* Due date to be announced.
* 20% deduction each day this is late.

Contents

[Set Up 1](#_Toc503772288)

[Angular CLI 1](#_Toc503772289)

[Creating and Running an Application 1](#_Toc503772290)

[Nested Components 3](#_Toc503772291)

[Passing Data to a Nested Component 5](#_Toc503772292)

[Services 6](#_Toc503772293)

[Pipes 8](#_Toc503772294)

[Custom Pipes 10](#_Toc503772295)

[Basic Routing in Angular 12](#_Toc503772296)

[Modules 13](#_Toc503772297)

[Enabling Route Parameters 14](#_Toc503772298)

[Managing Sessions 16](#_Toc503772299)

[Enabling Production Mode 18](#_Toc503772300)

## Set Up

The Angular team uses the Node package manager to download the required packages. Be sure to download and install a recent version of NodeJS otherwise your project will not build properly:

<https://nodejs.org/en/download/>

### Angular CLI

Angular relies on such a huge code base. Angular CLI is here to help. Angular CLI is endorsed by the Angular team to help automate Angular development and to manage the many different configurations and pieces needed.

The latest version with good documentation can be found here:

<https://github.com/angular/angular-cli>

Install angular-cli with the following command:

**>npm install -g @angular/cli**

## Creating and Running an Application

Example 1: Creating and Running an Application

This example shows the steps needed to create a new Angular application with Angular CLI. In a folder where you want your application, create the application with the following command. Then wait about 5 minutes:

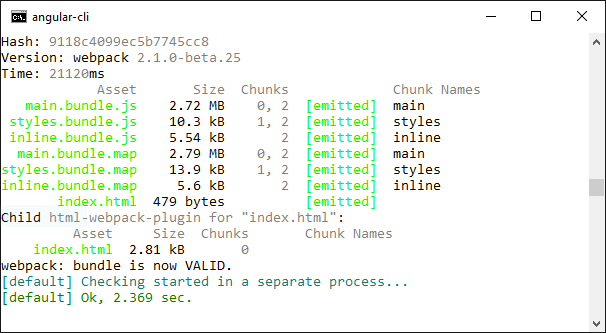
**>ng new myapp**

After the installation, cd into the **myapp** directory where index.html is and run the following command to build and launch your application:

>**cd myapp**

**myapp>ng serve**

When you see the following output you can view the application.



To view your application, navigate to the following address in your browser:

<http://localhost:4200/>

Once the browser launches, the application will stay open and respond to changes as you update the code.



## Nested Components

Nested components effectively allow us to create custom directives. The Angular team calls this a component directive. Here is a component that can be accessed with its selector as a directive:

@***Component***({  
 *// Selector uses lower case with hyphens.* **selector**: **'second-directive'**,  
 **template**: **`<h3>This is the second directive!</h3>`**,  
})  
**export class** NewDirective {  
}

Using the selector, the new component can be implemented within another component.

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

@Component({

selector: 'app-root',

template: `<h1>Hello world! {{title}}</h1>

<second-directive></second-directive>`

})

export class AppComponent {

public title = 'This is Angular!';

}

Example 2: Creating a Directive

This example shows the implementation of a component named *NewDirective* which can be implemented in an existing partial view. First, to build this example and to review the set-up from the last class. We will begin by adding the required configuration files and index.html page which loads all of the required resources which they reference. To get the latest versions of these files go to <https://github.com/angular/quickstart>

Here is the class and corresponding component that contains are brand new directive. This can be re-used by any component that refers to it with the selector <second-directive>.

**app/newDirective.ts**

|  |
| --- |
| import { Component } from '@angular/core';  // Define second directive.  @Component({  // Selector uses lower case with hyphens.  selector: 'second-directive',  template: `<h3>This is the second directive!</h3>`,  })  export class NewDirective {  } |

Next, this is the component that loads the new component.

