# Inclass 1

* Due November 8, 9AM
* Please submit all answers in the spaces provided in this word document.
* Worth 8%
* 20% deducted each day this is late.

# Starting Angular

This section introduces essential structures of Angular to help you quickly ramp up with this technology for professional, data-driven development.

## Advantages of the Angular Framework

Angular is one of the leading client-side JavaScript frameworks today for interfacing with back-end server applications from a web browser. Angular is currently popular because it offers:

* Excellent separation of presentation and logic for data driven applications.
* Excellent support for single page application development.
* Convenient two-way data binding and validation.
* Support by Google and a large global developer community.
* Excellent browser compatibility.

## Terminology Introduction

To help start this discussion, here are some general definitions:

### Template / View

The template contains HTML elements plus directives and expressions to present content.

### Components

A component is a class which encapsulates data and logic to support a view. It is similar to a controller. However, the component class is preceded by a decorator which contains metadata about how the class is to be instantiated and used.

### Directives

A directive invokes custom behavior for HTML elements. Angular provides many pre-defined directives to speed up development. Angular also allows the development of custom directives.

### Models

Models are named references that store data. They are declared as variables or properties of a class. They can then be referenced in the template so their data can be presented or obtained through the HTML interface that is rendered.

### Router

A router receives page requests and selects the appropriate component.

### Two Way Data Binding

Two Way data binding refers to the process of simultaneous “data model updates” between references to it in the view and component. If a model value in the view changes, its value in the component is updated and vice versa.

### Services

A service is encapsulated logic for a specific task. The service is accessible to all components. Angular has many pre-defined services like an HTTP service to manage get, post, put, and delete operations. You can of course create your own services.

### Module

A module is a library of component classes. The module also contains metadata to describe how to load the components and how to provide services to the components such as routing.

## TypeScript

TypeScript is an open source programming language that is maintained by Microsoft. TypeScript is built on top of JavaScript so you can actually run JavaScript code within it. TypeScript allows for a better development experience by enabling stronger typing and object-oriented modularization in the development environment. TypeScript must be transpiled to JavaScript to run in a browser.

Angular and many current leading JavaScript based frameworks have adopted TypeScript as the main development language.

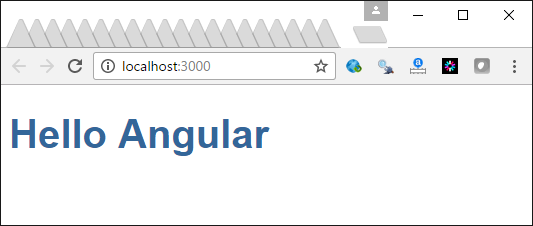
## Set Up

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:

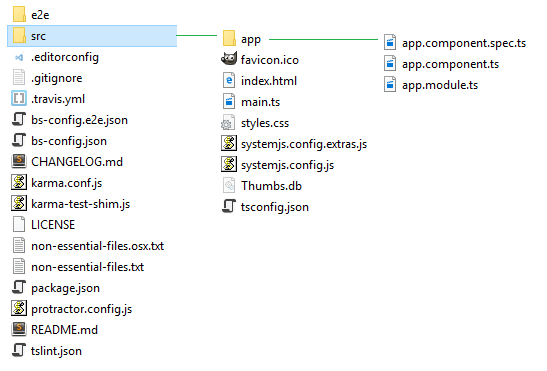
<https://nodejs.org/en/download/>

Example 1: Angular 4 Hello World!

□ This example shows how to build a simple Angular 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 latest files for this Angular project can be downloaded from the Angular team’s Angular Quickstart tutorial at <https://github.com/angular/quickstart>

Angular’s quickstart repository is updated whenever Angular is updated so you can always get the latest version when needed.

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.



### Deleting Non-Essential Files (Optional)

Once you extract the contents of the Git repository you will notice a bunch of files at the root level. Several of these files serve no purpose other than enabling Git management for the original repository. You can quickly delete these *non-essential* files by entering the following commands while in the project folder:

##### OS/X (bash)

xargs rm -rf < non-essential-files.osx.txt

rm src/app/\*.spec\*.ts

rm non-essential-files.osx.txt

##### Windows

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

rd .git /s /q

rd e2e /s /q

### Initial Configuration and Html Files

Several configuration files exist in the project files to assist supporting your Angular application. When starting out you can generally ignore these files but it helps to understand their role.

#### package.json

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

* a *scripts* section defines node commands that you can execute to initiate tasks for your project.
* a *dependencies* section to indicates which packages need to be downloaded for use in your Angular application.
* a *devDependencies* section references packages for utilities to perform a range of tasks such as for running a lite server and for testing.

