# Inclass 1

* Due (to be announced)
* 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/>

### Angular CLI

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

The latest version with good documentation can be found here:

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

### Preparing and Installing Angular CLI

Git may be required first before you perform the install. If you do not have it yet you can get it at:

<https://git-scm.com/download/>

After the installing Git, close terminal or the command prompt and restart it before following the next set of instructions.

Then install angular-cli with the following command:

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

## Creating and Running an Application

Example 1: Creating and Running an Application

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

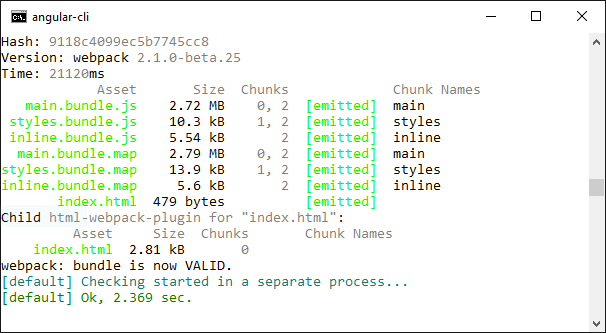
**>ng new myapp**

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

>**cd myapp**

**myapp>ng serve**

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



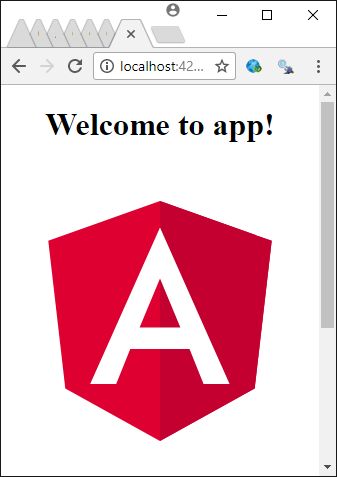
To view your application, navigate to the following address in your browser:

<http://localhost:4200/>

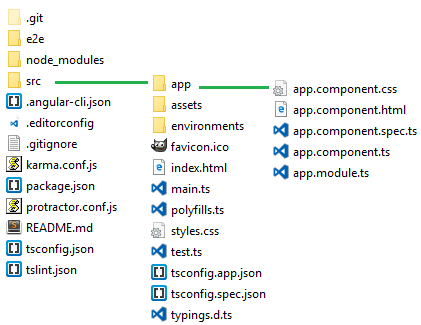
Once the browser launches, the application will stay open and respond to changes as you update the code.

Example 1: Angular 4 Hello World!

□ Follow the examples above. When finished the output should appear as follows:



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.



### Important Angular Application 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": "myapp",  "version": "0.0.0",  "license": "MIT",  "scripts": {  "ng": "ng",  "start": "ng serve",  "build": "ng build",  "test": "ng test",  "lint": "ng lint",  "e2e": "ng e2e"  },  "private": true,  "dependencies": {  "@angular/animations": "^4.2.4",  "@angular/common": "^4.2.4",  "@angular/compiler": "^4.2.4",  "@angular/core": "^4.2.4",  "@angular/forms": "^4.2.4",  "@angular/http": "^4.2.4",  "@angular/platform-browser": "^4.2.4",  "@angular/platform-browser-dynamic": "^4.2.4",  "@angular/router": "^4.2.4",  "core-js": "^2.4.1",  "rxjs": "^5.4.2",  "zone.js": "^0.8.14"  },  "devDependencies": {  "@angular/cli": "1.4.7",  "@angular/compiler-cli": "^4.2.4",  "@angular/language-service": "^4.2.4",  "@types/jasmine": "~2.5.53",  "@types/jasminewd2": "~2.0.2",  "@types/node": "~6.0.60",  "codelyzer": "~3.2.0",  "jasmine-core": "~2.6.2",  "jasmine-spec-reporter": "~4.1.0",  "karma": "~1.7.0",  "karma-chrome-launcher": "~2.1.1",  "karma-cli": "~1.0.1",  "karma-coverage-istanbul-reporter": "^1.2.1",  "karma-jasmine": "~1.1.0",  "karma-jasmine-html-reporter": "^0.2.2",  "protractor": "~5.1.2",  "ts-node": "~3.2.0",  "tslint": "~5.7.0",  "typescript": "~2.3.3"  }  } |

#### src/main.ts

main.ts compiles the application and loads the main module to run in the browser. You really do need to modify main.ts and generally can stay away from it.

