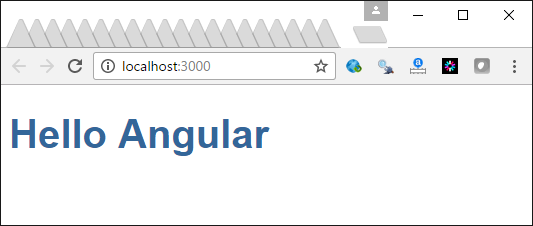
# Inclass 3

* Due 11:59PM June 25
* Please submit all answers in the spaces provided in this word document.
* Worth 8%
* 20% deducted each day this is late.

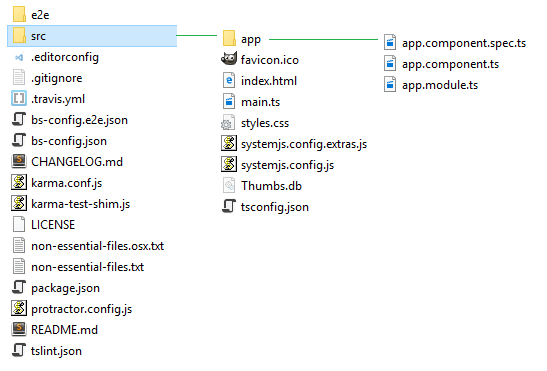
# Angular 4

Example : Angular 4 Hello World!

□ This example shows how to build a simple Angular 4 application. Once you get it running it should like the following:



This is the file structure once you get it set up. This appears to look like a lot of code but actually we won’t use most of the files here in this lesson.



The files are downloaded from the Angular team’s Angular Quickstart tutorial at <https://github.com/angular/quickstart>



You can use git to obtain the files or you can download a zip file containing all of the files and extract it into your work directory.

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/>

The package.json file inside your root folder tells the Node package manager which packages to install.

**package.json**

|  |
| --- |
| {  "name": "angular-quickstart",  "version": "1.0.0",  "description": "QuickStart package.json from the documentation, supplemented with testing support",  "scripts": {  "build": "tsc -p src/",  "build:watch": "tsc -p src/ -w",  "build:e2e": "tsc -p e2e/",  "serve": "lite-server -c=bs-config.json",  "serve:e2e": "lite-server -c=bs-config.e2e.json",  "prestart": "npm run build",  "start": "concurrently \"npm run build:watch\" \"npm run serve\"",  "pree2e": "npm run build:e2e",  "e2e": "concurrently \"npm run serve:e2e\" \"npm run protractor\" --kill-others --success first",  "preprotractor": "webdriver-manager update",  "protractor": "protractor protractor.config.js",  "pretest": "npm run build",  "test": "concurrently \"npm run build:watch\" \"karma start karma.conf.js\"",  "pretest:once": "npm run build",  "test:once": "karma start karma.conf.js --single-run",  "lint": "tslint ./src/\*\*/\*.ts -t verbose"  },  "keywords": [],  "author": "",  "license": "MIT",  "dependencies": {  "@angular/common": "~4.0.0",  "@angular/compiler": "~4.0.0",  "@angular/core": "~4.0.0",  "@angular/forms": "~4.0.0",  "@angular/http": "~4.0.0",  "@angular/platform-browser": "~4.0.0",  "@angular/platform-browser-dynamic": "~4.0.0",  "@angular/router": "~4.0.0",  "angular-in-memory-web-api": "~0.3.0",  "systemjs": "0.19.40",  "core-js": "^2.4.1",  "rxjs": "5.0.1",  "zone.js": "^0.8.4"  },  "devDependencies": {  "concurrently": "^3.2.0",  "lite-server": "^2.2.2",  "typescript": "~2.1.0",  "canonical-path": "0.0.2",  "tslint": "^3.15.1",  "lodash": "^4.16.4",  "jasmine-core": "~2.4.1",  "karma": "^1.3.0",  "karma-chrome-launcher": "^2.0.0",  "karma-cli": "^1.0.1",  "karma-jasmine": "^1.0.2",  "karma-jasmine-html-reporter": "^0.2.2",  "protractor": "~4.0.14",  "rimraf": "^2.5.4",  "@types/node": "^6.0.46",  "@types/jasmine": "2.5.36"  },  "repository": {}  } |

The systemjs.config.js file creates a map to find the necessary packages and root application module. This file is referenced later by the index.html file.

