tsConfig": "src/tsconfig.app.json",

"assets": [

"src/favicon.ico",

"src/assets",

"src/api" 1

...

|  |  |
| --- | --- |
| [1](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/http.html#CO2-1) | You have to add this line to the "architect" section of the angular.json file. This will allow us to access. You’ll need to restart ng serve for the changes to take effect |

Listing 11-3 shows the contents of books.json

Listing 11-3. /src/api/books.json

[

{

"title": "War and Peace",

"author": "Leo Tolstoy",

"price": 15,

"rating": 4.5

},

{

"title": "Nicholas Nickleby",

"author": "Charles Dickens",

"price": 12,

"rating": 3.5

},

{

"title": "Memoirs of Sherlock Holmes",

"author": "Arthur Conan Doyle",

"price": 35,

"rating": 5

},

{

"title": "Scarlet Letter",

"author": "Nathaniel Hawthorne",

"price": 12,

"rating": 3

},

{

"title": "Don Quixote",

"author": "Cervantes",

"price": 20,

"rating": 4

}

]

Listing 11-4.app.component.ts

import { Component, OnInit } from '@angular/core';

import { BookService } from './app.service';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent implements OnInit {

title = 'http';

books: any[];

errorMessage:any;

constructor(private bookservice: BookService) {} 1

ngOnInit() { 2

this.bookservice.getBooks().subscribe( 3

book => {

this.books = book

},

error => this.errorMessage = <any>error

);

}

}

|  |  |
| --- | --- |
| [1](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/http.html#CO3-1) | Let’s inject our BookService into the constructor |
| [2](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/http.html#CO3-2) | I chose call the BookService’s getBooks() on **ngInit** so that we get to pull the data (via http) only when the Component’s data are already bound to the HTML elements; and when the HTML elements have been fully rendered in the browser |
| [3](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/http.html#CO3-3) | bookservice.getBooks() returns an Observable object. In order to get to its payload, we have to subscribe() to it. The subscribe() method accepts 3 parameters, but we’re only using two in this example. The first parameter is a lambda function, where the http.get’s payload is passed. This is where we get to assign the actual content of the get() method to our Component’s books property (this.books). The second parameter of the subscribe() method is another lambda where we can handle errors (if they occur). |

Chapter 12 Routing

Routing is Angular’s way to build SPA (Single Page Applications). The whole idea of SPA is to provide routing that’s local in the browser — no round trips to the server. So, Angular’s routing is also referred to as client-side or browser-side routing.

Think of routing this way; if you have multiple components, what we’d like to do is to provide anchor elements in the main HTML (index.html) such that each anchor link points to a Component. Whenever an anchor link is clicked by the user, the whole screen gets occupied by the content of the Component associated with tha anchor link.

Let’s see how we the index.html looks like (shown in listing 11-1)

Listing 11-1. index.html

<h1>{{ title }}</h1>

<ul>

<li>

<a [routerLink]="['/one']">One</a> 1

</li>

<li>

<a [routerLink]="['/two']">Two</a>

</li>

</ul>

<router-outlet></router-outlet> 2

|  |  |
| --- | --- |
| [1](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/routing.html#CO1-1) | We provide an anchor element so the user can click on it. We use the the routerLink built-in directive to tie this anchor link to a Component. Right now, it’s just pointing to an array — ['/one'] — this array element. When a user clicks this link, the URL changes to "http://someURL:4200/#/One". When the browser’s URL changes, the Angular router will look for a route definition that matches the URL path segment. The route definitions are found in "app.module.ts" (see listing 11-2) |
| [2](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/routing.html#CO1-2) | The <router-outlet> element is where we will display the content of the selected Component |

Listing 11-2. app.module.ts

import { BrowserModule } from '@angular/platform-browser';

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

import { AppRoutingModule } from './app-routing.module';

import { AppComponent } from './app.component';

import { RouterModule } from '@angular/router';

import { OneComponent } from './one/one.component'; 1

import { TwoComponent } from './two/two.component';

import { WelcomeComponent } from './welcome/welcome.component';

@NgModule({

declarations: [ 2

AppComponent,

OneComponent,

TwoComponent,

WelcomeComponent

],

imports: [

BrowserModule,

AppRoutingModule,

RouterModule.forRoot([ 3

{path: 'one', component: OneComponent }, 4

{path: 'two', component: TwoComponent },

{path: 'welcome', component: WelcomeComponent },

{path: '', redirectTo: 'welcome', pathMatch: 'full'}

], {useHash: true})