**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\"",  …  "lint": "tslint ./src/\*\*/\*.ts -t verbose"  },  "keywords": [],  "author": "",  "license": "MIT",  "dependencies": {  "@angular/common": "~4.3.4",  "@angular/compiler": "~4.3.4",  "@angular/core": "~4.3.4",  …  "rxjs": "5.0.1",  "zone.js": "^0.8.4"  },  "devDependencies": {  "concurrently": "^3.2.0",  "lite-server": "^2.2.2",  "typescript": "~2.1.0",  "jasmine-core": "~2.4.1",  "karma": "^1.3.0",  …  "karma-chrome-launcher": "^2.0.0",  "karma-cli": "^1.0.1",  "@types/node": "^6.0.46",  "@types/jasmine": "2.5.36"  },  "repository": {}  } |

#### systemjs.config.js

The systemjs.config.js file creates a map to load the necessary packages at start up. This file is referenced later by the index.html file. Once again you do not need to be concerned with the contents of this file. It is better to just use the systemjs.config.js file that is provided in the repository where you obtain your git starter files.

**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 filename and/or no extension  packages: {  app: {  defaultExtension: 'js',  meta: {  './\*.js': {  loader: 'systemjs-angular-loader.js'  }  }  },  rxjs: {  defaultExtension: 'js'  }  }  });  })(this); |

#### main.ts

The bootstrapModule() function inside main.ts loads the root module of our application. Our application begins in AppModule.

**main.ts**

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

#### Index.html

**index.html**

You can find the index.html page in the root folder above the app folder. There are three noteworthy sections in the document:

1. The first section loads the JavaScript libraries. For example, it loads the system.config.js file which in turn loads up all of the necessary Angular modules.
2. The second section loads the main file which references our actual Angular application.
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> |

### Application Files

Now that the basic set up for an Angular application is in place it is time to start building our application.

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 property named *name*. In the decorator area, 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 { } |

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 1

Given the following diagram:



Match the letters one of the following terms

1. Service 2. Model 3. Decorator 4. Defines name of HTML element.
2. \_\_\_\_3\_\_\_\_\_
3. \_\_\_\_\_2\_\_\_\_
4. \_\_\_\_4\_\_\_\_\_

Exercise 2

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

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

Exercise 3

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)

|  |
| --- |
| //like 'using' in C#  //import includes libraries.  import { Component } from '@angular/core';  //Component explains how the class will be processed .It must preceed the class definition.  //A combination loike this is called:  //'Component directive'  @Component({ // decorator  //metadata -  selector: 'my-app',  template: `<h1>Hello {{name}}<br>  My last name is {{myLastName}}</h1>`,  })  //class definition  export class AppComponent { name = 'Angular wedeaweawe';  myLastName='Xu'; } |

## Debugging Tips

Hopefully you are all going to keep your debugger window open at all times when writing any sort of JavaScript. I have put together this video highlighting how to debug your application.

<https://www.youtube.com/watch?v=IDijSikrRZ0>

Exercise 4

Inside the app folder, create a file named app.html. Then place this content inside the file:

<h1>Hello {{name}}</h1>

Next, replace this line:

template: `<h1>Hello {{name}}</h1>`,

With this line:

templateUrl: './app.html'

1. Save your files and run your project. Next using the debugger, set a breakpoint beside the area where the name model is defined inside app.component.ts. Show a screenshot with the Chrome debugger Sources tab while open at the app.component.ts file and while halted at the break point that you set. Show the screenshot here.

|  |
| --- |
|  |

1. Resume execution of your application so the page renders in the browser. Show a screenshot of the debugger with all of the code for app.component.ts visible in the debugger and with the contents of the web page also fully visible. Please resize your browser to capture and present this nicely with a size that fits well in this page. Show a screenshot of this here:

|  |
| --- |
|  |

## 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>` |

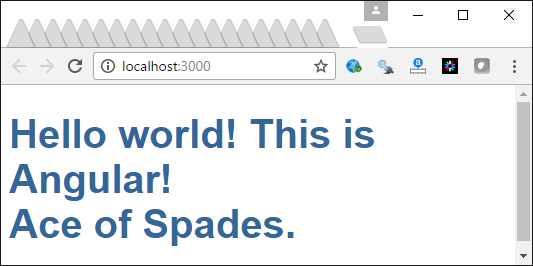
Example 2: Classes and Back Ticks

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!';  // Declare and initialize a PlayingCard object.  public card: PlayingCard = {  cardVal: "Ace",  suit: "Spades"  };  } |

The output from this example becomes:



Exercise 5

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}}.<br>  {{card2.cardVal}} of {{card2.suit}}.<br></h1>`  })  export class AppComponent {  public title = 'This is Angular!';  // Declare and initialize a PlayingCard object.  public card: PlayingCard = {  cardVal: "Ace",  suit: "Spades"  };  public card2:PlayingCard ={  cardVal:"Queen",  suit:"Diamonds"  }  } |

## Two Way Data Binding [(ngModel)] Directive

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 3: 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.