**systemjs.config.js**

|  |
| --- |
| /\*\*  \* System configuration for Angular samples  \* Adjust as necessary for your application needs.  \*/  (function (global) {  System.config({  paths: {  // paths serve as alias  'npm:': 'node\_modules/'  },  // map tells the System loader where to look for things  map: {  // our app is within the app folder  'app': 'app',  // angular bundles  '@angular/core': 'npm:@angular/core/bundles/core.umd.js',  '@angular/common': 'npm:@angular/common/bundles/common.umd.js',  '@angular/compiler': 'npm:@angular/compiler/bundles/compiler.umd.js',  '@angular/platform-browser':  'npm:@angular/platform-browser/bundles/platform-browser.umd.js',  '@angular/platform-browser-dynamic':  'npm:@angular/platform-browser-dynamic/bundles/platform-browser-dynamic.umd.js',  '@angular/http': 'npm:@angular/http/bundles/http.umd.js',  '@angular/router': 'npm:@angular/router/bundles/router.umd.js',  '@angular/forms': 'npm:@angular/forms/bundles/forms.umd.js',  // other libraries  'rxjs': 'npm:rxjs',  'angular-in-memory-web-api':  'npm:angular-in-memory-web-api/bundles/in-memory-web-api.umd.js'  },  // packages tells the System loader how to load when no file or extension  packages: {  app: {  defaultExtension: 'js',  meta: {  './\*.js': {  loader: 'systemjs-angular-loader.js'  }  }  },  rxjs: {  defaultExtension: 'js'  }  }  });  })(this); |

Now that the basic set up for an Angular application is in place it is time to start building our application. To keep our application separate from the rest of the Angular set up, create an app folder inside the project root folder.

For this example, the application’s starting component logic is defined within the *AppComponent* class. This class is defined in an export class so it can be used by other classes. The *AppComponent* class stores data in a *title* property. In the component section, a *template* option stores an inline view. Also, a *selector* option there defines a custom element so the component can be included in any HTML view. Notice here that the *title* property value is accessed within the template option inside double curly braces.

**app/app.component.ts**

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  selector: 'my-app',  template: `<h1>Hello {{name}}</h1>`,  })  export class AppComponent { name = 'Angular'; } |

Angular modules group components and directives into libraries. Every application must have at least one module. In this case, our root module is named *AppModule*. The root module tells the application what to load and how to load it. For this case our starting component, AppComponent, is referenced by this root module.

**app/app.module.ts**

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

The code inside main.ts initializes the root module. Main.ts is referenced by the systemjs.config.js file which is referenced in index.html.

**main.ts**

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

**index.html**

Finally, here is our index.html page which goes in the root folder above the app folder. There are three noteworthy sections in the document:

1. The first section loads the JavaScript libraries.
2. The second section loads the main file in the format that we specified.
3. The **<my-app>** tag loads the contents that we defined in app.component.

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <title>Angular QuickStart</title>  <base href="/">  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1">  <link rel="stylesheet" href="styles.css">  <!-- Polyfill(s) 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/systemjs/dist/system.src.js"></script>  <script src="systemjs.config.js"></script>  <script>  System.import('main.js').catch(function(err){ console.error(err); });  </script>  </head>  <body>  <my-app>Loading AppComponent content here ...</my-app>  </body>  </html> |

Next, open a command prompt as administrator or terminal and navigate to the project root folder. To load the Angular 4 framework and necessary packages, run the command:

**npm install**

Once the packages are loaded, to launch your application in a browser, run the command:

**npm start**

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

Exercise

Which element in index.html references the template which is associated with this selector option inside the @Component annotation inside app.component.ts? (1 mark)

|  |
| --- |
| <my-app></my-app> |

Exercise

How is AppComponent identified as the starting component inside app.module.ts? (1 mark)

|  |
| --- |
| Bootstrap: [ AppComponent] |

Exercise

Add a second property to the AppComponent class to store your last name. Modify the code in the template option to show your last name with the rest of the original content. Show your revised **app.component.ts** file here: (1 mark)

