# Angular 4 Continued

* Answers in this document are due July 2, 1159PM
* 20% deduction each day this is late.

Contents

[Angular 4 Continued 1](#_Toc482336716)

[Getting the Latest Build 1](#_Toc482336717)

[Nested Components 2](#_Toc482336718)

[Passing Data to a Nested Component 4](#_Toc482336719)

[Services 6](#_Toc482336720)

[Validation 8](#_Toc482336721)

[Form Validation 8](#_Toc482336722)

[Validating Controls and Showing Error Messages 8](#_Toc482336723)

[Asynchronous Calls Using Observables 10](#_Toc482336724)

[Lazy Loaded Modules 13](#_Toc482336725)

[Modules 15](#_Toc482336726)

## Getting the Latest Build

You can always get the latest steps from the Angular team. Download it from <https://github.com/angular/quickstart>

Example : Getting the Latest

If you used the Git clone command to download the project, you can discard the .git folder with the following command:

rm -rf .git **# OS/X (bash)**

rd .git /S/Q **# Windows**

Then, delete non-essential files (optional) You can quickly delete the non-essential files that concern testing and QuickStart repository maintenance (including all git-related artifacts such as the .git folder and .gitignore!) by entering the following commands while in the project folder:

**OS/X (bash)**

xargs -a non-essential-files.txt rm -rf

rm app/\*.spec\*.ts

rm non-essential-files.txt

**Windows**

for /f %i in (non-essential-files.txt) do del %i /F /S /Q

rd .git /s /q

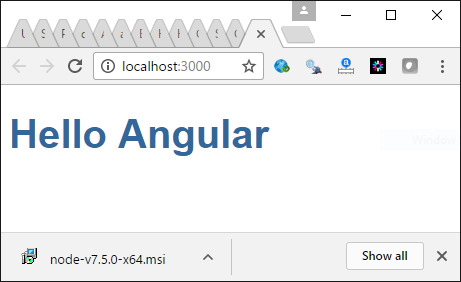
rd e2e /s /q

Next, from the same directory where package.json file resides type:

npm install

npm start

Check to ensure the application runs. You should see the application appear if successful.



## 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: 'my-app',

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

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

})

export class AppComponent {

public title = 'This is Angular2!';

}

Example : 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**

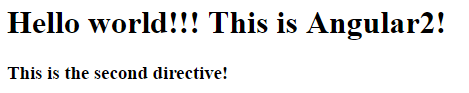
|  |
| --- |
| import { Component } from '@angular/core';  @Component({  selector: 'my-app',  template: `<h1>Hello world! {{title}}</h1><second-directive></second-directive>`  })  export class AppComponent {  public title = 'This is Angular2!';  } |

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

What does a module do?

|  |
| --- |
| A module allows you to group and load all components/classes so that they’re available throughout the entire app. |

Exercise

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

|  |
| --- |
| declarations |

Exercise

Explain in your own words what the selector option does.

|  |
| --- |
| The selector lets you assign a tag name/reference to your component, which can be reference by other components. |

Exercise

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

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

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

|  |
| --- |
| 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>  <third-directive></third-directive>`,  })  export class NewDirective {  } |

### 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 : 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 . 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 with the changes shown by the highlighting.

**app/newDirective.ts**

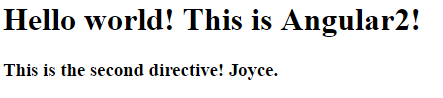
|  |
| --- |
| import { Component, Input } from '@angular/core';  // Define second directive.  @Component({  selector: 'second-directive',  template: `<h3>This is the second directive! {{firstName}}</h3>`,  })  // This is really just a component.  export class NewDirective {  @Input()  firstName: string;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

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: 'my-app',  template: `<h1>Hello world! {{title}}</h1>  <second-directive [firstName]="author"></second-directive>`  })  export class AppComponent {  public title = 'This is Angular2!';  public author = "Joyce.";  } |

Here is our output from this example:



Exercise

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

|  |
| --- |
| It defines an input binding that exposes its properties to the parent, so that the parent can pass in data into the child component. |

Exercise

Starting with , 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:

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

Show your ThirdDirective component here:

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

Show your modified SecondDirective component here:

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

## Services

Just like in Angular 1, Angular2 services allow you to modularize your code so it can be re-used throughout the application. Angular 4 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 : 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**

