

Bootstrapping Angular

An NgModule describes how the application parts fit together. Every application has at least one Angular module, the *root* module that you bootstrap to launch the application. By convention, it is usually called AppModule.

Angular needs to know how the pieces of your application fit together and what other files and libraries the app requires. This information is called metadata

Some of the metadata is in the @Component decorators that you added to your component classes. Other critical metadata is in @NgModule decorators.

The most important @NgModule decorator annotates the top-level AppModule class.

The Angular CLI generated an AppModule class in src/app/app.module.ts when it created the project. This is where you opt-in to the FormsModule.

Bootstrapping Angular

If you use the Angular CLI to generate an app, the default AppModule is as follows:

```
import { BrowserModule } from '@angular/platform-browser';  
import { NgModule } from '@angular/core';  
import { FormsModule } from '@angular/forms';  
import { HttpClientModule } from '@angular/common/http';  
import { AppComponent } from './app.component';
```

/* the AppModule class with the [@NgModule](#) decorator */

```
@NgModule({  
  declarations: [  
    AppComponent  
  ],  
  imports: [  
    BrowserModule,  
    FormsModule,  
    HttpClientModule  
  ],  
  providers: [],  
  bootstrap: [AppComponent]  
})  
export class AppModule { }
```

Bootstrapping Angular

After the import statements is a class with the `@NgModule decorator`.

- The `@NgModule` decorator identifies AppModule as an `NgModule` class. `@NgModule` takes a metadata object that tells Angular how to compile and launch the application.
- *declarations*—this application's lone component.
- *imports*—import `BrowserModule` to have browser specific services such as DOM rendering, sanitization, and location.
- *providers*—the service providers.
- *bootstrap*—the *root* component that Angular creates and inserts into the index.html host web page.
- The default application created by the Angular CLI only has one component, AppComponent, so it is in both the `declarations` and the `bootstrap` arrays.

Bootstrapping Angular

- The [declarations](#) array
- The module's [declarations](#) array tells Angular which components belong to that module. As you create more components, add them to [declarations](#).
- You must declare every component in exactly one [NgModule](#) class. If you use a component without declaring it, Angular returns an error message.
- The [declarations](#) array only takes declarables. Declarables are components, [directives](#) and [pipes](#). All of a module's declarables must be in the [declarations](#) array. Declarables must belong to exactly one module. The compiler emits an error if you try to declare the same class in more than one module.
- These declared classes are visible within the module but invisible to components in a different module unless they are exported from this module and the other module imports this one.

Bootstrapping Angular

An example of what goes into a declarations array follows:

[declarations](#): [YourComponent, YourPipe, YourDirective],

```
declarations: [  
  YourComponent,  
  YourPipe,  
  YourDirective  
],
```

A declarable can only belong to one module, so only declare it in one `@NgModule`. When you need it elsewhere, import the module that has the declarable you need in it.

Only `@NgModule` references go in the [imports](#) array.

Bootstrapping Angular

The imports array

The module's imports array appears exclusively in the `@NgModule` metadata object. It tells Angular about other NgModules that this particular module needs to function properly.

This list of modules are those that export components, directives, or pipes that the component templates in this module reference. In this case, the component is AppComponent, which references components, directives, or pipes in BrowserModule, FormsModule, or HttpClientModule. A component template can reference another component, directive, or pipe when the referenced class is declared in this module or the class was imported from another module.

Bootstrapping Angular

The providers array

- The providers array is where you list the services the app needs. When you list services here, they are available app-wide. You can scope them when using feature modules and lazy loading.

The [bootstrap](#) array

- The application launches by bootstrapping the root AppModule, which is also referred to as an entryComponent. Among other things, the bootstrapping process creates the component(s) listed in the [bootstrap](#) array and inserts each one into the browser DOM.
- Each bootstrapped component is the base of its own tree of components. Inserting a bootstrapped component usually triggers a cascade of component creations that fill out that tree.
- While you can put more than one component tree on a host web page, most applications have only one component tree and bootstrap a single root component.
- This one root component is usually called AppComponent and is in the root module's [bootstrap](#) array.
-

Bundling Angular Project

Webpack is a popular module bundler, a tool for bundling application source code in convenient *chunks* and for loading that code from a server into a browser.

Webpack is a powerful module bundler. A bundle is a JavaScript file that incorporates assets that belong together and should be served to the client in a response to a single file request. A bundle can include JavaScript, CSS styles, HTML, and almost any other kind of file.

Webpack roams over your application source code, looking for import statements, building a dependency graph, and emitting one or more bundles. With plugins and rules, Webpack can preprocess and minify different non-JavaScript files such as TypeScript, SASS, and LESS files.

ANGULAR PROJECT index.html



```
<!doctype html>
<html lang="en">
<head>
  <meta charset='utf-8'>
  <title>MyFirstApp</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>
<script type="text/javascript" src="inline.bundle.js"></script><script type="text/javascript" src="polyfills.bundle.js"></script>
<script type="text/javascript" src="styles.bundle.js"></script><script type="text/javascript" src="vendor.bundle.js"></script>
<script type="text/javascript" src="main.bundle.js"></script></body>
</html>
```

ANGULAR PROJECT index.html

inline.bundle.js: A tiny file that helps webpack load resources

main.bundle.js: The bulk of our Angular application

polyfills.bundle.js: The things needed to let Angular work in older browsers

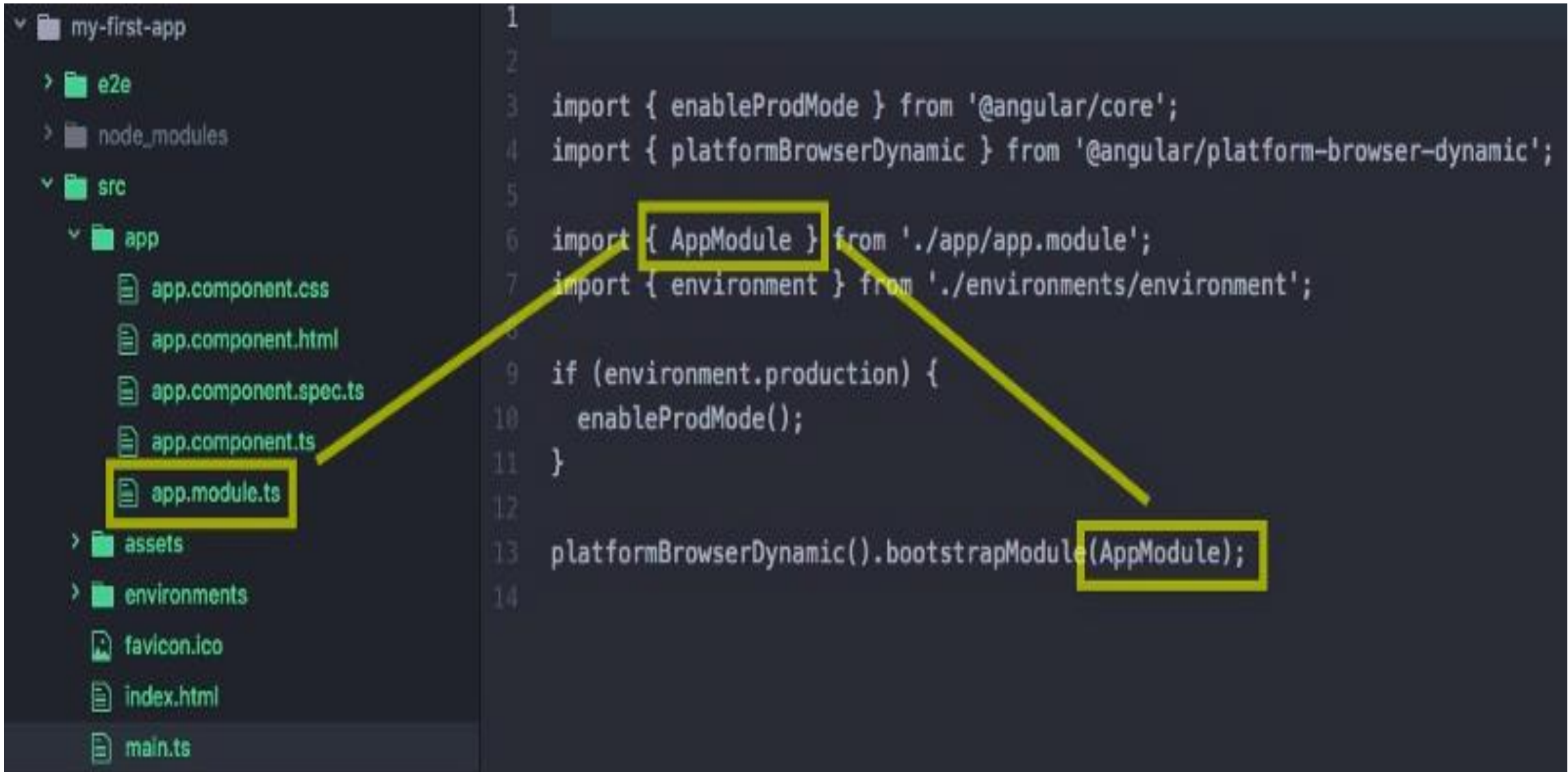
styles.bundle.js: The styles!

vendor.bundle.js: The Angular specific libraries

ANGULAR PROJECT index.html

These script imports are then executed starting with our main.ts file. The last line is the most important as it bootstrap starts our angular application by passing an AppModule to the bootstrapModule method. Angular then knows to go to the app.module.ts file that AppModule is referring to.