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  selector: 'my-app',  template: `  <h1>Hello {{name}}</h1>  <h2>Last name: {{lastName}}  `,  })  export class AppComponent {  name = 'Angular 4!! You are soo cool';  lastName = 'Chung' } |

## Export Classes

If we plan to create multiple objects of the same type which must implement a uniform set of properties, we can create a class. The keyword *export* while allow us to reference the class from a separate file later.

|  |
| --- |
| export class PlayingCard {  cardVal: string;  suit: string;  } |

## Back-ticks

Back-ticks allow us to create an HTML that spans multiple lines.

|  |
| --- |
| template: `<h1>Hello world! {{title}} <br/>  {{card.cardVal}} of {{card.suit}}.</h1>` |

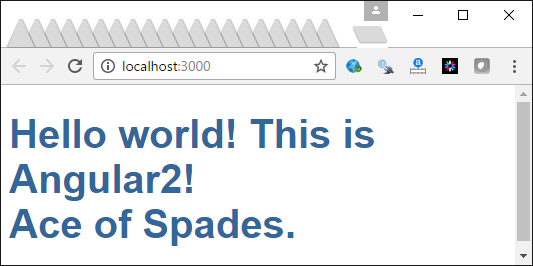
Example : Classes and Back Ticks

This example continues from Example 1 to implement an export class and a multi-line template. In this case the *card* object is created with the *PlayingCard* class and then it is displayed. The export keyword will allow us to reference the class in a separate file later. Back-ticks allow us to write our HTML tags on multiple lines. To build this example, start with Example 1 and replace the contents of the *app.component.ts* file.

**app/app.component.ts**

|  |
| --- |
| import { Component } from '@angular/core';  export class PlayingCard {  cardVal: string;  suit: string;  }  @Component({  selector: 'my-app',  // Multi-line content is allowed with back ticks.  template: `<h1>Hello world! {{title}} <br/>  {{card.cardVal}} of {{card.suit}}.</h1>`  })  export class AppComponent {  public title = 'This is Angular 4!';  // Declare a PlayingCard object.  public card: PlayingCard = {  cardVal: "Ace",  suit: "Spades"  };  } |

The output from this example becomes:



Exercise

Add a second card object to the *AppComponent* class and store the Queen of Diamonds. Then modify the template to show the new card and suit in addition to the ace of spades. Show your revised app.component.ts file here: (1 mark)

|  |
| --- |
| import { Component } from '@angular/core';  export class PlayingCard {  cardVal: string;  suit: string;  }  @Component({  selector: 'my-app',  // Multi-line content is allowed with back ticks.  template: `<h1>Hello world! {{title}} <br/>     {{card.cardVal}} of {{card.suit}}.</h1>`  })  export class AppComponent {  public title = 'This is Angular 4!';  // Declare a PlayingCard object.  public card: PlayingCard = {  cardVal: "Ace",  suit: "Spades"  };  public card2: PlayingCard = {  cardVal: "Queen",  suit: "Spades"  };  }  f= |

## Two Way Data Binding [(ngModel)]

So far we have only shown how to display the properties of a component. We can allow these property values to be modified from the HTML though too with the ngModel directive in combination with HTML inputs similar to the one shown below.

|  |
| --- |
| Card: <input [(ngModel)]="card.cardVal"> |

Example : Two Way Data Binding

To build this example, start with Example 2. Then, we need to load FormsModule into app.module.ts. The module file is like a package or namespace that groups classes together.

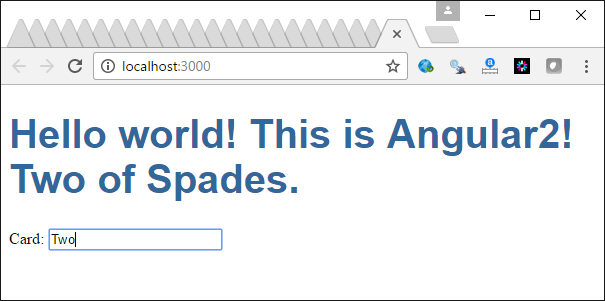
|  |
| --- |
| import { NgModule } from '@angular/core';  import { BrowserModule } from '@angular/platform-browser';  import { FormsModule } from '@angular/forms';  import { AppComponent } from './app.component';  @NgModule({  imports: [BrowserModule, FormsModule],  declarations: [AppComponent],  bootstrap: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