**src/main.ts**

|  |
| --- |
| import { enableProdMode } from '@angular/core';  import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';  import { AppModule } from './app/app.module';  import { environment } from './environments/environment';  if (environment.production) {  enableProdMode();  }  // bootstrapModule loads the starting module (library).  platformBrowserDynamic().bootstrapModule(AppModule)  .catch(err => console.log(err)); |

#### src/app/app.component.ts

In the default Angular CLI build, the application’s root component is defined within *AppComponent*.

In the @Component decorator area, a *templateUrl* option which references an HTML view. Also, a *selector* option there defines a custom element so the component can be included in any HTML view.

In the class that follows, data and functions can be defined to support the component. For this case, the *AppComponent* class stores a property (called a model) named *title*. This class is defined in an export class so it can be imported into other classes.

**src/app/app.component.ts**

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  selector: 'app-root',  templateUrl: './app.component.html',  styleUrls: [ './app.component.css']  })  export class AppComponent {  title = 'app';  } |

#### src/app/app.component.html

app.component.html contains the html for the partial view. Notice that the value of title can be displayed with the html when it is included in double curly braces inside the HTML.

**src/app/app.component.html**

|  |
| --- |
| <h1>  Welcome to **{{**title**}}**!  </h1> |

#### src/app/app.modules.ts

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.

**src/app/app.module.ts**

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

#### src/index.html

The index.html file contains the parent view for our single page application. Notice that the selector for AppComponent is referenced below.

|  |
| --- |
| <!doctype html>  <html lang="en">  <head>  <meta charset="utf-8">  <title>Myapp</title>  <base href="/">  <meta name="viewport" content="width=device-width, initial-scale=1">  <link rel="icon" type="image/x-icon" href="favicon.ico">  </head>  <body>  <app-root></app-root>  </body>  </html> |

When looking at the file, you may notice that absence of JavaScript. JavaScript references are inserted into the application when it is transpiled before it runs in the browser.



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. \_Decorator\_\_\_\_\_\_\_\_
3. \_\_Model\_\_\_\_\_\_\_
4. \_\_\_\_\_\_\_\_\_

Exercise 2

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

|  |
| --- |
| Bootstrap root 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)

|  |
| --- |
|  |

## Debugging Tips

Hopefully you are all going to keep your debugger window open at all times when working with JavaScript or TypeScript. I have put together this video highlighting how to debug your application.

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

Exercise 4

Inside app.component.html replace the content of this file with:

|  |
| --- |
| <h1>  Welcome to **{{**title**}}**!  </h1> |

1. Save your changes and run your project. Next using the debugger, set a breakpoint beside the area where the title 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 its elements tab visible so it shows the message *Welcome to app* nested inside the HTML. 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>` |

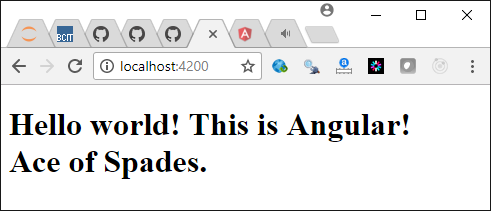
Example 2: Classes and Back Ticks

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: 'app-root',  // 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)

|  |
| --- |
|  |

## 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.

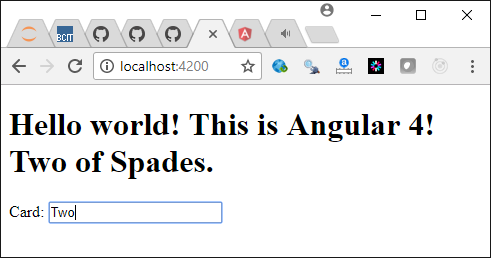
|  |
| --- |
| import { NgModule } from '@angular/core';  import { BrowserModule } from '@angular/platform-browser';  import { FormsModule } from '@angular/forms';  import { AppComponent } from './app.component';  @NgModule({  imports: [BrowserModule, FormsModule],  declarations: [AppComponent],  bootstrap: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

|  |
| --- |
|  |

## 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: 'app-root',  // 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)

|  |
| --- |
|  |

## 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 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: 'app-root',  // 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. The only task you need to complete for this exercise is to place this line of code inside the correct section of the app.component.ts file:

alert(card.suit);

Show your revised app.component.ts file:

|  |
| --- |
|  |

## 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: 'app-root',  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 second 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.

|  |
| --- |
|  |

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. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ defines a data reference.

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

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

4. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ stores HTML and presents content.

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

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

Exercise 11

Please answer true or false to the following questions:

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