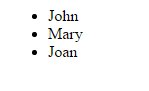
|  |
| --- |
| import { Component } from '@angular/core';  // This is the service.  export class MyDataService {  names: Array<any>;  constructor() {  this.names = ['John', 'Mary', 'Joan'];  }  getNames() {  return this.names;  }  } |

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: 'my-app',  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 Angular2!';  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

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)

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

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

|  |
| --- |
| import { Component } from '@angular/core';  import { MyDataService } from './myDataService';  @Component({  selector: 'my-app',  template: `<h1>Hello world! {{title}}</h1>  <ul><li \*ngFor="let name of names">{{name}}</li>  <li>{{myName}}</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 Angular2!';  names: Array<any>;  myName: string;  // Create instance of 'MyDataService' right in the constructor  // header.  constructor(myDataService: MyDataService) {  // Use service to call getNames() method.  this.names = myDataService.getNames();  this.myName = myDataService.getMyName();  }  } |

Exercise

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:

|  |
| --- |
| import { Component } from '@angular/core';  // This is the service.  export class TemperatureService {  names: Array<any>;  myName: string;  constructor() {  this.names = ['John', 'Mary', 'Joan'];  this.myName = "Sam";  }  celsiusToFahrenheit(fahrenheit: number) {  return (fahrenheit - 32) \* (5/9);  }  } |

Show your updated app.component.ts file here:

|  |
| --- |
| import { Component } from '@angular/core';  import { MyDataService } from './myDataService';  import { TemperatureService } from './temperatureService';  @Component({  selector: 'my-app',  template: `<h1>Hello world! {{title}}</h1>  <ul><li \*ngFor="let name of names">{{name}}</li>  <li>{{myName}}</li>  </ul>  <h1>Fahrenheit: {{fahrenheit}}</h1>  <h1>Celsius: {{celsius}}</h1>`,  // 'providers' allows you to create and pass an instance  // of the class to the constructor header.  providers: [MyDataService, TemperatureService]  })  export class AppComponent {  public title = 'This is Angular2!';  names: Array<any>;  myName: string;  fahrenheit: number;  celsius: number;  // Create instance of 'MyDataService' right in the constructor  // header.  constructor(myDataService: MyDataService, temperatureService: TemperatureService) {  // Use service to call getNames() method.  this.names = myDataService.getNames();  this.myName = myDataService.getMyName();  this.fahrenheit = 6;  this.celsius = temperatureService.celsiusToFahrenheit(this.fahrenheit);  }  } |

## Validation

Similar to Angular JS 1, template validation is possible with the following common validators:

* required
* minlength
* maxlength
* pattern

### Form Validation

We have the ability to check for the form’s validity before enabling submit if we set up the form using *ngForm*:

<form (ngSubmit)="onSubmit()" #myForm="ngForm">

[disabled]="!myForm.form.valid">Submit</button>

</form>

### Validating Controls and Showing Error Messages

We can define our control to implement ngModel if we name it in the manner that is highlighted in green:

<!-- Define control. -->

<input type="text" pattern="[a-zA-Z ]\*" minlength="3" required

[(ngModel)]="myName" name="firstName" #firstName="ngModel">

We may have several validators such as pattern, minlength and required as shown above. Regardless of the validators used we can check for validation status to determine when and how to display an error message:

<!-- Show error message if the control is not valid in general. -->

<div [hidden]="firstName.valid || firstName.pristine">

This control is invalid.

</div>

We also have the ability to show validator specific error messages. The ‘?’ syntax used below is only evaluated if it is not null:

<!-- Show 'invalid' status when required validator is not satisfied. -->

<p \*ngIf="firstName?.errors?.required">This field is required.</p>

Example : Simple Validation

This example shows how to show and hide error messages based on required, pattern and minlength validators. With incorrect entries, error messages are displayed and the submit button is disabled. The button is enabled and the error messages are hidden when the data conforms to the validation requirements. To build this example, start with the solution for .

The validators and form input control requires access to the FormsModule dependency. To provide access add in the following highlighted code to **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';  @NgModule({  imports: [BrowserModule, FormsModule],  declarations: [AppComponent],  bootstrap: [AppComponent]  })  export class AppModule { } |

Then, replace the code in **main.ts** with the following.