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

|  |  |
| --- | --- |
| [1](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/routing.html#CO2-1) | Any additional component that you will use for routing should be imported in the module file. In this example project, I created 3 additional components (OneComponent, TwoComponent and WelcomeComponent); all of them needs to be imported |
| [2](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/routing.html#CO2-2) | All the additional Components also needs to be declared |
| [3](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/routing.html#CO2-3) | This is how we define the routes for the app. We use the RouterModule.forRoot() method to create the route definitions for the app. The route definition is an array of JSON objects. Each JSON object is a route |
| [4](file:///Users/ted/Library/Mobile%20Documents/com~apple~CloudDocs/adoc/routing.html#CO2-4) | A route object has two properties. The first one is the **path**, this is the path that’s specified in the [routerLink] directive (see Listing 11-1). The second one is the name of the Component that we’d like to associate with this path |

If you’re following this project, you’ll need to generate. Example 11-1 shows the angular CLI commands to generate the 3 other components we need for this example.

Example 11-1. How to generate additional components.

ng g c OneComponent

ng g c TwoComponent

ng g c WelcomeComponent

Listing 11-3 shows the contents of **one.component.ts**

Listing 11-3. /app/one/one.component.ts

import { Component, OnInit } from '@angular/core';

@Component({

templateUrl: './one.component.html',

styleUrls: ['./one.component.css']

})

export class OneComponent implements OnInit {

title = "This is one component"

constructor() { }

ngOnInit() {

}

}

Appendix A: How to work in a Cloud9 environment

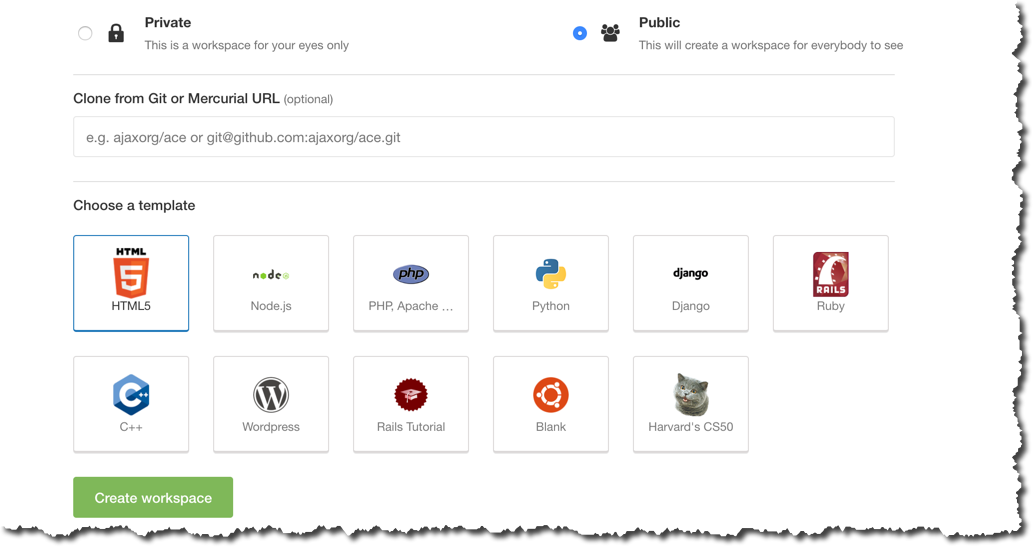
Presumably, you already have a C9 account. I already have an existing C9 account prior to Amazon acquiring C9. At the time of writing, sign ups on <https://c9.io/signup> are no longer accepted. You will instead be referred to AWS C9 service. Anyway, if you happen to have a C9 account, here’s how to use it a dev environment for Angular development.

First, login to <https://c9.io/login>, then create a workspace.

sudo apt-get update

Next, we need to install nvm (Node Virtual manager).