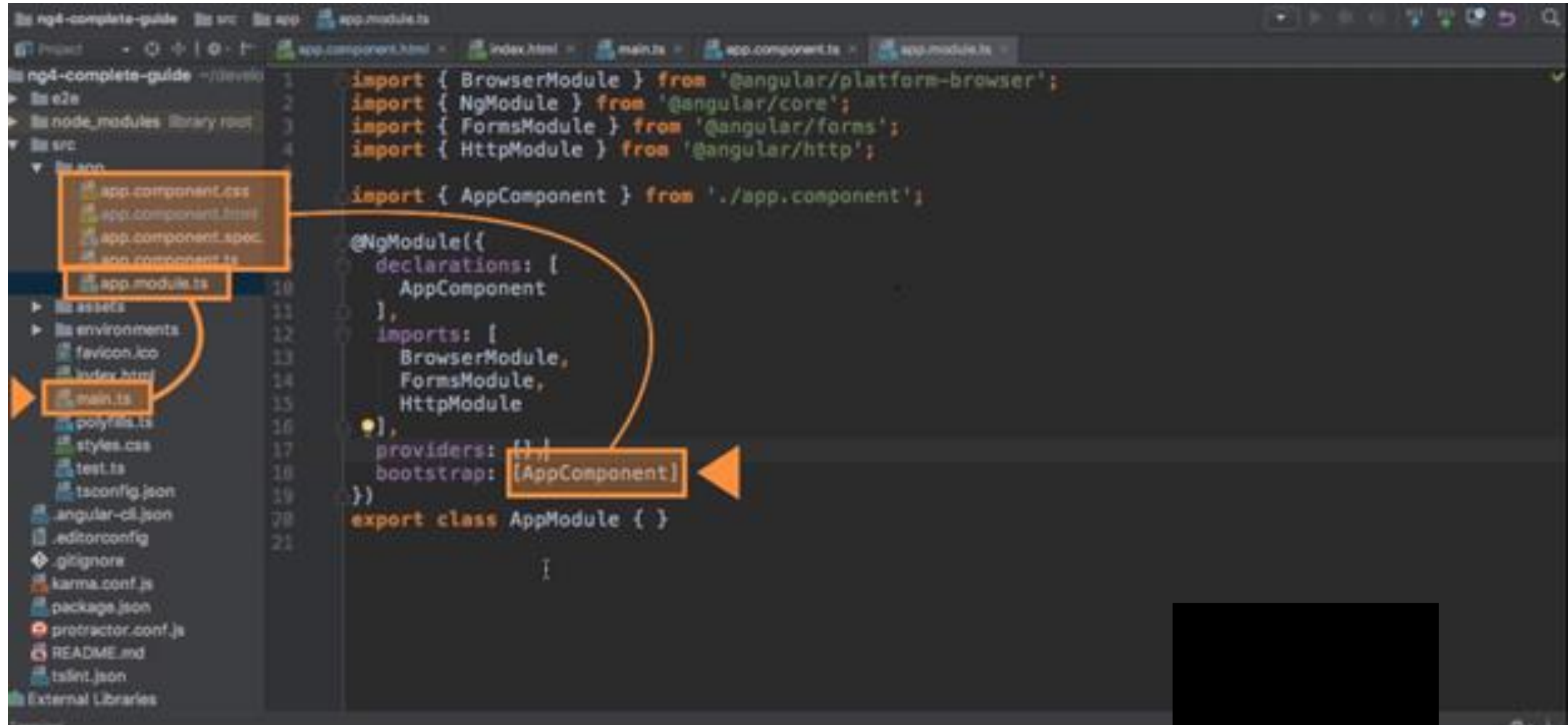
ANGULAR PROJECT



The image shows a file explorer on the left and a code editor on the right. The file explorer displays the structure of a project named 'my-first-app'. The 'src' directory is expanded, showing subdirectories 'app', 'assets', and 'environments'. The 'app' directory contains 'app.component.css', 'app.component.html', 'app.component.spec.ts', 'app.component.ts', and 'app.module.ts'. The 'app.module.ts' file is highlighted with a yellow box. A yellow arrow points from this box to the 'AppModule' import in the code editor. Another yellow arrow points from the 'AppModule' argument in the 'bootstrapModule' call to a yellow box around 'AppModule' in the same call.

```
1  
2  
3 import { enableProdMode } from '@angular/core';  
4 import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';  
5  
6 import { AppModule } from './app/app.module';  
7 import { environment } from './environments/environment';  
8  
9 if (environment.production) {  
10   enableProdMode();  
11 }  
12  
13 platformBrowserDynamic().bootstrapModule(AppModule);  
14
```

ANGULAR PROJECT



ANGULAR PROJECT

In `app.module.ts` there is a bootstrap array that lists all of the components that should be known to Angular at the moment it analyzes our `index.html` file. This is where we reference our app component that we created in `app.component.ts` and Angular moves on to that file.

ANGULAR PROJECT

app.module.ts

```
1
2 import { BrowserModule } from '@angular/platform-browser';
3 import { NgModule } from '@angular/core';
4 import { FormsModule } from "@angular/forms";
5
6 import { AppComponent } from '../app.component';
7
8 @NgModule({
9   declarations: [
10     AppComponent
11   ],
12   imports: [
13     BrowserModule,
14     FormsModule
15   ],
16   providers: [],
17   bootstrap: [AppComponent]
18 })
19 export class AppModule { }
20
```


ANGULAR PROJECT

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  name = '';
}
```

ANGULAR PROJECT

Angular then analyzes the app component in the `app.component.ts` file and recognizes the app-root selector that it saw in our `index.html` file. This tells Angular how to handle the app-root in the body of our `html` and to insert the HTML of our app component (stored in the `app.component.html` file) into that part of the DOM.

ANGULAR PROJECT

Lastly, the way we bind data from our app component to the component's HTML is through using directives. In our `app.component.html` file, you can see that we created the input field and attached `[(ngModel)]='name'` to it. This tag in the HTML tells Angular to go to our component and look in our `AppComponent` class to find what 'name' is equal to. Since `name` is set to an empty string in our component, the text field starts out as blank. As we type, the value of `name` in our component gets updated dynamically.

ANGULAR PROJECT

We are also able to print that value on the browser by interpolating the component's property with double curly brackets {{}}.

```
<input type="text" [(ngModel)]="name">  
<p>{{ name }}</p>
```

And that's how our basic Angular application works!

Eight building blocks of Angular 2.X:

- **Modules:** Angular apps are modular and Angular has its own modularity system called Angular modules or NgModules.
- **Components:** A component is that which controls the View
- **Templates:** A template is a form of HTML that tells Angular how to render the component. The View is defined using template.
- **Metadata:** Metadata tells Angular how to process a class
- **Data Binding:** A mechanism for coordinating between "parts of a template" and "parts of a component". There are four forms of data binding syntax
- **Directives:** A directive is a class with directive metadata. When Angular renders templates, it transforms the DOM according to the instructions given by directives.
- **Services:** Service is a broad category encompassing any value, function, or feature that an application needs. It is typically a class with a narrow, well-defined purpose.
- **Dependency Injection:** Dependency injection is a way to supply a new instance of a class with the fully-formed dependencies it requires. Most dependencies are services. Angular uses dependency injection to provide new components with the services they need.

About package.json

The CLI command **ng new** creates a package.json file when it creates the new workspace. This package.json is used by all projects in the workspace, including the initial app project that is created by the CLI when it creates the workspace.

Initially, this package.json includes a starter set of packages, some of which are required by Angular and others that support common application scenarios. You add packages to package.json as your application evolves.

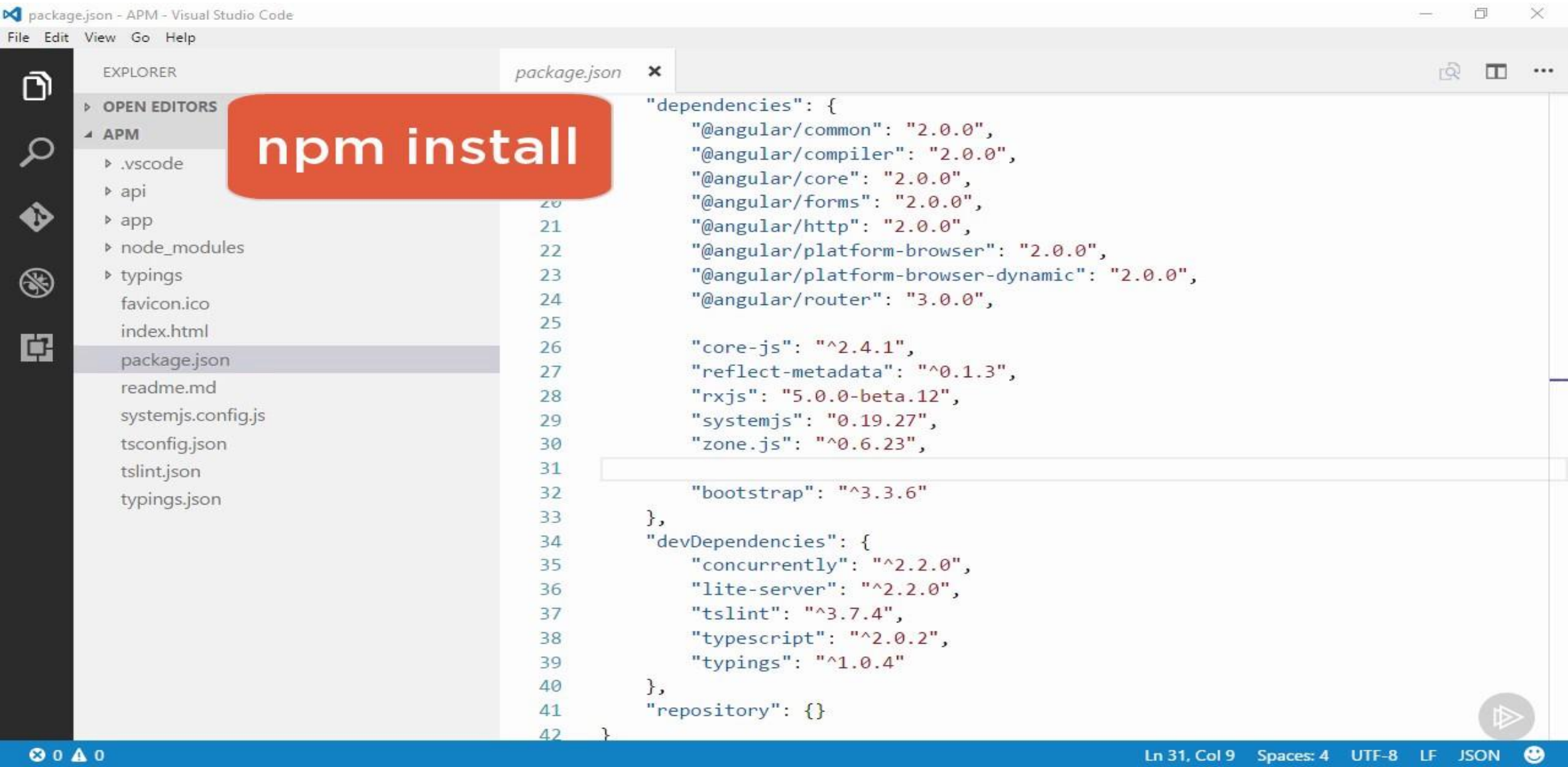
The dependencies section of package.json contains:

Angular packages: Angular core and optional modules; their package names begin @angular/.

Support packages: 3rd party libraries that must be present for Angular apps to run.

Polyfill packages: Polyfills plug gaps in a browser's JavaScript implementation.

About package.json



The screenshot shows the Visual Studio Code interface with a file explorer on the left and a code editor on the right. The file explorer shows a project structure with files like .vscode, api, app, node_modules, typings, favicon.ico, index.html, package.json, readme.md, systemjs.config.js, tsconfig.json, tslint.json, and typings.json. The package.json file is selected. The code editor shows the content of package.json, which includes dependencies and devDependencies. A red button with the text "npm install" is overlaid on the left side of the code editor.

```
package.json
```

```
{
  "dependencies": {
    "@angular/common": "2.0.0",
    "@angular/compiler": "2.0.0",
    "@angular/core": "2.0.0",
    "@angular/forms": "2.0.0",
    "@angular/http": "2.0.0",
    "@angular/platform-browser": "2.0.0",
    "@angular/platform-browser-dynamic": "2.0.0",
    "@angular/router": "3.0.0",

    "core-js": "^2.4.1",
    "reflect-metadata": "^0.1.3",
    "rxjs": "5.0.0-beta.12",
    "systemjs": "0.19.27",
    "zone.js": "^0.6.23",

    "bootstrap": "^3.3.6"
  },
  "devDependencies": {
    "concurrently": "^2.2.0",
    "lite-server": "^2.2.0",
    "tslint": "^3.7.4",
    "typescript": "^2.0.2",
    "typings": "^1.0.4"
  },
  "repository": {}
}
```

Ln 31, Col 9 Spaces: 4 UTF-8 LF JSON

Purpose of Dependencies/API

@angular/core

Critical run-time parts of the framework needed by every application. Includes all metadata decorators, Component, Directive, dependency injection, and the component life-cycle hooks. Contains core functionality component views, DI and change detection.

@angular/compiler

Angular's Template Compiler. It reads your templates and can convert them to code that makes the application run and render. Typically you don't interact with the compiler directly; rather, you use it indirectly via platform-browser-dynamic or the offline template compiler.

Purpose of Dependencies/API

@angular/common

Provides the commonly needed services, pipes, and directives such as ngIf and ngFor.

@angular/platform-browser

Contains the functionality to bootstrap the application in a browser. Basically it includes everything DOM and browser related, especially the pieces that help render into the DOM. May not be required if you use Angular on the platform other than browser (e.g. angular-iot).

This package also includes the bootstrapStatic() method for bootstrapping applications for production builds that pre-compile templates offline.

Purpose of Dependencies/API

@angular/platform-browser-dynamic

Contains implementations for the dynamic bootstrap of the application. Includes providers and a bootstrap method for applications that compile templates on the client (thus, you can skip this module if you use ahead-of-time compilation). Use this package for bootstrapping your application during development.

Purpose of Dependencies/API

To install Angular dependencies run the command (copy the below into a single line):