**app\app.component.ts**

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  selector: 'my-app',  template:  `  <section>  <form (ngSubmit)="onSubmit()" #myForm="ngForm">  Name:  <input type="text" pattern="[a-zA-Z ]\*" minlength="3" required  [(ngModel)]="myName" name="firstName" #firstName="ngModel" >  <div [hidden]="firstName.valid || firstName.pristine">  This control is invalid.</div>  <p \*ngIf="firstName?.errors?.required">This field is required.</p>  <p \*ngIf="firstName?.errors?.pattern">  Only alphabetical characters are allowed.</p>  <p \*ngIf="firstName?.errors?.minlength">  This entry must have at least three characters.</p>  <button type="submit" class="btn btn-default"  [disabled]="!myForm.form.valid">Submit</button>  </form>  </section>  `  })  export class AppComponent {  myName: string;  constructor() {  this.myName = "frank";  }  } |

Exercise

1. Add a maxlength validator to ensure the maximum length does not exceed 15 characters. Add code to display an error message if the 15 character limit is exceeded. Show your revised app.component.ts file here:

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  selector: 'my-app',  template:  `  <section>  <form (ngSubmit)="onSubmit()" #myForm="ngForm">  Name:  <input type="text" pattern="[a-zA-Z ]\*" minlength="3" maxlength="15" required  [(ngModel)]="myName" name="firstName" #firstName="ngModel" >  <div [hidden]="firstName.valid || firstName.pristine">  This control is invalid.</div>  <p \*ngIf="firstName?.errors?.required">This field is required.</p>  <p \*ngIf="firstName?.errors?.pattern">  Only alphabetical characters are allowed.</p>  <p \*ngIf="firstName?.errors?.minlength">  This entry must have at least three characters.</p>  <p \*ngIf="firstName?.errors?.maxlength">  This entry must have maximum 15 characters.</p>  <button type="submit" class="btn btn-default"  [disabled]="!myForm.form.valid">Submit</button>  </form>  </section>  `  })  export class AppComponent {  myName: string;  constructor() {  this.myName = "frank";  }  } |

1. Why isn’t it possible to make the error message from step (a) appear?

|  |
| --- |
| It prevents the field from fitting in more than 15 characters, so the maximum character limit isn’t reached at all. |

## Asynchronous Calls Using Observables

Often, web applications will need to make requests for remote resources. Waiting for the resources can take several seconds and there is no guarantee the resource will arrive. There are several ways to manage this request so the call can be made on a separate thread and the resource can be received by the main thread in the application when the response arrives. One way to manage this is with observables. You can think of an Observable as a stream of events published by some source. We listen for events in this stream by subscribing to the Observable. In these subscriptions we specify the actions to take when the web request produces a success event or a fail event.

[https://angular.io/docs/ts/latest/guide/server-communication.html#!#http-client](https://angular.io/docs/ts/latest/guide/server-communication.html#!)

Example : Asynchronous Calls

To demonstrate how to use an observable, this example shows how to perform an asynchronous call for JSON. To begin, you could start with and add the following file.

**app/test.json**

|  |
| --- |
| [  {  "first": "Jane",  "last": "Chan"  },  {  "first": "Bill",  "last": "Good"  }  ] |

Next, the following code can be added to your application. This code imports a third party library, rxjs, which has been endorsed for managing observables by the AngularJS team. The code in app.mynameservice.ts is basically a template that can easily be modified to enable get, post, put or delete actions with a remote service. For this case, the Observable is set up to retrieve JSON from the test.json file using a get request. Also notice that we are passing a reference through the constructor to the Http class which has access to the get, post, put and delete functions. This reference is passed from the module a little later.

**app/app.mynameservice.ts**

|  |
| --- |
| import { Injectable } from '@angular/core';  import { Http, Response } from '@angular/http';  import { Observable } from 'rxjs/Observable';  import 'rxjs/add/operator/map';  import 'rxjs/add/operator/catch';  @Injectable()  export class MyNameService {  private dataUrl = './app/test.json'; // URL to web API  constructor(private http: Http) { }  getNames(): Observable<string[]> {  return this.http.get(this.dataUrl)  .map(this.extractData)  .catch(this.handleError);  }  private extractData(res: Response) {  let body = res.json();  return body || {};  }  private handleError(error: any) {  // In a real world app, we might use a remote logging infrastructure  // We'd also dig deeper into the error to get a better message  let errMsg = (error.message) ? error.message :  error.status ? `${error.status} - ${error.statusText}` : 'Server error';  console.error(errMsg); // log to console instead  return Observable.throw(errMsg);  }  } |

Next, replace the code inside app.component.ts with the following code. This revised AppComponent creates an instance of the service through it’s constructor. The instance can be created in the constructor header with the help of the providers option. Note too how, since we are using a service that returns an Observable we can set up a subscription with handlers for the following cases:

1. Data is returned.
2. The request experiences an error.
3. A final block is entered regardless of whether 1 or 2 is selected.