**app/app.component.ts**

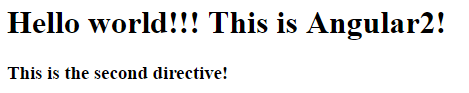
|  |
| --- |
| import { Component } from '@angular/core';  @Component({  selector: 'app-root',  template: `<h1>Hello world! {{title}}</h1><second-directive></second-directive>`  })  export class AppComponent {  public title = 'This is Angular!';  } |

Our module groups and loads all components.

**app/app.module.ts**

|  |
| --- |
| import { NgModule } from '@angular/core';  import { BrowserModule } from '@angular/platform-browser';  import { NewDirective } from './newDirective';  import { AppComponent } from './app.component';  @NgModule({  imports: [ BrowserModule ],  declarations: [ AppComponent, NewDirective ],  bootstrap: [ AppComponent ]  })  export class AppModule { } |

When the browser launches the following .



Exercise 1

What does a module do?

|  |
| --- |
|  |

Exercise 2

What option inside @NgModule loads the directive so it can be used by any component within the module?

|  |
| --- |
|  |

Exercise 3

Explain in your own words what the selector option does.

|  |
| --- |
|  |

Exercise 4

Create a third directive which is nested in the second directive of Example 2. Display the text “hello from the third directive” in this new directive. Show the file that contains your new component here (2 marks):

|  |
| --- |
|  |

Show the updated newDirective.ts file here (1 mark):

|  |
| --- |
|  |

### Passing Data to a Nested Component

To pass data from a parent component to a child component input binding is required in the child to expose its properties. The parent can then set the value of the child’s properties.

Example 3: Passing Data to a Nested Component

This example demonstrates how a child component can receive data from the parent component. This example begins with the code from Example 2. Then, a child component is needed with an input binding so it can receive data from the parent through the child’s firstName property. To make this work, you can modify the newDirective.ts file from Example 2 with the changes shown by the highlighting.

**app/newDirective.ts**

|  |
| --- |
| import { Component, Input } from '@angular/core';  // Define second directive.  @Component({  selector: 'second-directive',  template: `<h3>This is the second directive! {{firstName}}</h3>`,  })  // This is really just a component.  export class NewDirective {  @Input()  firstName: string;  } |

Then, in the parent component you can reference the property of the child using square brackets around the child property inside the selector.

**app/app.component.ts**

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  selector: 'app-root',  template: `<h1>Hello world! {{title}}</h1>  <second-directive [firstName]="author"></second-directive>`  })  export class AppComponent {  public title = 'This is Angular!';  public author = "Joyce.";  } |

Here is our output from this example:



Exercise 5

What is the role of the input decoration in the child component?

|  |
| --- |
|  |

Exercise 6

Starting with Example 3, create a directive called ThirdDirective. Reference ThirdDirective from SecondDirective. The SecondDirective must remain nested inside AppComponent. Pass data from SecondDirective to ThirdDirective and show this data in the output of ThirdDirective. Show your modified app.module.ts file here:

|  |
| --- |
|  |

Show your ThirdDirective component here:

|  |
| --- |
|  |

Show your modified SecondDirective component here:

|  |
| --- |
|  |

## Services

Just like in Angular 1, Angular services allow you to modularize your code so it can be re-used throughout the application. Angular services are defined in simple classes. We no longer need any convoluted references to them. We just call the function in the class and the code can be re-used by any component.

Example 4: Creating a Service

This example shows how to create a simple service that can be re-used from any component. To build this example, start with Example 2.

Then add in this new service.

**app/myDataService.ts**

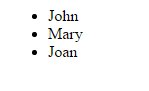
|  |
| --- |
| import { Component } from '@angular/core';  // This is the service.  export class MyDataService {  names: Array<any>;  constructor() {  this.names = ['John', 'Mary', 'Joan'];  }  getNames() {  return this.names;  }  } |

Next, replace the contents of app.component.ts with the following code which creates an instance of the service and calls it’s getNames() method. You will notice that we have a new way to create an instance of a class right in the constructor header. To do this we must first declare a provider in the @Component area. Then we can create an object of MyDataService in the constructor header as shown with the green highlighting below.