Create a workspace, choose Ubuntu blank.



sudo apt-get update

then install nvm

nvm install node

then create an angular project

and then, make this change to package.json

{

"name": "hello",

"version": "0.0.0",

"scripts": {

"ng": "ng",

"start": "ng serve --host $IP --port $PORT --public-host --disable-host-check",

"build": "ng build",

"test": "ng test",

"lint": "ng lint",

"e2e": "ng e2e"

},

find out the hostname in Cloud 9

npm start

open another browser, (missing instructions)

Notes

<https://medium.com/better-programming/expressionchangedafterithasbeencheckederror-in-angular-what-why-and-how-to-fix-it-c6bdc0b22787>

How to get values of a DOM element on the template from the Component

1. import ElementRef from angular core
2. name the DOM element from the template using the # marker
3. Declare @ViewChild()
4. Then, do something inside one of the lifecycle hooks, e.g. afterViewInit or OnInit. NOTE that if you use the @ViewChild on an element which has an \*ngIf (maybe even \*ngFor), it might result to “undefined errors”. TODO > find the lifecycle hook where you can use @ViewChild and the structural directives.

# Simple binding

**Task** — Bind data from a back-end source or model to a UI element (DOM element)

**Solution** — Use the interpolation operator of Angular

The code listing below shows the annotated snippet of app.component.ts (this is the default component created during the project creation using Angular CLI).

import { Component } from '@angular/core';

@Component({

selector: 'app-root',

template: ` 1

<h1>{{lastname}} , {{firstname}}</h1> 2

`

})

export class AppComponent {

lastname: string = "Doe"; 3

firstname: string = "John";

}

|  |  |
| --- | --- |
| [1](http://localhost:8080/simple_binding.html#CO1-1) | You can use the *template* property instead of the *templateUrl* property; if the HTML template isn’t very complicated, there’s nothing wrong with this technique. Using *template* has the advantage of having the Component and Template on the same file. The only problem is, you lose the code hinting and auto-completion abilities of the editor |
| [2](http://localhost:8080/simple_binding.html#CO1-2) | The {{ }} double curly brace pair is called the *interpolation* operator. You can put properties of the associated Component (that are declared public) inside the interpolation operator |
| [3](http://localhost:8080/simple_binding.html#CO1-3) | lastname is declared as a property of the AppComponent class. A property is *public* by default in TypeScript. Any public property (or function) in TypeScript can be accessed from the HTML template |

You can even use functions inside the interpolation operator, as shown in the sample code below.

import { Component } from '@angular/core';

@Component({

  selector: 'app-root',

  template: `

    <h2>{{ doSomething() }}</h2>

  `

})

export class AppComponent {

  doSomething(): string {

    return "Hello World";

  }

}

This code sample works because the doSomething() function returns something (a string, in this case. You may return other types, not only string. Try to return an booleans, numbers or even arrays).

# Property Binding

If we were to use an image in our HTML template, we might code it like the following;

import { Component } from '@angular/core';

@Component({

selector: 'app-root',

template: `

<h1>{{lastname}} , {{firstname}}</h1>

<img src="{{imageUrl}}">. 1

`

})

export class AppComponent {

lastname: string = "Doe";

firstname: string = "John";

imageUrl: string = "/assets/img/john\_doe.png"; 2

}

|  |  |
| --- | --- |
| [1](http://localhost:8080/property_binding.html#CO1-1) | We’re setting the *src* property of the *img* tag to a Component value using interpolation; no surprises here |
| [2](http://localhost:8080/property_binding.html#CO1-2) | We’re exposing the *imageUrl* as a public property of the AppComponent class, so that the HTML template can resolve its value; no surprises here either |

While we can surely use interpolation to set properties of DOM elements, interpolation is preferred if we’re using it to populate the innerHTML of an element; but when setting the value of a DOM element’s properties, the preferred way to do it is to use the bracket operator — like in the following code;

import { Component } from '@angular/core';

@Component({

selector: 'app-root',

template: `

<h1>{{lastname}} , {{firstname}}</h1>

<img [src]="imageUrl" [style.width]="imageWidth"> 1

`

})

export class AppComponent {

lastname: string = "Doe";

firstname: string = "John";

imageUrl: string = "/assets/img/john\_doe.png";

imageWidth: string = "80px";

}

|  |  |
| --- | --- |
| [1](http://localhost:8080/property_binding.html#CO2-1) | Surround the property name (*src*, in the case of the *img* tag), with square brackets and set its value to a public property of the Component (imageUrl and imageWidth in our case) |

# Event Binding

You can react to user events by binding DOM events to a function in the Component, like this;

<button (click)=”doSomething()”>Click me</button>

Of course, you need to ensure that the function doSomething() is implemented in the associated TypeScript class.