**app/app.component.ts**

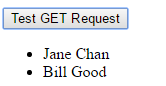
|  |
| --- |
| import { Component } from '@angular/core';  import { MyNameService } from './app.mynameservice';  // This component consumes the re-usable service.  @Component({  selector: 'my-app',  template: `<button (click)="getSomeData()">Test GET Request</button>  <ul>  <li \*ngFor="let myData of myNames">{{myData.first}} {{myData.last}}</li>  </ul>`,  // Providers allow us to inject an object instance through the constroctor.  // In this case we enable injection of MyDataService into AppComponent.  providers: [MyNameService]  })  export class AppComponent {  myNames: Array<any>;  \_myDataService: MyNameService;  // Since we are using a provider above we can receive  // an instance through an instructor.  constructor(myDataService: MyNameService) {  // Store local reference to MyDataService.  this.\_myDataService = myDataService;  }  getSomeData() {  this.\_myDataService.getNames()  // Subscribe to changes in the observable object  // that is returned by getRemoteData.  .subscribe(  // You basically get three handlers.  // 1. Handle successful data.  data => {  this.myNames = data  console.log(JSON.stringify(data))  },  // 2. Handle error.  error => {  alert(error)  },  // 3. Execute final instructions when successful.  () => {  console.log("Finished")  });  }  } |

To enable the HTTP get request we have to reference the HttpModule in our main module.

**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 { HttpModule } from '@angular/http';  @NgModule({  imports: [  BrowserModule,  FormsModule,  HttpModule,  ],  declarations: [  AppComponent,  ],  bootstrap: [AppComponent]  })  export class AppModule { } |

The output from running the program after these changes is as follows:



Exercise

What is the role of the providers option in AppComponent?

|  |
| --- |
| It makes a service available for use by the component. |

Exercise

What event happens if the address to the JSON is invalid? Please explain how this failed event is detected in app.component.ts.

|  |
| --- |
| myDataService.getNames() listens to whether the Observable object returns data or an error message, and implements the appropriate handler for them. Over in app.mynameservice.ts, MyNameService.getNames() catches any errors that might result from the http request and makes the Observable object throw the error if there is one. |

## Lazy Loaded Modules

If you recall from example 8 in day 4 you will remember that we discussed routing. If you check the sources tab of the debugger when running it, you will notice that loading one module loads all components within it. This is not always a bad option but it is never a good idea to load a large application all at once – otherwise the user will have to wait for an unpleasant amount of time and your network traffic could go through the roof. To avoid such heavy page requests components can be grouped within different modules and these modules can be loaded on demand. Loading on demand is also called Lazy Loading.

Example : Lazy Loaded Modules

This example shows how to implement lazy loading so a module which groups different color components is not actually fetched by the browser until the user requests it.

**Part A: Review Code**

In this first section, we will build the basic routing application that was discussed in example 7 of day 3.

This demonstration begins with Example 2. 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: 'my-app',  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 { 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, routing ],  declarations: [ AppComponent, PageDefault,  PageAComponent, PageBComponent ],  bootstrap: [ AppComponent ],  })  export class AppModule { } |

**Part B: New Code**

This example starts with part A to enable lazy loaded modules. The following components are added inside a new ‘colors’ folder.

**app/colors/colors.red.ts**

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

**app/colors/colors.green.ts**

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  template: `This is green.<br/>`  })  export class GreenComponent { } |

**app/colors/colors.component.ts**

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  template: `  <h4>Colors</h4>  <a routerLink="/colors/red" routerLinkActive="active">red</a> |  <a routerLink="/colors/green" routerLinkActive="active">green</a>`  })  export class ColorsComponent { } |

To navigate from one color component to the next we can have a separate router. This routes are stored inside the *colorsRouting* object.

**app/colors/colors.routing.ts**

|  |
| --- |
| import { ModuleWithProviders } from '@angular/core';  import { Routes, RouterModule } from '@angular/router';  import { GreenComponent } from './colors.green';  import { RedComponent } from './colors.red';  import { ColorsComponent } from './colors.component';  const ROUTES = [  { path: '', component: ColorsComponent },  { path: 'green', component: GreenComponent },  { path: 'red', component: RedComponent }  ];  export const colorsRouting: ModuleWithProviders = RouterModule.forChild(ROUTES); |

When we declare our module we import our colorsRouting object and declare all color components that are grouped by the module.