Next replace the app.component.ts file with this new version of the code:

**app/app.component.ts**

|  |
| --- |
| import { Component } from '@angular/core';  export class PlayingCard {  cardVal: string;  suit: string;  }  @Component({  selector: 'my-app',  // Multi-line content allowed with back ticks.  template: `<h1>Hello world! {{title}} <br/>  {{card.cardVal}} of {{card.suit}}.</h1>  Card: <input [(ngModel)]="card.cardVal">`  })  export class AppComponent {  public title = 'This is Angular 4!';  // Declare a PlayingCard object.  public card: PlayingCard = {  cardVal: "Ace",  suit: "Spades"  };  } |

The new version of the program will show the card value and suit along with an input that allows you to change the *cardVal* property.



Exercise

Add another input which allows the user to input the suit of the *card* object. Show your revised app.component.ts file here: (1 mark)

|  |
| --- |
| import { Component } from '@angular/core';  export class PlayingCard {  cardVal: string;  suit: string;  }  @Component({  selector: 'my-app',  // Multi-line content allowed with back ticks.  template: `<h1>Hello world! {{title}} <br/>     {{card.cardVal}} of {{card.suit}}.</h1>  Card: <input [(ngModel)]="card.cardVal">  Suit: <input [(ngModel)]="card.suit">`  })  export class AppComponent {  public title = 'This is Angular 4!';  // Declare a PlayingCard object.  public card: PlayingCard = {  cardVal: "Ace",  suit: "Spades"  };  } |

## Looping (\*ngFor)

You can loop with \*ngFor to loop through a collection of objects. Each iteration a named object is created.

<ul><li \*ngFor="let card of cards">**{{**card.cardVal**}}**</li></ul>

Example : Looping through JSON

□ This example creates a list item tag for every object inside an array of cards. The card collection is defined in the JSON at the bottom of app.component.ts. The collection is assigned to the cards property inside the *AppComponent* class.

|  |
| --- |
| import { Component } from '@angular/core';  export class PlayingCard {  cardVal: string;  suit: string;  }  @Component({  selector: 'my-app',  // Multi-line content allowed with back ticks.  template: `<h1>Hello world! {{title}} <br/>  <!-- Show cards in unordered list. -->  <ul><li \*ngFor="let card of cards">{{card.cardVal}}</li></ul>`  })  export class AppComponent {  public title = 'This is Angular 4!';  // Include card data in collection as public property.  public cards = CARDS;  }  // Define card data.  var CARDS: PlayingCard[] = [  { cardVal: "Ace", suit: "Spades" },  { cardVal: "Two", suit: "Clubs" },  { cardVal: "Six", suit: "Hearts" },  ]; |

Here is the output.



Exercise

Replace the code that builds an unordered list with *ngFor* to iterate through all cards to display the card value and suit within a table by using <tr> and <td> tags along with a <table> tag. Place the card value and suit in separate columns. (2 marks)

|  |
| --- |
| import { Component } from '@angular/core';  export class PlayingCard {  cardVal: string;  suit: string;  }  @Component({  selector: 'my-app',  // Multi-line content allowed with back ticks.  template: `<h1>Hello world! {{title}}</h1>  <!-- Show cards in unordered list. -->  <table>  <tr>  <th>Value</th>  <th>Suit</th>  </tr>  <tr \*ngFor="let card of cards">  <td>{{card.cardVal}}</td>  <td>{{card.suit}}</td>  </tr>  </table>`  })  export class AppComponent {  public title = 'This is Angular 4!';  // Include card data in collection as public property.  public cards = CARDS;  }  // Define card data.  var CARDS: PlayingCard[] = [  { cardVal: "Ace", suit: "Spades" },  { cardVal: "Two", suit: "Clubs" },  { cardVal: "Six", suit: "Hearts" },  ]; |

## Click Handling (click)

Clicks can be managed with the (click) handler. For example:

<li \*ngFor="let card of cards" (click)="onSelect(card)">

You can use the expression that follows the (click) handler to call a function that is defined in the component for additional processing. In the line of code above the onSelect() function is called.