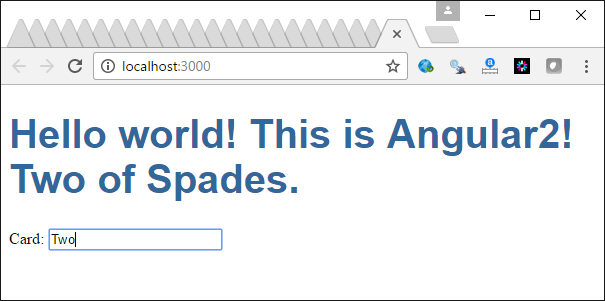
|  |
| --- |
| import { NgModule } from '@angular/core';  import { BrowserModule } from '@angular/platform-browser';  import { FormsModule } from '@angular/forms';  import { AppComponent } from './app.component';  @NgModule({  imports: [BrowserModule, FormsModule],  declarations: [AppComponent],  bootstrap: [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 6

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">  Card: <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) Directive

You can loop with \*ngFor to loop through a collection of objects.

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

Example 4: 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 7

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}} <br/>  <!-- 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) Directive

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 5: 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 8

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:

|  |
| --- |
| onSelect(card: PlayingCard) {  this.selectedCard = card;  alert("hi");  }  } |

## Validation

Form validation is automated in Angular 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 5: 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 Example 2.

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 **app.component.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 9

Add a new control which allows the input of a last name value. This field is required. Only alphabetical characters are allowed. The minimum length is two letters. Appropriate messages appear to inform the user how to validate this field. Show your revised version of app.component.ts after making these changes.

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  selector: 'my-app',  template:  `  <section>  <form (ngSubmit)="onSubmit()" #myForm="ngForm">  firstName:  <input type="text" pattern="[a-zA-Z ]\*" minlength="3" required  [(ngModel)]="first" 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>  <br>  lastName :  <input type="text" pattern="[a-zA-Z ]\*" minlength="2" required  [(ngModel)]="last" name="lastName" #lastName="ngModel" >  <div [hidden]="lastName.valid || lastName.pristine">  This control is invalid.</div>  <p \*ngIf="lastName?.errors?.required">This field is required.</p>  <p \*ngIf="lastName?.errors?.pattern">  Only alphabetical characters are allowed.</p>  <p \*ngIf="lastName?.errors?.minlength">  This entry must have at least two characters.</p>  <br>  <button type="submit" class="btn btn-default"  [disabled]="!myForm.form.valid">Submit</button>  </form>  </section>    `  })  export class AppComponent {  first: string;  last: string;  constructor() {  this.first = "Karl";  this.last ="Xu";  }  } |

Exercise 10

To review some of the terminology, match the following terms with the definitions provided. Use each term only once.

template (view), Router, Model, Service, Directive, Decorator

1. \_\_\_\_\_\_\_\_\_\_\_\_ Model \_\_\_\_\_\_\_\_\_ defines a data reference.

2. \_\_\_\_\_\_\_\_\_\_ Router \_\_\_\_\_\_\_\_\_\_\_ selects templates (views) and components.

3. \_\_\_\_\_\_\_\_\_\_\_\_ Directive \_\_\_\_\_\_\_\_\_ modifies behavior of HTML.

4. \_\_\_\_\_\_\_\_\_\_\_ template (view)\_\_\_\_\_\_\_\_\_\_ stores HTML and presents content.

5. \_\_\_\_\_\_\_\_\_\_ Decorator \_\_\_\_\_\_\_\_\_\_\_ contains metadata which describes how the component class is processed.

6. \_\_\_\_\_\_\_\_\_\_\_ Service \_\_\_\_\_\_\_\_\_\_ encapsulates logic which can be used by many components.

Exercise 11

Please answer true or false to the following questions:

1. \_\_T\_\_\_ The package.json file refences dependences that are to be downloaded by the node package manager.
2. \_\_\_T\_\_ The package.json file defines commands that are executed by the node package manager.
3. \_\_\_T\_\_ main.ts sets the starting module for the application.
4. \_\_\_T\_\_ TypeScript has *number* and *string* datatypes among others.
5. \_\_F\_\_\_ TypeScript must be transpiled to JavaScript to run in a browser.
6. \_\_\_T\_\_ The selector tag defines an element name which can identify the html tag needed to instantiate and render a component.