**colors/colors.module.ts**

|  |
| --- |
| import { NgModule, Component } from '@angular/core';  import { CommonModule } from '@angular/common';  import { GreenComponent } from './colors.green';  import { RedComponent } from './colors.red';  import { ColorsComponent } from './colors.component';  import { colorsRouting } from './colors.routing';  @NgModule({  // Import routing for this module and make available for all components.  imports: [  CommonModule,  colorsRouting  ],  // Declare components grouped by this module.  declarations: [  ColorsComponent,  GreenComponent,  RedComponent  ]  })  export class ColorsModule { } |

Now we can modify our parent route by referencing our color module as shown by the code highlighted in yellow. This is teling the application not to load the module until the user clicks on the colors link.

**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';  import { ColorsModule } from './colors/colors.module';  const subroutes: Routes = [  { path: 'page-a', component: PageAComponent },  { path: 'page-b/:id/:firstname', component: PageBComponent },  { path: '', redirectTo: '/page-a', pathMatch: 'full' },  // This says load the ColorsModule which includes color components.  { path: 'colors', loadChildren: 'app/colors/colors.module#ColorsModule' },  { path: '\*\*', component: PageDefault },  ];  export const routing: ModuleWithProviders = RouterModule.forRoot(subroutes); |

To give users access to the colors link it can be added to the AppComponent which is loaded when the application starts up.

**app/app.component.ts**

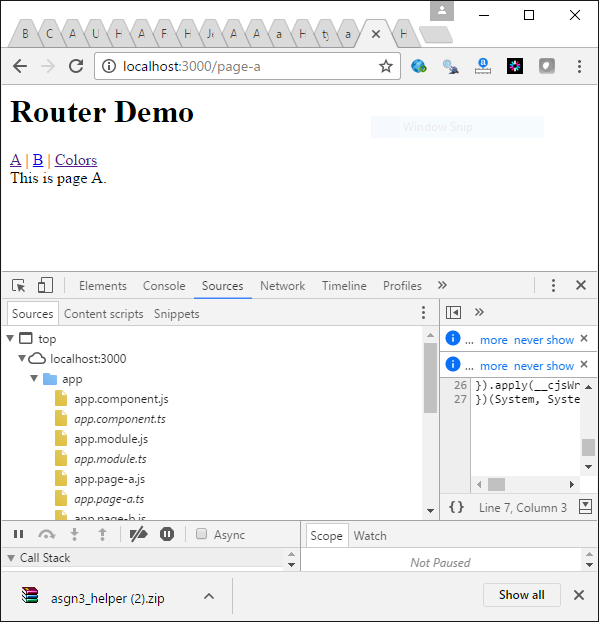
|  |
| --- |
| import { Component } from '@angular/core';  @Component({  selector: 'my-app',  template:  `<h1>Router Demo</h1>  <nav>  <a routerLink="/page-a" routerLinkActive="active">A</a> |  <a routerLink="/page-b/5/bob" routerLinkActive="active">B</a> |  <!-- Link to ColorsModule area -->  <a routerLink="/colors" routerLinkActive="active">Colors</a>  </nav>  <!-- Where router should display a view -->  <router-outlet></router-outlet>`  })  export class AppComponent { } |

Finally, check to ensure the index.html page already contains a base href tag define the application root for the router. This was already done in the example from last week.

**index.html**

|  |
| --- |
| <html>  <head>  <title>Angular QuickStart</title>  <base href="/">  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1">  <!-- 1. Load libraries -->  <!-- Polyfill for older browsers -->  <script src="node\_modules/core-js/client/shim.min.js"></script>  <script src="node\_modules/zone.js/dist/zone.js"></script>  <script src="node\_modules/reflect-metadata/Reflect.js"></script>  <script src="node\_modules/systemjs/dist/system.src.js"></script>  <!-- 2. Configure SystemJS -->  <script src="systemjs.config.js"></script>  <script>  System.import('app').catch(function(err){ console.error(err); });  </script>  </head>  <!-- 3. Display the application -->  <body>  <my-app>Loading...</my-app>  </body>  </html> |

If you run this app and refresh the browser you will notice that the colors folder does not show up in the debugger. Show a screenshot of the ‘Sources’ tab



Exercise

If you click on the colors link you the colors folder will appear underneath the app folder. Show a screenshot of this.

|  |
| --- |
|  |

Exercise

What does it mean to lazy load modules?

|  |
| --- |
| It means you delay loading modules that are part of the app that hasn’t been used/accessed by the user yet. |