## Showing and Hiding (\*ngIf)

The \*ngIf directive can be used to evaluate expressions in addition to showing and hiding content:

|  |
| --- |
| <!-- Show is selectedCard is defined. -->  <div \*ngIf="selectedCard">  <h2>**{{**selectedCard.cardVal**}}** \*\*</h2>  <input [(ngModel)]="selectedCard.cardVal" placeholder="name" />  </div> |

Example : Clicks and Showing and Hiding

This example builds on Example 4 to show a enable click handling as well as dynamic swapping of a tag based on a condition that is defined in an ngIf directive.

**app/app.component.ts**

|  |
| --- |
| import { Component } from '@angular/core';  export class PlayingCard {  cardVal: string;  suit: string;  }  @Component({  selector: 'my-app',  // Multi-line content allowed with back ticks.  template: `<h1>Hello world! {{title}} <br/>  <!-- Show cards in unordered list. -->  <ul><li \*ngFor="let card of cards" (click)="onSelect(card)">  {{card.cardVal}}</li>  </ul>  <div \*ngIf="selectedCard">  <h2>{{selectedCard.cardVal}} \*\*</h2>  <input [(ngModel)]="selectedCard.cardVal" placeholder="name"/>  </div>  `  })  export class AppComponent {  public title = 'This is Angular 4!';  // Include card data in class as public property.  public cards = CARDS;  selectedCard: PlayingCard;    onSelect(card: PlayingCard) {  this.selectedCard = card;  }  }  // Define card data.  var CARDS: PlayingCard[] = [  { cardVal:"Ace", suit:"Spades"},  { cardVal:"Two", suit:"Clubs" },  { cardVal:"Six", suit:"Hearts"},  ]; |

This output shows the list of cards and also allows you to modify the value of any that is selected.



Exercise

Modify the contents that appear when a list item is selected so if the new content is clicked an alert box appears. Show your revised app.component.ts file:

|  |
| --- |
| import { Component } from '@angular/core';  export class PlayingCard {  cardVal: string;  suit: string;  }  @Component({  selector: 'my-app',  // Multi-line content allowed with back ticks.  template: `<h1>Hello world! {{title}} <br/>  <!-- Show cards in unordered list. -->  <ul><li \*ngFor="let card of cards" (click)="onSelect(card)">  {{card.cardVal}}</li>  </ul>  <div \*ngIf="selectedCard">  <h2>{{selectedCard.cardVal}} \*\*</h2>  <input [(ngModel)]="selectedCard.cardVal" placeholder="name"/>  </div>         `  })  export class AppComponent {  public title = 'This is Angular 4!';  // Include card data in class as public property.  public cards = CARDS;  selectedCard: PlayingCard;    onSelect(card: PlayingCard) {  this.selectedCard = card;  alert("You selected: " + this.selectedCard.cardVal + " of " + this.selectedCard.suit);  }  }  // Define card data.  var CARDS: PlayingCard[] = [  { cardVal:"Ace", suit:"Spades"},  { cardVal:"Two", suit:"Clubs" },  { cardVal:"Six", suit:"Hearts"},  ]; |

## Styling

Styling can be set with the class annotation inside the *template* option of the component. Styles can also be set within the component annotation through a *styles* option.

|  |
| --- |
| <ul><li \*ngFor="let card of cards" (click)="onSelect(card)"  [class.selected]="card === selectedCard">{{card.cardVal}}</li></ul>  <div \*ngIf="selectedCard">  <input [(ngModel)]="selectedCard.cardVal" placeholder="name"/>  </div>`,  styles:[`  .selected {  background-color: #CFD8DC;  color: green;  }`] |

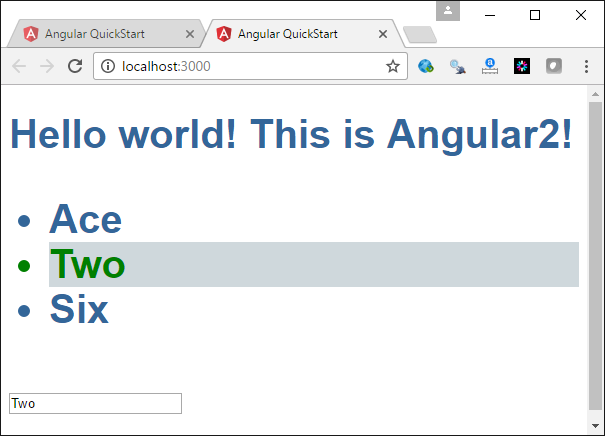
Of course the styles can be referenced from a separate css file.