```
npm i --save @angular/core @angular/compiler @angular/common  
@angular/platform-browser @angular/platform-browser-dynamic
```

Create Component, HTML Template and Styles

app.component.ts

```
import { Component } from '@angular/core';
@Component({
  selector: 'msg-app',
  templateUrl: 'app/app.component.html',
  styleUrls: ['app/app.component.css']
})
export class AppComponent {
  message = "Hello Angular 2 World with TypeScript";
}
```

Create Component, HTML Template and Styles

@Component decorator has been defined in angular **core** package. So we need to import it using import keyword. To import

@angular/core in component means we actually import

**node_modules/@angular/core/bundles/core.
umd.js**

Create Component, HTML Template and Styles

To understand @angular/core mapping with complete path, refer the file systemjs.config.js given in this example. Now find some metadata of @Component decorator.

selector : Defines component element.

templateUrl : Defines HTML template URL.

template : Here we can write inline HTML code.

styleUrls : Defines styles.

Create Component, HTML Template and Styles

Find the HTML template of AppComponent component.

app.component.html

```
<h1> {{message}} </h1>
```

In above HTML template we are fetching value of component property message defined in AppComponent. Now find the CSS used in our example. **app.component.css**

```
h1 {  
  color: #369;  
  font-family: Arial, Helvetica, sans-serif;  
  font-size: 250%;  
}
```

The above CSS has been defined in AppComponent, so this CSS will be applied only in its HTML template that is app.component.html .

Component Class Properties

Creating the Component Class

app.component.ts

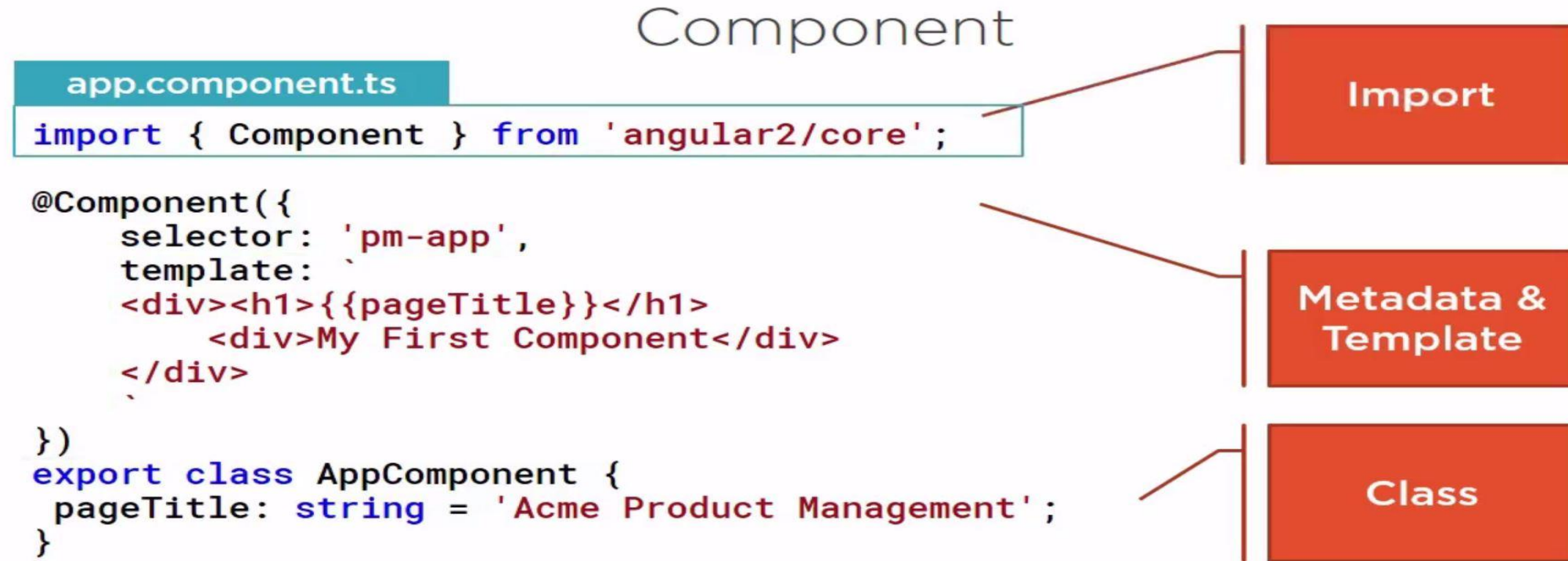
```
export class AppComponent {  
  pageTitle: string = 'Acme Product Management';  
}
```

Property
Name

Data Type

Default
Value

Component detail



Component Class

Creating the Component Class

app.component.ts

```
export class AppComponent {  
  pageTitle: string = 'Acme Product Management';  
}
```

class
keyword

Class Name

export
keyword

Component Name
when used in code

Decorator

Decorator

A function that adds **metadata** to a class, its members, or its method arguments.

Prefixed with an @.

Angular provides built-in decorators.

@Component()

Decorators

- Like attributes in C#/VB.NET

```
@Component({  
  selector: 'expense-app',  
  templateUrl: './app/app.component.html'  
})  
export class AppComponent { }
```

Defining the MetaData

Defining the Metadata

app.component.ts

```
@Component({  
  selector: 'pm-app',  
  template: `  
    <div><h1>{{pageTitle}}</h1>  
      <div>My First Component</div>  
    </div>  
  `,  
})  
export class AppComponent {  
  pageTitle: string = 'Acme Product Management';  
}
```

Component
decorator

Directive Name
used in HTML

View Layout

Binding

Importing What We Need

Importing What We Need

app.component.ts

```
import { Component } from 'angular2/core';
```

```
@Component({  
  selector: 'pm-app',  
  template: `  
    <div><h1>{{pageTitle}}</h1>  
      <div>My First Component</div>  
    </div>  
  `,  
})  
export class AppComponent {  
  pageTitle: string = 'Acme Product Management';  
}
```

import keyword

Angular library
module name

Member name

Angular Application Startup

Angular 2 Application Startup

index.html

```
System.import('app/main');
```

```
<body>
  <pm-app>Loading App ...
</pm-app>
</body>
```

main.ts (bootstrapper)

```
import { bootstrap }
  from 'angular2/
    platform/browser';

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

bootstrap(AppComponent);
```

app.component.ts

```
@Component({
  selector: 'pm-app',
  template: `
    <div>{{pageTitle}}</div>
  `
})
export class AppComponent
{ }
```

Styling a component

- You can add CSS directly to a component!

```
@Component({  
  selector: 'my-app',  
  template: '<h4 class="red">{{successString}}</h4>',  
  styles: [  
    `.red {  
      color: red;  
    }`  
  ]  
})
```



an array of strings that
takes CSS

What is TypeScript?

- ✓ A strict superset of JavaScript with added features
- ✓ Maintained by Microsoft
- ✓ Optional static typing
- ✓ Class-based object-oriented programming
- ✓ Resembles languages like Java and C/C++

What is TypeScript

- Open source language.
- Superset of JavaScript.
- Transpiles to plain JavaScript.
- Strongly typed.
- .ts Extension.
- Class-Based object orientation.

Type Script Editors

- Visual Studio.
- Visual Studio Code.
- Atom.
- Eclipse.

Data types in TypeScript

- Whenever a variable is created, the intention is to assign some value to that variable but what type of value can be assigned to that variable is dependent upon the datatype of that Variable. In TypeScript, type System represents different types of datatypes which are supported by TypeScript.

Data types in TypeScript

Built-in Datatypes: TypeScript has some pre-defined data-types-

BUILT-IN DATA TYPE	KEYWORD	DESCRIPTION
Number	number	It is used to represent both Integer as well as Floating-Point numbers
Boolean	boolean	Represents true and false
String	string	It is used to represent a sequence of characters
Void	void	Generally used on function return-types
Null	null	It is used when an object does not have any value
Undefined	undefined	Denotes value given to uninitialized variable
Any	any	If variable is declared with any data-type then any type of value can be assigned to that variable

Data types in TypeScript

Built-in Datatypes:

Examples:

```
let a: null = null;
```

```
let b: number = 123;
```

```
let c: number = 123.456;
```

```
let d: string = 'Geeks';
```

```
let e: undefined = undefined;
```

```
let f: boolean = true;
```

Using Logic in Angular

NgIf

The NgIf directive is used when you want to display or remove an element based on a condition.

If the condition is false the element the directive is attached to will be removed from the DOM.

The difference between `[hidden]='false'` and `*ngIf='false'` is that the first method simply hides the element. The second method with ngIf removes the element completely from the DOM.

Using Logic in Angular

```
export class AppComponent {  
  people: any[] = [  
    {  
      "name": "Douglas Pace",  
      "age": 35  
    },  
    {  
      "name": "Mcleod Mueller",  
      "age": 32  
    },  
    {  
      "name": "Day Meyers",  
      "age": 21  
    }  
  ];  
}
```

//The NgIf directive removes the li element from the DOM if person.age is less than 30.

Using Logic in Angular

App.component.html

```
<ul *ngFor="let person of people">  
  <li *ngIf="person.age < 35">  
    {{ person.name }} {{ person.age }}  
  </li>  
</ul>
```

Using Logic ngSwitch

- Lets imagine we wanted to print peoples names in different colours depending on *where* they are from. Green for UK, Blue for USA, Red for HK.
- With bootstrap we can change the text color by using the text-danger, text-success, text-warning and text-primary classes.

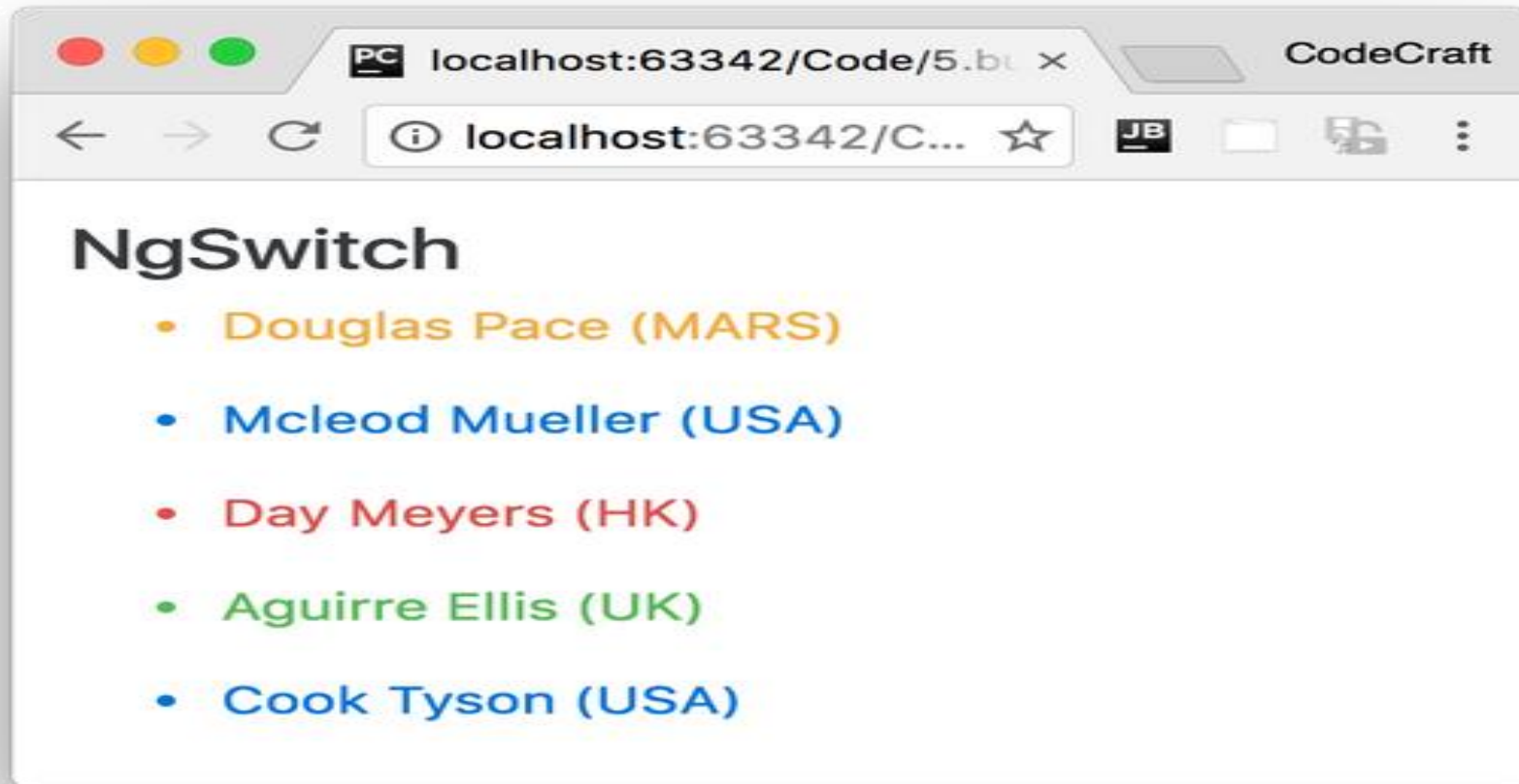
Using Logic app.component.ts