**app/app.component.ts**

|  |
| --- |
| import { Component } from '@angular/core';  import { MyDataService } from './myDataService';  @Component({  selector: 'app-root',  template: `<h1>Hello world! {{title}}</h1>  <ul><li \*ngFor="let name of names">{{name}}</li></ul>`,  // 'providers' allows you to create and pass an instance  // of the class to the constructor header.  providers: [MyDataService]  })  export class AppComponent {  public title = 'This is Angular!';  names: Array<any>;  // Create instance of 'MyDataService' right in the constructor  // header.  constructor(myDataService: MyDataService) {  // Use service to call getNames() method.  this.names = myDataService.getNames();  }  } |

When you run the program after making the changes you will notice that the data from the service is retrieved and displayed in the output:



Exercise 7

In the data service, create a new function that returns your name. Then modify app.component.ts to call your new function and show the name along with all of the original content. Show the revised myDataService.ts file here: (1 mark)

|  |
| --- |
|  |

Show your modified app.component.ts file here: (1 mark)

|  |
| --- |
|  |

Exercise 8

Starting with Example 4, create a second service in a separate file. This service must contain a function to convert Celsius to Fahrenheit. Adjust the code in app.component.ts to call the new service’s function to perform a conversion from 6 degrees Celsius to Fahrenheit. Show this result in HTML while also showing the original output from Example 4.

Hint: To create an instance of the service in the constructor you will need to include a reference to it in the providers list and in the constructor header. Your providers list and constructor header should look like the following once you are done:

providers: [MyDataService,TemperatureService]

constructor(myDataService: MyDataService, temperatureService: TemperatureService)

Show the code for your new service here:

|  |
| --- |
|  |

Show your updated app.component.ts file here:

|  |
| --- |
|  |

## Pipes

Pipes allow us to format values. Angular has some really helpful ones.

Example : Date Pipe

We have not discussed dates yet but Angular’s date pipe allows several helpful formatting options. For a full range of date and time formats see:

<https://angular.io/docs/ts/latest/api/common/index/DatePipe-pipe.html>

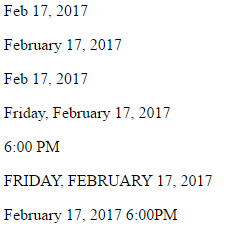
This example begins with the starter application from Angular CLI. Safari does not handle the date options so well so to enable this include the following reference in index.html:

|  |
| --- |
| <script src="https://cdn.polyfill.io/v2/polyfill.min.js?features=Intl.~locale.en"></script> |

Once you have that running replace the contents of AppComponent with the following code:

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  selector: 'app-root',  template: `<!-- Feb 17, 2017 -->  {{ mydate | date }}<br>  <!-- February 17, 2017 -->  <p>{{ mydate | date: 'yMMMMd' }}<br>  <!-- Feb 17, 2017 -->  <p>{{ mydate | date: 'mediumDate' }}<br>  <!-- Friday, February 17, 2017 -->  <p>{{ mydate | date: 'fullDate' }}<br>  <!-- 6:00 PM -->  <p>{{ mydate | date: 'shortTime' }}<br>  <!-- FRIDAY, FEBRUARY 17, 2017 -->  <p>{{ mydate | date:'fullDate' | uppercase}}</p>  <!-- February 17, 2017 6:00PM -->  <p>{{ mydate | date:'MMMM d, y h:mma' }}</p>  `  })  export class AppComponent {  // Months start counting at 0.  mydate = new Date(2017, 1, 17, 18, 0, 30); // October 23, 1940  } |

The output from this is:



Exercise

Using the formatting options at

<https://angular.io/docs/ts/latest/api/common/index/DatePipe-pipe.html>

See if you can figure out how to modify Example 5 to display the date in the following format:

Friday, February 17, 2017 6:00PM

If you are looking for automation for other common formats there likely is an Angular pipe for it somewhere online.