Example : Adding Styles

This example shows how to modify the app.component.ts file so apply styling to a selected item in the last of cards. To build this example, replace the app.component.ts file from Example 5 with the following:

|  |
| --- |
| import { Component } from '@angular/core';  export class PlayingCard {  cardVal: string;  suit: string;  }  @Component({  selector: 'my-app',  // Multi-line content allowed with back ticks.  template: `<h1>Hello world! {{title}} <br/>  <!-- Show cards in unordered list. -->  <ul><li \*ngFor="let card of cards" (click)="onSelect(card)"  [class.selected]="card === selectedCard">{{card.cardVal}}</li></ul>  <div \*ngIf="selectedCard">  <input [(ngModel)]="selectedCard.cardVal" placeholder="name"/>  </div>`,  styles:[`  .selected {  background-color: #CFD8DC;  color: green;  }`]  })  export class AppComponent {  public title = 'This is Angular 4!';  // Include card data in class as public property.  public cards = CARDS;  selectedCard: PlayingCard;    onSelect(card: PlayingCard) {  this.selectedCard = card;  }  }  // Define card data.  var CARDS: PlayingCard[] = [  { cardVal:"Ace", suit:"Spades"},  { cardVal:"Two", suit:"Clubs"},  { cardVal:"Six", suit:"Hearts"},  ]; |

The output shows the styles applied to the item that is selected from the list:



Exercise

Starting with Example 6, add an extra style so unselected items in the list are displayed with red font.

|  |
| --- |
| @Component({  selector: 'my-app',  // Multi-line content allowed with back ticks.  template: `<h1>Hello world! {{title}} <br/>  <!-- Show cards in unordered list. -->  <ul><li \*ngFor="let card of cards" (click)="onSelect(card)"     [class.selected]="card === selectedCard" [class.not-selected]="card !== selectedCard">{{card.cardVal}}</li></ul>  <div \*ngIf="selectedCard">  <input [(ngModel)]="selectedCard.cardVal" placeholder="name"/>  </div>`,  styles:[`  .selected {  background-color: #CFD8DC;  color: green;  }  .not-selected {  color: red;  }`]  }) |

## Basic Routing in Angular 4

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 : 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: '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 { 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 { } |

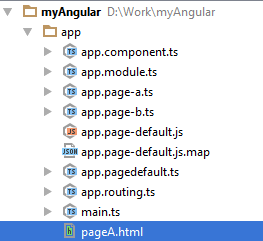
To specify a default target and reference for all tags in the Angular application, include this base tag inside the header of the index.html file.

**index.html**

|  |
| --- |
| <base href="/"> |

Example : TemplateUrl

This example shows how to separate your HTML into different pages. To do this, begin with Example 7, 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 : Enabling Route Parameters

This example shows how to enable route parameters. Start with the solution from . 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**

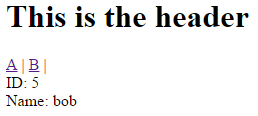
|  |
| --- |
| <a routerLink="/page-b/5/bob" routerLinkActive="active">B</a> |

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

How are the constructor and ngInit functions different?

|  |
| --- |
| The constructor sets up the class, and the ngOnInit is called after the class is set up so that whatever was passed in through the constructor is available to the rest of the class. |

Exercise

Add the following constant declaration to the constructor in Example 9.

|  |
| --- |
| // 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

Starting with Example 9, 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?

|  |
| --- |
| ngOnInit() |

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.

|  |
| --- |
|  |

## 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 { bootstrap } from '@angular/platform-browser-dynamic';  import { AppComponent } from './app.component'  import { enableProdMode } from '@angular/core';  enableProdMode();  bootstrap(AppComponent); |

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