```
Component({
  selector: 'ngswitch-example',
  template: `<h4>NgSwitch</h4>
<ul *ngFor="let person of people"
  [ngSwitch]="person.country">
  <li *ngSwitchCase="'UK'"
    class="text-success">{{ person.name }} ({{ person.country }})
  </li>
  <li *ngSwitchCase="'USA'"
    class="text-primary">{{ person.name }} ({{ person.country }})
  </li>
  <li *ngSwitchCase="'HK'"
    class="text-danger">{{ person.name }} ({{ person.country }})
  </li>
  <li *ngSwitchDefault
    class="text-warning">{{ person.name }} ({{ person.country }})
  </li>
</ul>`
})
```

Using Logic app.component.ts

```
class NgSwitchExampleComponent {  
  people: any[] = [  
    {  
      "name": "Douglas Pace",  
      "age": 35,  
      "country": 'MARS'  
    },  
    {  
      "name": "Mcleod Mueller",  
      "age": 32,  
      "country": 'USA'  
    },  
    {  
      "name": "Cook Tyson",  
      "age": 32,  
      "country": 'USA'  
    }  
  ];  
}
```

Result



Using Logic app.component.ts

- We bind an expression to the ngSwitch directive.
- The ngSwitchCase directive lets us define a condition which if it matches the expression in, will render the element it's attached to.
- If no conditions are met in the switch statement it will check to see if there is an ngSwitchDefault directive, if there is it will render the element that's attached to, however it is optional — if it's not present it simply won't display anything if no matching ngSwitchCase directive is found.
- The key difference between the ngIf solution is that by using NgSwitch we evaluate the expression only once and then choose the element to display based on the result.

Data Binding in Angular

Data binding is a core concept in Angular and allows to define communication between a component and the DOM, making it very easy to define interactive applications without worrying about pushing and pulling data.

WHAT IS DATA BINDING

Data binding is a core concept in Angular and allows to define communication between a component and the DOM, making it very easy to define interactive applications without worrying about pushing and pulling data.

There are four forms of data binding

1. Interpolation: `{{ value }}`
2. Property binding: `[property]="value"`
3. Event binding: `(event)="function"`
4. Two-way data binding: `[(ngModel)]="value"`

Data Binding in Angular

From the Component to the DOM

Interpolation: {{ value }}

This adds the value of a property from the component:

```
<li>Name: {{ user.name }}</li>
```

```
<li>Email: {{ user.email }}</li>
```

Property binding: [property]="value"

With property binding, the value is passed from the component to the specified property, which can often be a simple html attribute:

```
<input type="email" [value]="user.email">
```

Here are two more examples, one that applies a background-color from the value of selectedColor in the component and one that applies a class name if isSelected evaluates to true:

```
<div [style.background-color]="selectedColor">
```

```
<div [class.selected]="isSelected">
```

Data Binding in Angular

Event binding: (event)="function"

When a specific DOM event happens (eg.: click, change, keyup), call the specified specified method in the component.

Two-way data binding: [(ngModel)]="value"

Using what's called the banana in a box syntax, two-way data binding allows to have the data flow both ways. In this example, the user.email data property is used as the value for the input, but if the user changes the value, the component property gets updated automatically to the new value:

Example: component

Angular 2 - Data Binding Step 1 – Download any 2 images. For this example, we will download some simple images shown below.

Step 2 – Store these images in a folder called **Images** in the app directory. If the images folder is not present, please create it.

Example: component

Step 3 – Add the following content in app.component.ts as shown below.

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

```
@Component ({  
  selector: 'my-app',  
  templateUrl: 'app/app.component.html'  
})
```

```
export class AppComponent {  
  appTitle: string = 'Welcome';  
  appList: any[] = [ {  
    "ID": "1",  
    "url": 'app/Images/One.jpg'  
  },  
  {  
    "ID": "2",  
    "url": 'app/Images/Two.jpg'  
  } ];  
}
```

Example: component

Step 3 – Add the following content in app.component.ts as shown below.

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

```
@Component ({  
  selector: 'my-app',  
  templateUrl: 'app/app.component.html'  
})
```

```
export class AppComponent {  
  appTitle: string = 'Welcome';  
  appList: any[] = [ {  
    "ID": "1",  
    "url": 'app/Images/One.jpg'  
  },  
  {  
    "ID": "2",  
    "url": 'app/Images/Two.jpg'  
  } ];  
}
```

Example: component

```
<div *ngFor = 'let lst of appList'>  
  <ul>  
    <li>{{lst.ID}}</li>  
    <img [src] = 'lst.url'>  
  </ul>  
</div>
```

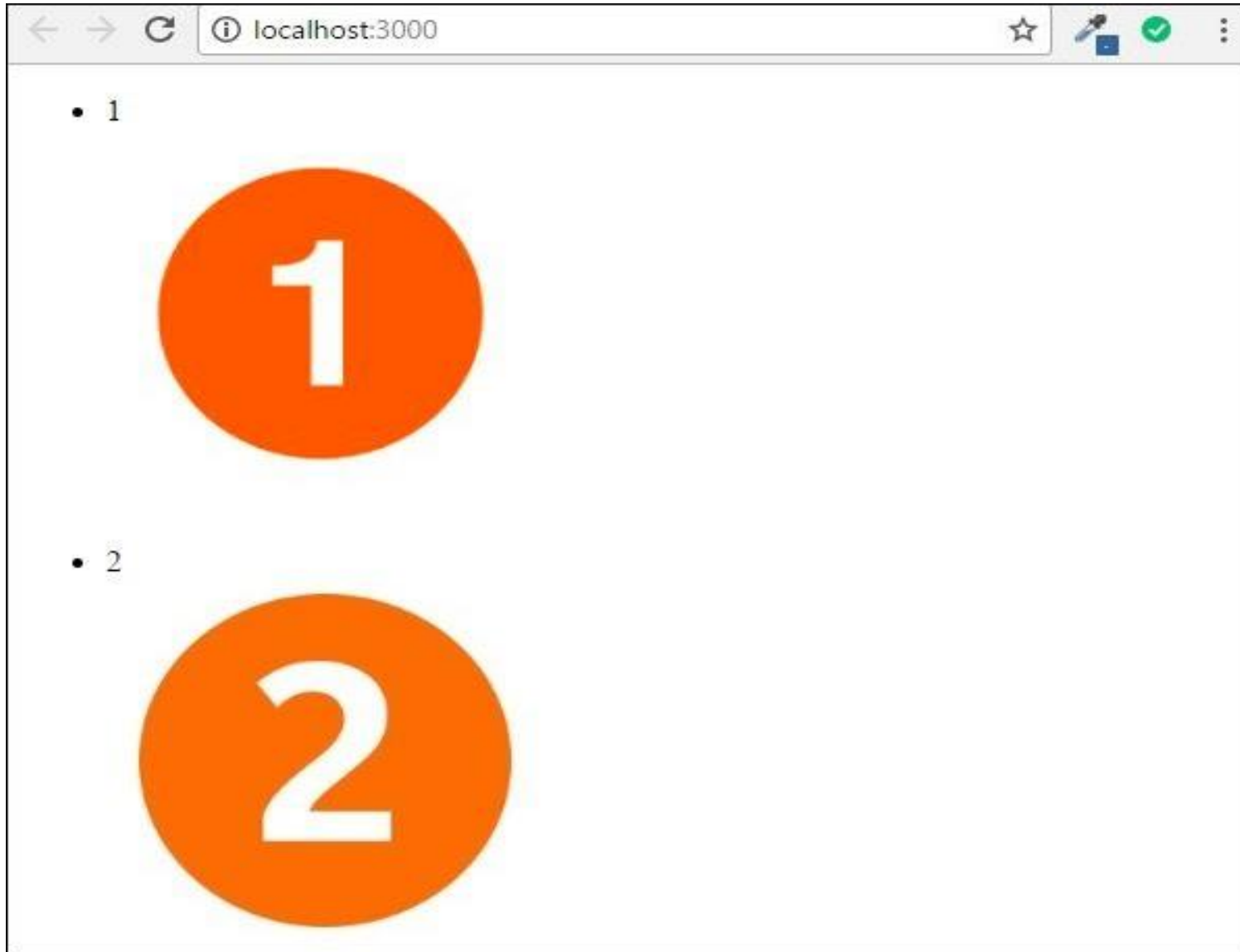
Example: component

In the above `app.component.html` file, we are accessing the images from the properties in our class.

Output

The output of the above program should be like this –

Example: component



Event binding

(focus)="myMethod()" // An element has received focus

(blur)="myMethod()" // An element has lost focus

(submit)="myMethod()" // A submit button has been pressed

(keydown)="myMethod()"

(keypress)="myMethod()"

(keyup)="myMethod()"

(mousedown)="myMethod()"

(click)="myMethod()"

(dblclick)="myMethod()"

Event binding

Ng1 directive	Functionality	Replaced with
ng-click	Binds an expression to the click event	(click)="doSomething()"
ng-change	Binds an expression to the input changed event	(change)="prop = \$event" (ngModelChange)=""
ng-dblclick	Binds an expression to the double click event	(dblclick)="onDoubleClick()"

Event binding

- Bind directly to events on the DOM model using ()

```
@Component({
  selector: 'my-app',
  template: `<button (click)="showMeAlert()">Click me!</button>`
})
export class AppComponent {
  showMeAlert() {
    alert("Show me the money");
  }
}
```

Example: component

```
import { Component } from '@angular/core';
@Component({
  selector: 'my-component',
  template: `
    <div>Hello my name is {{name}}.
    <button (click)="sayMyName()">Say my name</button>
  </div>
  `
})
export class MyComponent {
  name: string;
  constructor() {
    this.name = 'Max'
  }
  sayMyName() {
    console.log('My name is', this.name)
  }
}
```

Pipes

Transforming Data with Pipes

**Transform
bound
properties
before
display**

Built-in pipes

- date
- number, decimal, percent, currency
- json, slice
- etc

**Custom
pipes**

Using pipes

date

This is used to convert the input string to date format.

Syntax

Propertyvalue | date:"dateformat"

Parameters

dateformat – This is the date format the input string should be converted to.

Result

The property value will be converted to date format.

Using pipes