Example : Currency Pipe

This example shows how a currency pipe can be used to convert numeric amounts to US dollars. It begins with either the Angular CLI project or the Angular Quickstart build from the Angular team. Then replace the contents of app.component.ts with the following code:

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  selector: 'app-root',  template: `  {{price | currency:'USD':true}}<br/>  `,  })  export class AppComponent {  price: number = 23.23333;  constructor() {  }  } |

The output displayed from this pipe is:



### Custom Pipes

As you would expect it is possible to create your own custom pipes to provide quick and professional formatting of your inputs.

Example : Custom Pipes

This next example begins with the Angular CLI starter project. To build it, add a file named app.fahrenheitPipe.ts to the same folder where your app.component.ts file exists. This code is set up so a Celsius value is received. A conversion and formatting is then applied and this formatted Fahrenheit value is returned.

|  |
| --- |
| import { Pipe, PipeTransform } from '@angular/core';  @Pipe({ name: 'fahrenheitPipe' })  export class FahrenheitPipe implements PipeTransform {  transform(celsius: number): string {  let fah = celsius \* 9 / 5 + 32;  let output = fah.toString() + " " + '\u2109'; // Unicode for Fahrenheit;  return output;  }  } |

To be recognized by all classes in the module, this pipe must be imported into the AppModule and included in the declarations:

|  |
| --- |
| import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';  import { FormsModule } from '@angular/forms';  import { HttpModule } from '@angular/http';  import { FahrenheitPipe } from './app.fahrenheitPipe';  import { AppComponent } from './app.component';  @NgModule({  declarations: [  AppComponent,  FahrenheitPipe  ],  imports: [  BrowserModule,  FormsModule,  HttpModule  ],  providers: [],  bootstrap: [AppComponent]  })  export class AppModule { } |

We can then include the pipe in our app.component.ts file to display Celsius values as Fahrenheit.

|  |
| --- |
| import { Component } from '@angular/core';  import { FahrenheitPipe } from './app.fahrenheitPipe';  @Component({  selector: 'app-root',  template: `<p>{{celsius | fahrenheitPipe }}</p>  `,  })  export class AppComponent {  celsius: number;  constructor() {  this.celsius = 33;  }  } |

The output from all of this is the temperature in Fahrenheit:



Exercise

Create a pipe to receive a ten digit phone number in any format and output it as:

(604) 922-2222

|  |
| --- |
|  |

## Basic Routing in Angular

Routing lets us implement navigation in your Angular application so the proper component and view are selected and displayed whenever a link is clicked.

Example 8: Basic Routing

This demonstration begins with the helloworld app. The code is then transformed to implement hyperlinks that retrieve and display different child views when clicked. The enable these views, add these three components which each have inline views.

**app/app.page-a.ts**

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  template: `This is page A.`  })  export class PageAComponent { } |

**app/app.page-b.ts**

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  template: `This is another page.`  })  export class PageBComponent { } |

**app/app.pagedefault.ts**

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  template: `This page does not exist.`  })  export class PageDefault { } |

Next, replace the code inside app.component.ts file with the following code that contains a template which includes a menu with hyperlinks. The hyperlinks are created using the <routerLink> tag with a value of the reference from the router. The <router-outlet> element serves as a placeholder for the child view which appears when the appropriate link is selected.

**app/app.component.ts**

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  selector: 'app-root',  template:  `<h1>This is the header</h1>  <nav>  <a routerLink="/page-a" routerLinkActive="active">A</a> |  <a routerLink="/page-b" routerLinkActive="active">B</a>  </nav>  <!-- Where router should display a view -->  <router-outlet></router-outlet>`  })  export class AppComponent { } |

With a larger project you may want to store the routing logic in a separate file such as in app.routing.ts. The appRoutes array below defines a series of routing objects which include a hyperlink reference and corresponding component displayed. Since the components are referenced by the router they must first be imported. These components have been loaded through the import statements.