A more complete code example is found below. In here,

import { Component } from '@angular/core';

@Component({

selector: 'app-root',

template: `

<h1>{{lastname}} , {{firstname}}</h1>

<img [hidden]="hiddenImage"

[src]="imageUrl"

[style.width]="imageWidth"><br/>

<button (click)="hideClick()">show/hide</button> 1

`

})

export class AppComponent {

lastname: string = "Doe";

firstname: string = "John";

imageUrl: string = "/assets/img/john\_doe.png";

imageWidth: string = "80px";

hiddenImage: boolean = false;

hideClick() { 2

this.hiddenImage = !this.hiddenImage;

}

}

|  |  |
| --- | --- |
| [1](http://localhost:8080/event_binding.html#CO1-1) | You can use the actual names of the DOM event in Angular. See in here how we bound the *click* event by simply surrounding it in parens (click) then setting its value to a function in the component (hideClick() in this example) |
| [2](http://localhost:8080/event_binding.html#CO1-2) | When you bind DOM events in the HTML template, it usually calls a function in the Component class. Since we bound the *button* elements (click) in the template and set it to handleClick(), we need to implement that function here in the Component |

# Template variables

You can send data from the HTML template to the associated Component class using *template variables*. Let’s take a simple select element for example;

    <select (change)="changeColor(<parameter>)">

      <option>red</option>

      <option>green</option>

      <option>blue</option>

    </select>

What we’d like to do is to bind the *change* event to a function in the Component — pressumably, a function named changeColor() which takes on a string parameter (either red, green or blue, in this case). When the user clicks on the drop-down list, he can choose one of the colors; that will trigger the change event; our challenge is how to capture the value of the *select* element and send it to the changeColor() function as parameter.

As it turns out, in Angular, we can use the $event object to capture the value of the user’s selection. Whenever a DOM element raises an event (click, change, hover etc.), we can use $event object within the DOM element to inquire many things, one among them is the value property of the DOM element. So, how do get pass the selected color now to the Component? Let’s see the following code;

import { Component } from '@angular/core';

@Component({

selector: 'app-root',

template: `

<strong [style.color]="color">

Favorite Color

</strong>

<select (change)="changeColor($event.target.value)"> 1

<option>red</option>

<option>green</option>

<option>blue</option>

</select>

`

})

export class AppComponent {

color:string;

changeColor(color:string) { 2

this.color = color;

}

}

|  |  |
| --- | --- |
| [1](http://localhost:8080/template_variables.html#CO1-1) | When the *change* event is raised, we can use the the $event object to get the get the value of what the user selected in the dropdown; $event stands for, well, the *event* object and $event.target resolves to the actual DOM element that raised the event in the first place (in our case, the select DOM element); and to get the get the value of the *changed* select element, we write it as $event.target.value |
| [2](http://localhost:8080/template_variables.html#CO1-2) | The color parameter maps to the $event.target.value that was passed in the HTML template |

When you test the app, you should see something like this;

A screenshot of a cell phone

Description automatically generated

When the user clicks on any color in the dropdown, the text color of “Favorite Color” changes accordingly. This happens because the *color property* of the DOM element is bound to the *color property* of the AppComponent class.

# Two way binding

So far, we’ve bound headers, paragraphs and image sources to a value in the Component class, but all these DOM elements are static, they not dynamic and interactive. What if we try to bind a input elements value’s to a Component’s property? That would look something like this in code;

import { Component } from '@angular/core';

@Component({

  selector: 'app-root',

  template: `

    <h1>{{ name }}</h1>

    <input [value]="name" type="text">`

})

export class AppComponent {

  name: string = "John Doe"

}

This code effectively binds the input element’s value to the name property of the AppComponent class; but the binding is one way, as you will soon discover when the browser reloads

A screenshot of a cell phone

Description automatically generated

If you try to type on the input box, you’ll soon discover (to your disappointment) that only the value in the input box changes — as a result of your typing — but it doesn’t have any effect on the interpolated string. Clearly, this kind of binding is one way. When we’d like to achieve, is two-way binding. When we type something in the input box, the the innerHTML value of the H1 tag (in the template) should change accordingly.

A screenshot of a cell phone

Description automatically generated

We already know how to send data from the template to the component using template variables; why don’t we use that technique to try and see if it can work.

# @ViewChild

1. Strict mode was added in ECMAScript 5, you can find more about it here <https://mzl.la/2DSREWK> [↑](#footnote-ref-1)