```
import { Component } from '@angular/core';  
@Component ({  
  selector: 'my-app',  
  templateUrl: 'app/app.component.html'  
})
```

```
export class AppComponent {  
  newdate = new Date(2016, 3, 15);  
}
```

Using pipes

```
import {Component} from 'angular2/core'

@Component({
  selector: 'hero-birthday',
  template: `<p>The hero's birthday is
    {{ birthday | date }}</p>`
})
export class HeroBirthday {
  birthday = new Date(1988,3,15); // April 15, 1988
}
```


Using pipes :

app/app.component.ts

```
import { Component } from '@angular/core';
@Component ({
  selector: 'my-app',
  templateUrl: 'app/app.component.html'
})
export class AppComponent {
  Name1: string = 'Angular2';
  appList: string[] = ["Binding", "Display", "Services"];
}
```

Using pipes :

app/app.component.html

```
<div>
```

```
  The name is {{Name1}}<br>
```

```
  The first Topic is {{appList[0] | lowercase}}<br>
```

```
  The second Topic is {{appList[1] | lowercase}}<br>
```

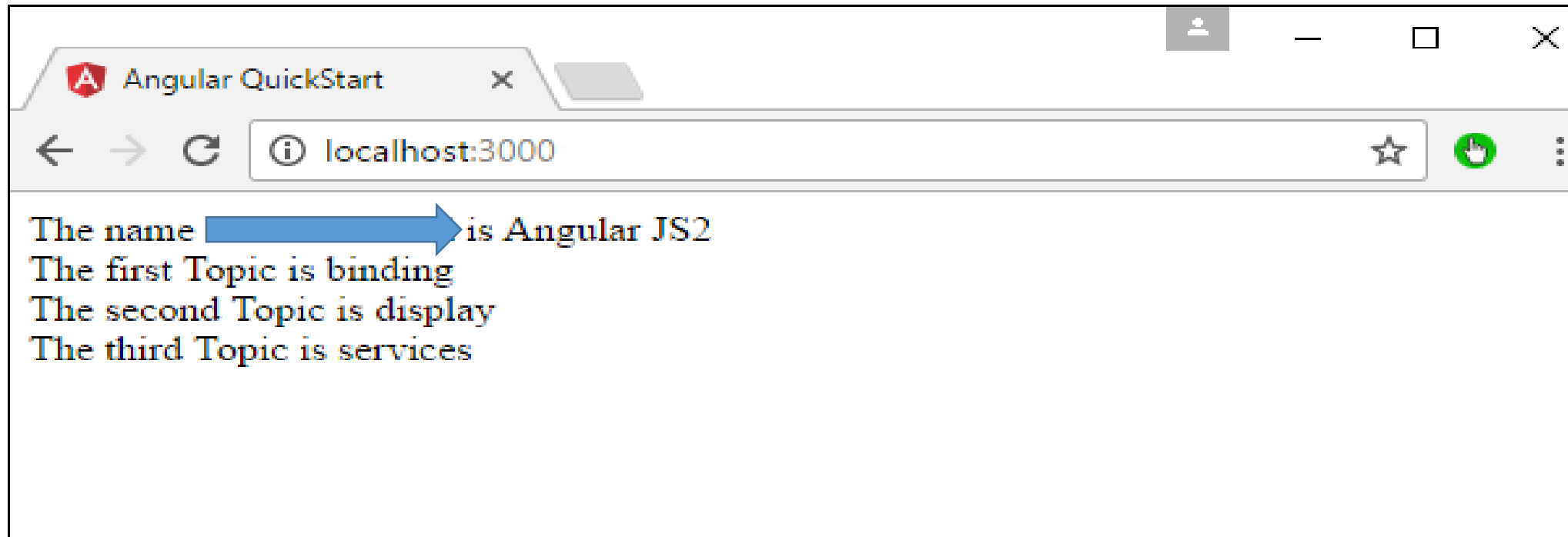
```
  The third Topic is {{appList[2] | lowercase}}<br>
```

```
</div>
```

Using pipes :

Output

Once you save all the code changes and refresh the browser, you will get the following output.



Using pipes :

app/app.component.html

```
<div>
```

```
  The name is {{Name1}}<br>
```

```
  The first Topic is {{appList[0] | uppercase}}<br>
```

```
  The second Topic is {{appList[1] | uppercase}}<br>
```

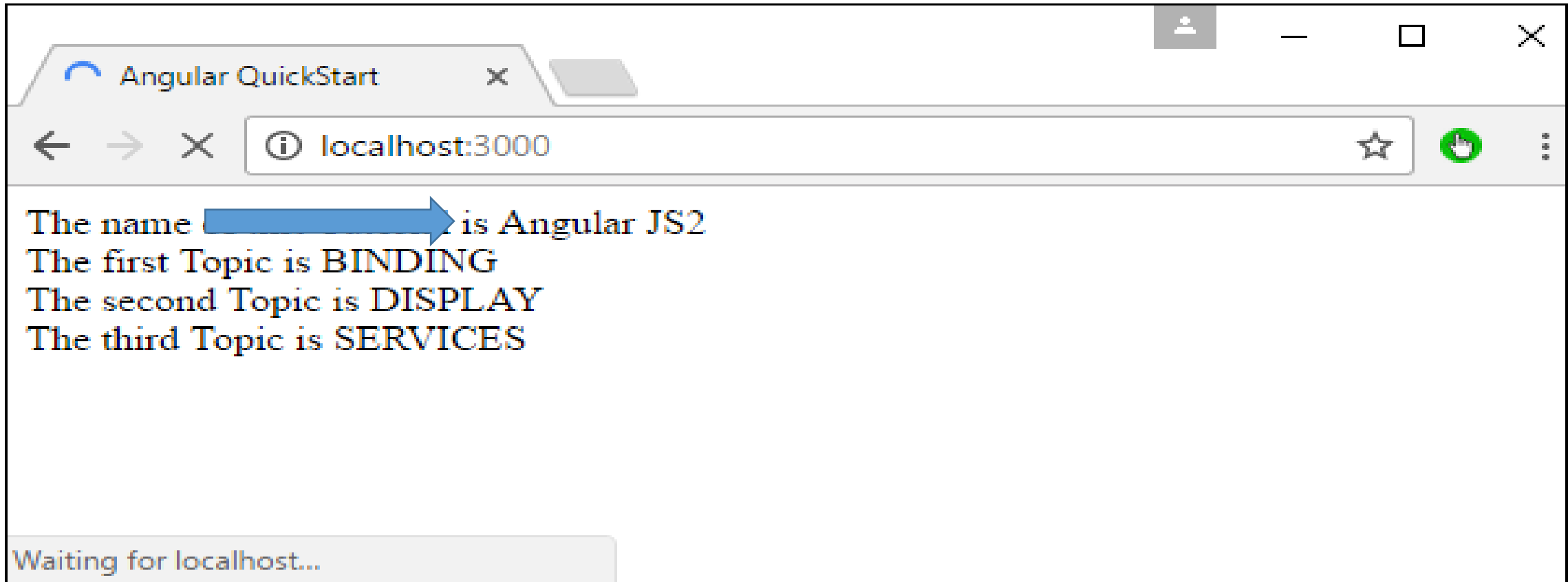
```
  The third Topic is {{appList[2] | uppercase}}<br>
```

```
</div>
```

Using pipes :

Output

Once you save all the code changes and refresh the browser, you will get the following output.



Using pipes : slice

This is used to slice a piece of data from the input string.

Propertyvalue | slice:start:end

Parameters

start – This is the starting position from where the slice should start.

end – This is the starting position from where the slice should end.

Result

The property value will be sliced based on the start and end positions.

Using pipes :SLICE

app/app.component.ts

```
import { Component } from '@angular/core';
@Component ({
  selector: 'my-app',
  templateUrl: 'app/app.component.html'
})
export class AppComponent {
  Name1: string = 'Angular2';
  appList: string[] = ["Binding", "Display", "Services"];
}
```

The name is {{Name1}}

The first Topic is {{appList[0] | slice:1:2}}

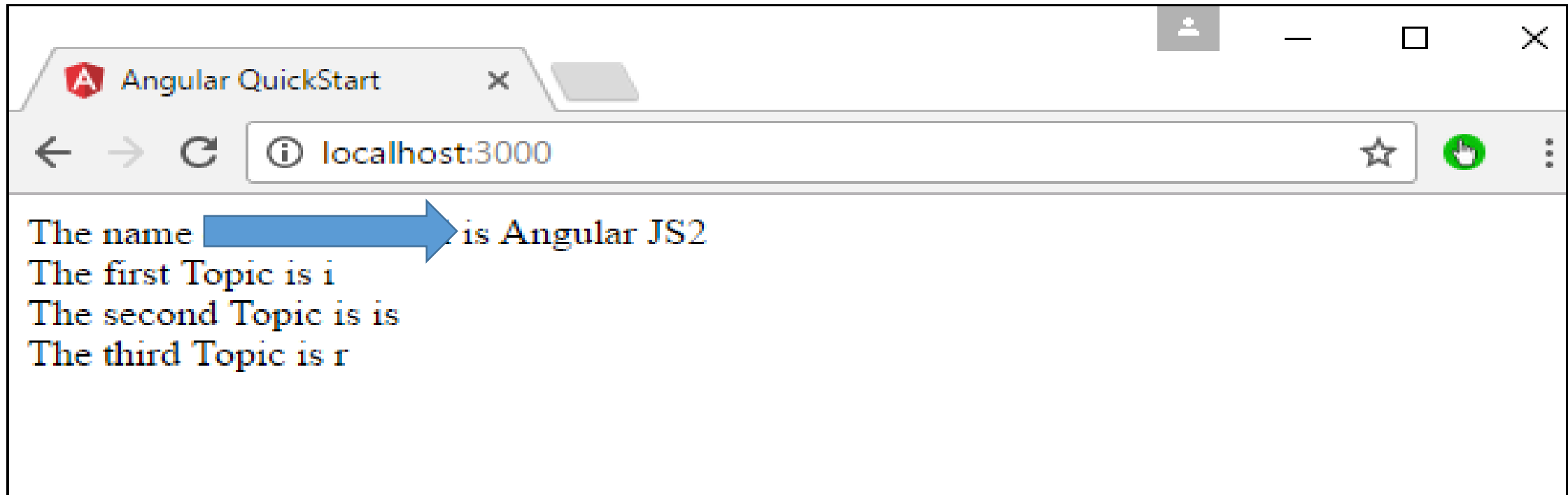
The second Topic is {{appList[1] | slice:1:3}}

The third Topic is {{appList[2] | slice:2:3}}

Using pipes :SLICE

Output

Once you save all the code changes and refresh the browser, you will get the following output.



Using pipes :currency

This is used to convert the input string to currency
format

Propertyvalue | currency

Result

The property value will be converted to currency
format.

ANGULAR PROJECT

FORM IN ANGULAR

WHAT IS FORMS

Forms are almost always present in any website or application.

Forms are an essential part of many web applications, being the most common way to enter and edit text-based data

Angular allows two types of forms :

1. Template-driven forms – simple forms that can be made rather quickly.
2. Reactive forms – Angular reactive forms, also known as model-driven forms, offers an easy way to use reactive patterns and validations. These are more complex forms that give us greater control over the elements in the form. They follow the reactive programming style that supports an explicit data management flow between non-UI data models (frequently retrieved from a server) and a UI-oriented form model that keeps the states and values of HTML controls on the app screen.

ANGULAR PROJECT

app.module.ts

```
1
2 import { BrowserModule } from '@angular/platform-browser';
3 import { NgModule } from '@angular/core';
4 import { FormsModule } from "@angular/forms";
5
6 import { AppComponent } from '../app.component';
7
8 @NgModule({
9   declarations: [
10     AppComponent
11   ],
12   imports: [
13     BrowserModule,
14     FormsModule
15   ],
16   providers: [],
17   bootstrap: [AppComponent]
18 })
19 export class AppModule { }
20
```

ngModel: Binding