**app/app.routing.ts**

|  |
| --- |
| import { ModuleWithProviders } from '@angular/core';  import { Routes, RouterModule } from '@angular/router';  import { AppComponent } from './app.component';  import { PageAComponent } from './app.page-a';  import { PageBComponent } from './app.page-b';  import { PageDefault } from './app.pagedefault';  const appRoutes: Routes = [  { path: 'page-a', component: PageAComponent },  { path: 'page-b', component: PageBComponent },  { path: '', redirectTo: '/page-a', pathMatch: 'full' },  { path: '\*\*', component: PageDefault }  ];  export const routing: ModuleWithProviders = RouterModule.forRoot(appRoutes); |

### Modules

Modules group components together. This not only helps to organize your code, the modularization also allows us to split the application up so it does not need to be loaded all at once. Later we will discuss how modules can be loaded when they are needed using a technique called lazy loading.

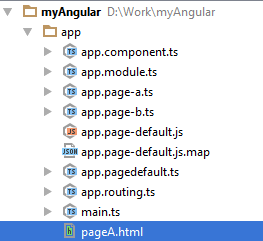
In app.module.ts, we are declaring AppModule which includes four different components and we are also importing the router defined in the file previously discussed.

**app/app.module.ts**

|  |
| --- |
| import { NgModule } from '@angular/core';  import { BrowserModule } from '@angular/platform-browser';  import { FormsModule } from '@angular/forms';  import { AppComponent } from './app.component';  import { PageDefault } from './app.pagedefault';  import { PageAComponent } from './app.page-a';  import { PageBComponent } from './app.page-b';  import { routing } from './app.routing';  @NgModule({  imports: [BrowserModule, FormsModule, routing],  declarations: [AppComponent, PageDefault,  PageAComponent, PageBComponent],  bootstrap: [AppComponent],  })  export class AppModule { } |

Example 9: TemplateUrl

This example shows how to separate your HTML into different pages. To do this, begin with Example 8, then in the app directory create the page pageA.html.



Paste this code inside the HTML file.

|  |
| --- |
| <**H3**>This is page A.</**H3**> |

Then replace the contents of **app.page-a.ts** with the following code to reference the new HTML page from the root directory of the application.

|  |
| --- |
| **import** { Component } **from '@angular/core'**; @Component({  **templateUrl**:**'./pageA.html'** }) **export class** PageAComponent { } |

### Enabling Route Parameters

Example 10: Enabling Route Parameters

This example shows how to enable route parameters. Start with the solution from Example 8. Then, replace the link information for page-b in app/app.routing.ts with the following code which sets a value for the parameter named ‘id’:

**app/app.routing.ts**

|  |
| --- |
| { path: 'page-b/:id/:firstname', component: PageBComponent }, |

Then, replace the routerLink in the template option of the parent component:

**app/app.component.ts**

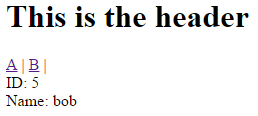
|  |
| --- |
| <a routerLink="/page-b/5/bob" routerLinkActive="active">B</a> |

Next replace the entire contents of page-b.ts. This new version retrieves the value of the parameter in the ngOnInit method and stores it in a property named *id*:

**app/app.page-b.ts**

|  |
| --- |
| import { Component } from '@angular/core';  import { ActivatedRoute, Params } from '@angular/router';  @Component({  template: `ID: {{id}}<br/>  Name: {{firstName}}`  })  export class PageBComponent {  id: number;  firstName: string;  // The constructor sets up the class.  constructor(private route: ActivatedRoute) {  }  // ngOnInit() gets called after the class is set up.  // At this point, the class is able to handle the parameters passed to it.  ngOnInit() {  this.route.params.forEach((params: Params) => {  let localID = params['id'];  let localName = params['firstname'];  this.id = localID;  this.firstName = localName;  });  }  } |

When you run the project now, you will see the following:



Exercise 11

How are the constructor and ngInit functions different?

|  |
| --- |
|  |

Exercise 12

Add the following constant declaration to the constructor in Example 10.

|  |
| --- |
| // Declare constant.  let str = "This is where the class is initialized."; |

Log the value of str in the console. Show a screenshot of the console with the output:

|  |
| --- |
|  |

Exercise 13

Starting with Example 10, set a breakpoint beside the constant declaration inside the constructor and also set a breakpoint inside ngOnInit().