```
import { Component } from '@angular/core';
//import { FormsModule } from '@angular/forms';
@Component({
  selector: 'app-hello-world',
  template: `
    <div>
      <label>Name:</label>
      <input type="text" [(ngModel)]="name">
      <h1>Hello {{name}}!</h1>
    </div>`
})
export class HelloWorldComponent {
  name = 'World';
}
```

Installing Bootstrap from NPM

For Bootstrap 3: `npm install bootstrap@3.3.7`

For Bootstrap 4: `npm install bootstrap jquery`

1: Configure angular.json:

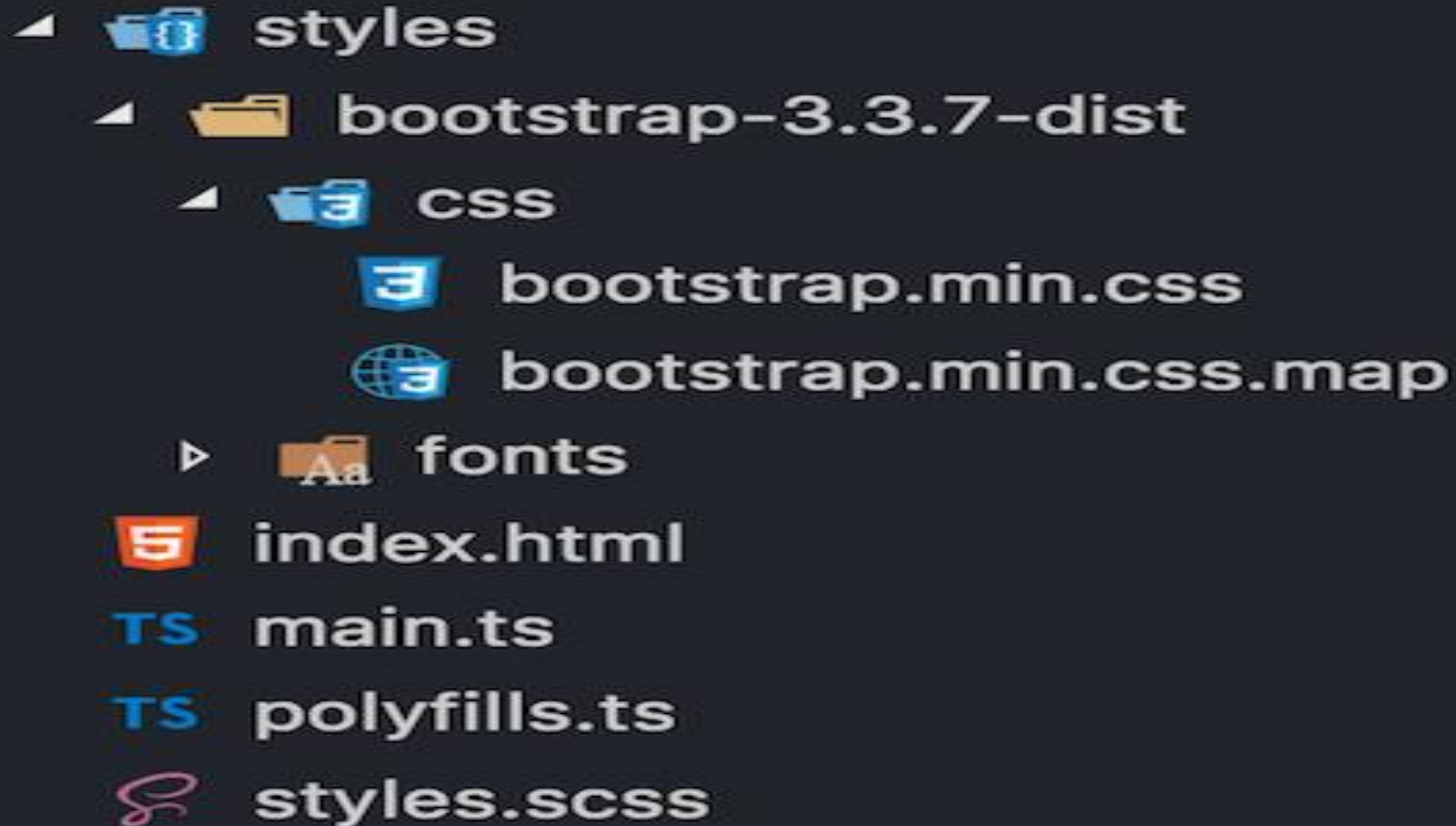
```
"styles": [  
  "src/styles.css",  
  "node_modules/bootstrap/dist/css/bootstrap.min.css"  
],  
"scripts": [  
  "node_modules/jquery/dist/jquery.min.js",  
  "node_modules/bootstrap/dist/js/bootstrap.min.js"  
]
```

2: Import directly in src/style.css or src/style.scss:

```
@import '~bootstrap/dist/css/bootstrap.min.css';
```

Note : Better, import all my styles in src/style.css since it's been declared in angular.json already.

Installing Bootstrap from NPM



A file explorer view showing the structure of a project after installing Bootstrap 3.3.7 from NPM. The directory tree is as follows:

- styles
 - bootstrap-3.3.7-dist
 - css
 - bootstrap.min.css
 - bootstrap.min.css.map
 - fonts
 - index.html
 - main.ts
 - polyfills.ts
 - styles.scss

Install and Use jquery in Angular

To install jquery in Angular use the following node npm command
`npm install jquery -- save`

└─ jquery

└─ dist

JS core.js

JS jquery.js

JS jquery.min.js

≡ jquery.min.map

JS jquery.slim.js

JS jquery.slim.min.js

≡ jquery.slim.min.map

Install and Use jquery in Angular

In your local angular development setup you can see jquery folder under node_modules

And then add the reference to jquery file in Angular.json or Angular-cli.json depending upon the version of Angular

In Latest versions like Angular 7 or Angular 6 it is Angular.json file.

```
"scripts": [  
  "node_modules/jquery/dist/jquery.min.js"  
]
```

Add the reference in scripts array of Angular.json

Install and Use jQuery in Angular

Add the reference in scripts array of Angular.json

And finally declare a variable called jQuery or \$ in the angular component where you want to use jQuery plugin as shown below.

```
import { Component, OnInit } from '@angular/core';
declare var $: any;
@Component({
  selector: 'app-jquery-example',
  templateUrl: './jquery-example.component.html',
  styleUrls: ['./jquery-example.component.css']
})
export class JQueryExampleComponent implements OnInit {
  constructor() { }
  ngOnInit() {
    $(document).ready(function() {
      alert('I am Called From jQuery');
    });
  }
}
```

Now you can see the alert saying I am Called from jQuery in the UI.

DATA BINDING

Text

We use two-way data binding with `ngModel` to bind to the `textValue` property on the component class.

Two-way binding allows us to use the property to set the initial value of an `<input>`, and then have the user's changes flow back to the property. We can also change the property value in the code and have these changes reflected in the `<input>`.

DATA BINDING : text.component.ts

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-text-box',
  template: `
    <h1>Text ({{textValue}})</h1>
    <input #textbox type="text" [(ngModel)]="textValue" required>
    <button (click)="logText(textbox.value)">Update Log</button>
    <button (click)="textValue="">Clear</button>
    <h2>Template Reference Variable</h2>
    Type: '{{textbox.type}}', required: '{{textbox.hasAttribute('required')}}',
    upper: '{{textbox.value.toUpperCase()}}'

    <h2>Log <button (click)="log="">Clear</button></h2>
    <pre>{{log}}</pre>`
  })
export class TextComponent {

  textValue = 'initial value';
  log = "";

  logText(value: string): void {
    this.log += `Text changed to '${value}'\n`;
  }
}
```

ngModel: Binding

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

```
@Component({  
  selector: 'app-two-way-binding',  
  template: `  
    <h1>  
      {{person.name.forename}} {{person.name.surname}}  
      ({{person.address.street}}, {{person.address.city}}, {{person.address.country}})  
    </h1>  
    <p>  
      <input [value]="person.name.forename"  
        (keyup)="person.name.forename=$event.target.value">
```

ngModel: Two Way Binding

```
<input [value]="person.name.surname"
      (input)="person.name.surname=$event.target.value">
</p>
<p>
  Street: <input [ngModel]="person.address.street"
                (ngModelChange)="person.address.street=$event">
</p>
<p>
  City: <input [(ngModel)]="person.address.city">
</p>
<p>
  Country: <input bindon-ngModel="person.address.country">
</p>
<p>
```

ngModel: Binding

```
<button (click)="reset()">Reset</button>
</p>
<pre>
  {{person | json:2}}
</pre>`
  })
export class TwoWayBindingComponent {

  person: any;

  constructor() {
    this.reset();
  }
}
```


ngModel: Binding

```
reset(): void {  
  this.person = {  
    name: {  
      forename: 'John',  
      surname: 'Doe'  
    },  
    address: {  
      street: 'Lexington Avenue',  
      city: 'New York',  
      country: 'USA'  
    }  
  };  
}
```

ngModel: Binding

The Application

Here is the output from this component.

Doe (Lexington Avenue, New York, USA)

Doe

Street:

City:

Country:

```
{
  "name": {
    "forename": "",
    "surname": "Doe"
  },
  "address": {
    "street": "Lexington Avenue",
    "city": "New York",
    "country": "USA"
  }
}
```

ngModel: Binding-TEXTAREA

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-text-area',
  template: `
    <h1>Text Area</h1>
    <textarea ref-textarea [(ngModel)]="textValue"
rows="4"></textarea><br/>
    <button (click)="logText(textarea.value)">Update Log</button>
    <button (click)="textValue="">Clear</button>
```

ngModel: Binding-TEXTAREA

```
<h2>Log <button (click)="log=""></button></h2>
  <pre>{{log}}</pre>`
})
export class TextAreaComponent {

  textValue = 'initial value';
  log = '';

  logText(value: string): void {
    this.log += `Text changed to '${value}'\n`;
  }
}
```

ngModel: Two Way Binding-TEXTAREA

- textarea behaves in a similar way to the textbox. Again, we use two-way data binding with ngModel to bind to the textValue property on the component class to keep user and programmatic changes in sync.
- This time we use the *canonical* form to specify the template reference variable but we could just as easily use #textarea instead.

ngModel: Binding-TEXTAREA

The Application

Here is the output from this component.

Text Area

hi how r u?

Update Log

Clear

Log

Clear

Text changed to 'hi how r u?'

ngModel: Two Way Binding-CHECKBOX

- The change event is triggered when the user clicks on a checkbox. We use this event to pass a template reference variable into a function to log user action. We can determine whether or not the checkbox has been selected using the checked property of the argument.
- We also use the checked property to highlight the selected checkbox labels using class bindings. This uses the styles property to set the text color to OrangeRed for all the selected checkboxes.

ngModel: Binding-CHECKBOX

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

@Component({
  selector: 'app-checkbox',
  template: `
    <h1>Checkbox</h1>
    <p>
      <label [class.selected]="cb1.checked">
        <input #cb1 type="checkbox" value="one" (change)="logCheckbox(cb1)"> One
      </label>
      <label [class.selected]="cb2.checked">
        <input #cb2 type="checkbox" value="two" (change)="logCheckbox(cb2)"> Two
      </label>
      <label [class.selected]="cb3.checked">
        <input #cb3 type="checkbox" value="three" (change)="logCheckbox(cb3)"> Three
      </label>
    </p>
  `
})
```


ngModel: Binding-CHECKBOX

```
<h2>Log <button (click)="log="">>Clear</button></h2>
  <pre>{{log}}</pre>,
  styles: ['.selected {color: OrangeRed;}']
})
export class CheckboxComponent {

  log = '';

  logCheckbox(element: HTMLInputElement): void {
    this.log += `Checkbox ${element.value} was ${element.checked ? '' :
'un'}checked\n`;
  }
}
```

ngModel: Binding-CHECKBOX

The Application

Here is the output from this component.

Checkbox

☐ One ☐ Two ☒ Three

Log

Checkbox two was checked
Checkbox two was unchecked
Checkbox three was checked

ngModel: Binding-CHECKBOX

The Application

Here is the output from this component.

Checkbox

☐ One ☐ Two ☒ Three

Log

Checkbox two was checked
Checkbox two was unchecked
Checkbox three was checked

ngModel: Binding-RADIO

- The radio field behaves in a similar way to the checkbox. We use the change event on the `<input>` element to execute a template statement which passes a template reference variable into a component method.
- The value property property of the `HTMLInputElement` argument is used to log user's selection. This value property corresponds to the value attribute of the HTML `<input>` element.

ngModel: Two Way Binding-RADIO

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

```
@Component({  
  selector: 'app-radio',  
  template: `  
    <h1>Radio</h1>  
    <p>  
      <label [class.selected]="r1.checked">  
        <input #r1 type="radio" name="r" value="one" (change)="logRadio(r1)"> One  
      </label>  
      <label [class.selected]="r2.checked">  
        <input #r2 type="radio" name="r" value="two" (change)="logRadio(r2)"> Two  
      </label>  
      <label [class.selected]="r3.checked">  
        <input #r3 type="radio" name="r" value="three" (change)="logRadio(r3)"> Three  
      </label>  
    </p>  
  `
```

ngModel: Binding-RADIO

```
<h2>Log <button (click)="log="">>Clear</button></h2>
<pre>{{log}}</pre>`,
styles: ['.selected {color: OrangeRed;}']
})
export class RadioComponent {

  log = '';

  logRadio(element: HTMLInputElement): void {
    this.log += `Radio ${element.value} was selected\n`;
  }
}
```

ngModel: Binding-RADIO

The Application

Here is the output from this component.

Radio

☐ One ☐ Two ☐ Three

Log

ngModel: Binding-Drop-down List

- We use an array to populate the drop-down with a list of values. The built-in ngFor directive comes in handy here to loop through the array and set up the option values.
- [(ngModel)] sets up two-way data binding on the select element. This will ensure that the drop-down list will display the current item when it is opened. When a new value is chosen from the list, two-way binding also ensures that the current property on the component is automatically updated.
- An additional binding to the change event of the select element logs the user action by calling a method in the template statement.

ngModel: Binding-Drop-down List

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

```
@Component({  
  selector: 'app-drop-down',  
  template: `  
    <h1>Drop-down List</h1>  
    <select #select [(ngModel)]="current" (change)="logDropdown(select.value)">  
      <option *ngFor="let item of list" [value]="item.id">{{item.name}}</option>  
    </select>  
  
    <h2>Log <button (click)="log="">Clear</button></h2>  
    <pre>{{log}}</pre>  
  `)  
})
```

ngModel: Binding-Drop-down List

```
export class DropDownComponent {  
  
  list: any = [  
    {id: 1, name: 'one'},  
    {id: 2, name: 'two'},  
    {id: 3, name: 'three'}  
  ];  
  current = 2;  
  log = "";  
  
  logDropdown(id: number): void {  
    const NAME = this.list.find((item: any) => item.id === +id).name;  
    this.log += `Value ${NAME} was selected\n`;  
  }  
}
```

ngModel: Binding-Drop-down List

The Application

Here is the output from this component.

Drop-down List

two ▼

Log

Property binding

```
import { Component } from '@angular/core';  
@Component({  
  selector: 'app-property',  
  template: `  
    <h1 [textContent]="Name: " + person.name"></h1>  
  
    <button (click)="person = male" [disabled]="person.sex=='m'">Male</button>  
    <button (click)="person = female" [disabled]="person.sex=='f'">Female</button>
```

Property binding

```
<p><img [src]="person.photo" [alt]="person.name" [title]="person.name"></p>
  <p [hidden]="!person.rating">
    Rating: <span [innerHTML]="'#10032;'.repeat(person.rating)"></span>
  </p>`
})
export class PropertyComponent {
  female = {
    name: 'Turanga Leela',
    sex: 'f',
    rating: 4,
    photo: 'assets/images/leela.jpg'
  };
}
```

Property binding

```
male = {  
  name: 'Philip J. Fry',  
  sex: 'm',  
  photo: 'assets/images/fry.jpg'  
};  
person: any = this.female;  
}
```

Property binding

The Application

Here is the output from this component.

Name: Turanga Leela

☒ Male

☐ Female



Rating: ☆☆☆☆

Property binding

- Property binding means that all sorts of previous built-in directives in Angular 1 go away

Ng1 Directive	Functionality	Replaced with
ng-disabled	Disables the control – used on inputs, selects, textareas	[disabled]="expression"
ng-src/ng-href	Replacement for src/href properties since interpolation didn't play well with them	[src]="expression" [href]="expression"
ng-show/ng-hidden	Hides/shows an element based on the truthiness of an expression	[hidden]="expression"

Property binding

Text

- We use two-way data binding with `ngModel` to bind to the `textValue` property on the component class.
- Two-way binding allows us to use the property to set the initial value of an `<input>`, and then have the user's changes flow back to the property. We can also change the property value in the code and have these changes reflected in the `<input>`.

Property binding

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

```
@Component({  
  selector: 'app-text-box',  
  template: `  
    <h1>Text ({{textValue}})</h1>  
    <input #textbox type="text" [(ngModel)]="textValue" required>  
    <button (click)="logText(textbox.value)">Update Log</button>  
    <button (click)="textValue="">Clear</button>  
  
    <h2>Template Reference Variable</h2>  
    Type: '{{textbox.type}}', required: '{{textbox.hasAttribute('required')}}',  
    upper: '{{textbox.value.toUpperCase()}}'
```

Property binding

```
<h2>Log <button (click)="log="">Clear</button></h2>
  <pre>{{log}}</pre>`
})
export class TextComponent {

  textValue = 'initial value';
  log = '';

  logText(value: string): void {
    this.log += `Text changed to '${value}'\n`;
  }
}
```

Property binding

The Application

Here is the output from this component.

Text (initial value)

Template Reference Variable

Type: 'text', required: 'true', upper: 'INITIAL VALUE'

Log

What is Angular Module?

Angular apps are modular and Angular has its own modularity system called Angular modules or NgModules.

Every Angular app should have at least one module class, the root module. We bootstrap that module to launch the application.

Angular 2 NgModules contains imports, declarations, and bootstrap.

imports - other modules whose exported classes are needed by component templates declared in this module.

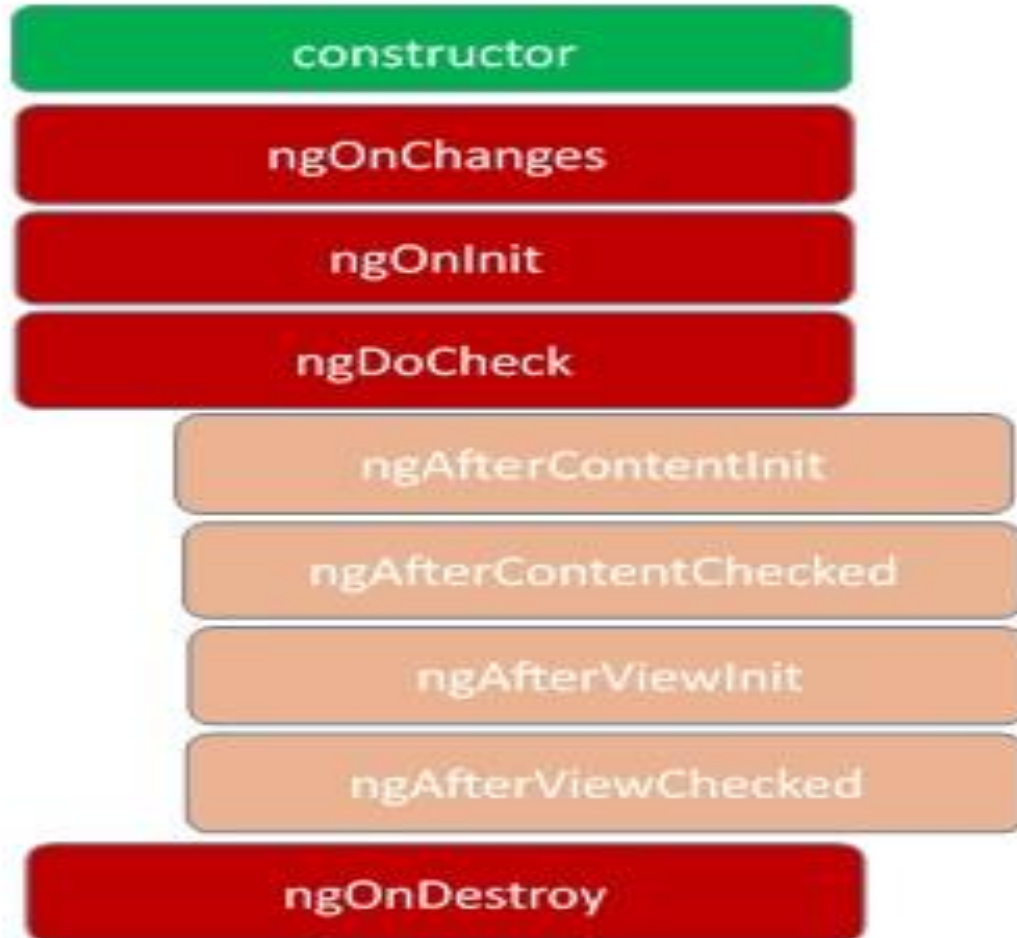
declarations - the view classes that belong to this module. Angular has three kinds of view classes: components, directives, and pipes.

bootstrap - the main application view, called the root component, that hosts all other app views. Only the root module should set this bootstrap property.

Angular Life Cycle

In Angular, every component has a life-cycle, a number of different stages it goes through. There are 8 different stages in the component lifecycle. Every stage is called as lifecycle hook event. So, we can use these hook events in different phases of our application to obtain control of the components. Since a component is a TypeScript class, every component must have a constructor method. The constructor of the component class executes, first, before the execution of any other lifecycle hook events. If we need to inject any dependencies into the component, then the constructor is the best place to inject those dependencies. After executing the constructor, Angular executes its lifecycle hook methods in a specific order.

Angular Life Cycle



Angular Life Cycle

ngOnChanges – This event executes every time when a value of an input control within the component has been changed. Actually, this event is fired first when a value of a bound property has been changed. It always receives a change data map, containing the current and previous value of the bound property wrapped in a SimpleChange.

ngOnInit – This event initializes after Angular first displays the data-bound properties or when the component has been initialized. This event is basically called only after the ngOnChanges() events. This event is mainly used for the initialize data in a component.

Angular Life Cycle

ngDoCheck – This event is triggered every time the input properties of a component are checked. We can use this hook method to implement the check with our own logic check. Basically, this method allows us to implement our own custom change detection logic or algorithm for any component.

ngOnDestroy – This method will be executed just before Angular destroys the components. This method is very useful for unsubscribing from the observables and detaching the event handlers to avoid memory leaks. Actually, it is called just before the instance of the component is finally destroyed. This method is called just before the component is removed from the DOM.