1. Which function gets called first – ngOnInit() or the constructor?

|  |
| --- |
|  |

1. Show a screenshot of your debugger halted beside the breakpoint inside ngOnInit() inside the PageBComponent class.

|  |
| --- |
|  |

(c) Show a screenshot of your debugger halted inside the constructor of PageBComponent.

|  |
| --- |
|  |

## Managing Sessions

You may want to store values for the duration of a visitors stay on your site as they navigate from page to page. For example, you might want to store a session token while the user is logged in to your web application. That way the client application can access the token to send it with any page request. You can store this information in JavaScript’s storage object.

|  |
| --- |
| // Save data.  sessionStorage.setItem('your\_key', 'your\_value');  // Retrieve data.  let contents = sessionStorage.getItem('your\_key'); |

With *sessionStorage* data is stored until the browser tab is closed. sessionStorage is ideal when storage requirements are temporary and when more data must be stored than in a cookie.

Example : Storage Object

This example begins with the basic routing example from Example 10. Revise the app.page-b.ts code so it allows a user to input anything in a text box. The code in the mychange() function automatically saves whatever is entered in the session.

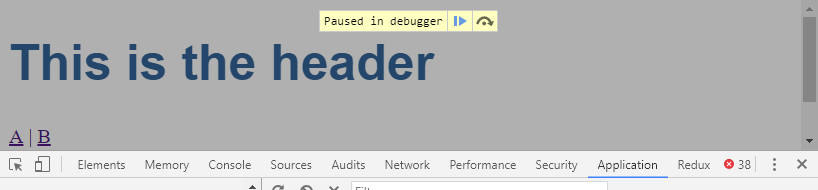
|  |
| --- |
| import { Component } from '@angular/core';  @Component({  template: `This is page b. {{retrievedFromStorage}}  <input type="input" (ngModelChange)="mychange($event)" [ngModel]="mymodel">  `  })  export class PageBComponent {  constructor() {  }  mychange(val:string) {  sessionStorage.setItem("autosave", val);  console.log(val); // updated value  }  } |

Then in app.page-a.ts, add code to display whatever was saved with the session. Specific items are retrieved using the key ‘autosave’.

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  template: `This is page A.<br>This content was set in page-b: {{someItem}}`  })  export class PageAComponent {  someItem: string;  constructor() {  if (sessionStorage.getItem("autosave")) {  this.someItem = sessionStorage.getItem("autosave");  }  }  } |

Exercise 14

Take a screenshot of the local session storage data which is visible in the Chrome debugger window from the Application tab.



Show your screenshot here:

|  |
| --- |
|  |

Exercise 15

Add a second input and method to app.page-b.ts which also saves data to a new key value pair. Modify app.page-a.ts to show the data from the second key value pair while also showing data from the first key-value pair that is provided with the example. Show your new app.page-b.ts code here:

|  |
| --- |
|  |

Show your new app.page-a.ts code here:

|  |
| --- |
|  |

## Enabling Production Mode

You will notice that when running Angular, the following message appears in the console:



Production mode eliminates several of the debug features to speed up deployment of the application. If you ever wish to do this you can enable production mode. Inside main.ts add the code that is highlighted in yellow:

|  |
| --- |
| import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';  import { AppModule } from './app/app.module';  import { enableProdMode } from '@angular/core';  enableProdMode();  platformBrowserDynamic().bootstrapModule(AppModule); |

**Note:** I do not recommend switching to production mode unless you are deploying your application.