Angular Life Cycle

Lifecycle

ngOnChanges

Called after a bound input property changes

ngOnInit

Called once the component is initialized

ngDoCheck

Called during every change detection run

ngAfterContentInit

Called after content (ng-content) has been projected into view

ngAfterContentChecked

Called every time the projected content has been checked

ngAfterViewInit

Called after the component's view (and child views) has been initialized

ngAfterViewChecked

Called every time the view (and child views) have been checked

ngOnDestroy

Called once the component is about to be destroyed



Angular Life Cycle

```
import { Component } from '@angular/core';
import { OnInit, OnDestroy, AfterContentInit, AfterViewInit } from '@angular/core';

@Component({
  selector: 'sample',
  templateUrl: 'app/sample/sample.component.html',
})
export class SampleComponent implements OnInit, OnDestroy, AfterContentInit, AfterViewInit {
  constructor() {
    console.log('Constructor');
  }
  ngOnInit() {
    console.log('OnInit');
  }
  ngOnDestroy() {
    console.log('OnDestroy');
  }
  ngAfterContentInit() {
    console.log('AfterContentInit');
  }
  ngAfterViewInit() {
    console.log('AfterViewInit');
  }
}
```

ANGULAR FORM

ANGULAR FORM

Forms are used in most web applications as they allow users to submit input when interacting with the application. Among countless use cases, they are useful for logging in users, searching for information and submitting feedback.

Forms are an essential part of many web applications, being the most common way to enter and edit text-based data. Front-end JavaScript frameworks such as Angular, often have their own idiomatic ways of creating and validating forms that you need to get to grips with to be productive.

What is Angular Module?

Every Angular app has a root module, conventionally named AppModule, which provides the bootstrap mechanism that launches the application. An app typically contains many functional modules.

```
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{
  }
```

ANGULAR FORM

Angular provides two approaches, template-driven forms and model-driven or reactive forms, for working with forms:

The template driven approach makes use of built-in directives to build forms such as `ngModel`, `ngModelGroup`, and `ngForm` available from the `FormsModule` module.

The model driven approach of creating forms in Angular 6 makes use of `FormControl`, `FormGroup` and `FormBuilder` available from the `ReactiveFormsModule` module.

With a template driven form, most of the work is done in the template; and with the model driven form, most of the work is done in the component class.

TEMPLATE VS RECTIVE (MODEL DRIVEN) FORM

Template Driven

Based on HTML DOM and Angular Directives.

Use to create Simple Forms

Limit Controls over Form Elements.

Normal Validations.

Reactive Forms

Use to create Complex Forms.

Based on Class Body Programming.

Full Control on Form Elements

Sync with HTML form

Form Validators In Angular

- ✓ Our form is valid all the time, regardless of what input the user types into the controls.
- ✓ Validators are rules which an input control has to follow. If the input doesn't match the rule then the control is said to be invalid.
- ✓ Since it's a signup form most of the fields should be required.
- ✓ We can apply validators either by adding attributes to the template or by defining them on our FormControl in our model.

Form control state

- ✓ Angular comes with a small set of pre-built validators to match the ones we can define via standard HTML5 attributes, namely required, minlength, maxlength and pattern which we can access from the Validators module.
- ✓ The form control instance on our model encapsulates state about the control itself, such as if it is currently valid or if it's been touched.
- ✓ dirty is true if the user has changed the value of the control.
- ✓ The opposite of dirty is pristine. This would be true if the user hasn't changed the value, and false if the user has changed the value.

Form State - Touched & Untouched

- ✓ A controls is said to be touched if the the user focused on the control and then focused on something else. For example by clicking into the control and then pressing tab or clicking on another control in the form.
- ✓ The difference between touched and dirty is that with touched the user doesn't need to actually change the value of the input control.
- ✓ touched is true of the field has been touched by the user, otherwise it's false.
- ✓ The opposite of touched is the property untouched.

Form State - Valid & Invalid

```
<pre>Valid? {{ myform.controls.email.valid }}</pre>
```

- ✓ valid is true if the field doesn't have any validators or if all the validators are passing.
- ✓ Again the opposite of valid is invalid, so we could write:

```
<pre>Invalid? {{ myform.controls.email.invalid }}</pre>
```
- ✓ This would be true if the control was invalid and false if it was valid.

ANGULAR TEMPLATE FORM-DEMO

#App Setup

Here's our file structure:

- | - app/
 - | - app.component.html
 - | - app.component.ts
 - | - app.module.ts
- | - main.ts
- | - user.interface.ts
- | - index.html
- | - styles.css
- | - tsconfig.json

// 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 ], // import forms module here
  declarations: [ AppComponent ],
  bootstrap: [ AppComponent ]
})

export class AppModule { }
```

app.component.html

```
<!-- app.component.html -->
...
<form #f="ngForm" novalidate (ngSubmit)="save(f.value, f.form.valid)">
<!-- we will place our fields here -->
<!--name-->
<div>
<label>Name</label>
<!--bind name to ngModel, it's required with minimum 5 characters-->
<input type="text" name="name" [(ngModel)]="user.name" #name="ngModel" required minlength
    ="5">
<!--show error only when field is not valid & it's dirty or form submitted-->
<small [hidden]="name.valid || (name.pristine && !f.submitted)">
Name is required (minimum 5 characters).
</small>
</div>
```

app.component.html

```
<!--address group-->
<div ngModelGroup="address">
  <!--street-->
  <div>
    <label>Street</label>
    <input type="text" name="street" [(ngModel)]="user.address.street" #street="ngModel"
    required>
    <small [hidden]="street.valid || (street.pristine && !f.submitted)" class="text-danger">
      Street is required.
    </small>
  </div>
  <!--postcode-->
  <div>
    <label>Post code</label>
    <input type="text" name="postcode" [(ngModel)]="user.address.postcode">
  </div>
</div>
<button type="submit">Submit</button>

</form>
```

//The **ngModelGroup** directive allows you to group together related inputs so that you structure the object represented by the form in a useful and predictable way.

ngModelGroup is often used in combination with fieldset as they mostly represent the same idea of “grouping together inputs.”

App.component.ts

```
import { Component, OnInit } from '@angular/core';
import { User } from './user.interface';
@Component({
  //moduleId: module.id,
  selector: 'my-app',
  templateUrl: 'app.component.html',
})
```

App.component.ts

```
export class AppComponent implements OnInit {  
  public user: User; // our model  
  ngOnInit() { // we will initialize our form here  
    this.user = {  
      name: "",  
      address: {  
        street: "",  
        postcode: '8000' // set default value to 8000  
      }  
    };  
  }  
  
  save(model: User, isValid: boolean)  
  {  
    // check if model is valid// if valid, call API to save customer  
    console.log(model, isValid);  
  }  
}
```

Demo

```
// user.interface.ts
```

```
export interface User {  
  name: string; // required with minimum 5 characters  
  address: {  
    street: string; // required  
    postcode: string;  
  }  
}
```

Index.html

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>TestProject</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>
  <my-app></my-app>
</body>
</html>
```

ANGULAR REACTIVE FORM-DEMO

Reactive form validation

- In a reactive form, the source of truth is the component class. Instead of adding validators through attributes in the template, you add validator functions directly to the form control model in the component class. Angular then calls these functions whenever the value of the control changes.

ANGULAR REACTIVE FORM-DEMO

Reactive forms is an Angular technique for creating forms in a reactive style. Reactive forms offer the ease of using reactive patterns, testing, and validation.

ANGULAR REACTIVE FORM

- First, we need to import some of the modules from the **@angular/forms**.
- FormGroup
- FormBuilder
- Validators
- Also, we need have imported **ReactiveFormsModule** in **app.module.ts** file.

ANGULAR REACTIVE FORM :

FormGroup is one of the three fundamental building blocks used to define forms in **Angular**, along with FormControl and FormArray

The **.form-group** class of bootstrap is the easiest way to add some structure to **forms**.

```
<form [formGroup]="userForm" (ngSubmit)="onFormSubmit()">  
  Name: <input formControlName="name" placeholder="Enter Name">  
  Age: <input formControlName="age" placeholder="Enter Age">  
  <button type="submit">Submit</button>  
</form>
```


ANGULAR REACTIVE FORM :

A FormGroup aggregates the values of each child FormControl into one object, with each control name as the key. It calculates its status by reducing the status values of its children. For example, if one of the controls in a group is invalid, the entire group becomes invalid.

ANGULAR REACTIVE FORM :

Angular Form Builder . **Angular** has a new helper Class called **FormBuilder** . **FormBuilder** allows us to explicitly declare forms in our components. It is also used for the validation management with angular reactive form.

We use **FormBuilder to Create & Manage Forms** .

FormBuilder creates reactive form with minimum code using FormGroup, FormControl and FormArray.

ANGULAR REACTIVE FORM :

FormBuilder creates reactive form with minimum code using **FormGroup**, **FormControl**.

The FormBuilder class helps you create controls.

Inside the src/app/ap.component.ts file import the FormBuilder class from the @angular/forms package:

```
import { Component } from '@angular/core';  
import { FormControl, FormGroup, FormBuilder } from  
  '@angular/forms';
```

ANGULAR REACTIVE FORM

```
// app.component.ts
```

```
constructor(private fb: FormBuilder) {  
    this.createForm();  
}
```

We have used form builder to handle all the validation. So in that constructor, we are creating a form with the validation rules.

ANGULAR REACTIVE FORM : Validators

Provides a set of built-in validators that can be used by form controls.

A validator is a function that processes a FormControl or collection of controls and returns an error map or null. A null map means that validation has passed.

Validators functions :

`required()`

Validator that requires the control have a non-empty value.

ANGULAR REACTIVE FORM

The `formGroup`, `formControlName` and `formGroupName` are directives to bind our form model to our HTML form.

We use the `formGroup` property in the `<form>` tag to bind the form with `exampleForm` form group and we use the `formControlName` property to bind the `<input>` tag to individual form controls.

Validation rules to the Form Object.

<!-- app.component.html -->

```
<div class="container">
```

```
  <h1>
```

```
    Welcome to {{title}}!!
```

```
  </h1>
```

```
  <form [formGroup]="angForm" novalidate>
```

```
    <div class="form-group">
```

```
      <label>Name:</label>
```

```
      <input class="form-control" formControlName="name" type="text">
```

```
    </div>
```

```
    <div *ngIf="angForm.controls['name'].invalid && (angForm.controls['name'].dirty || angForm.controls['name'].touched)"
```

```
      class="alert alert-danger">
```

```
        <div *ngIf="angForm.controls['name'].errors.required">
```

```
          Name is required.
```

```
        </div>
```

Validation rules to the Form Object.

</div>

```
<div class="form-group">
```

```
  <label>Address:</label>
```

```
  <input class="form-control" formControlName="address" type="text">
```

```
</div>
```

```
<div *ngIf="angForm.controls['address'].invalid && (angForm.controls['address'].dirty || angForm.controls['address'].touched)"
```

```
  class="alert alert-danger">
```

```
    <div *ngIf="angForm.controls['email'].errors.required">
```

```
      email is required.
```

```
    </div>
```

```
</div>
```

```
<button type="submit"
```

```
  [disabled]="angForm.pristine || angForm.invalid" class="btn btn-success">
```

```
    Save
```

```
</button>
```

```
</form>
```

```
<br />
```

```
<p>Form value: {{ angForm.value | json }}</p>
```

```
<p>Form status: {{ angForm.status | json }}</p>
```

```
</div>
```


app.component.ts file looks like this.

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

import { FormGroup, FormBuilder, Validators } from '@angular/forms';

@Component({
  selector: 'app-root',

  templateUrl: './app.component.html',

  styleUrls: ['./app.component.css']
})

export class AppComponent {

  title = 'Angular Form Validation';

  angForm: FormGroup;

  constructor(private fb: FormBuilder) {

    this.createForm();

  }

  createForm() {

    this.angForm = this.fb.group({

      name: ['', Validators.required ],

      address: ['', Validators.required ]

    });

  }

}
```

app.module.ts file looks like this.

```
import { BrowserModule } from '@angular/platform-browser';
```

```
import { NgModule } from '@angular/core';
```

```
import { ReactiveFormsModule } from '@angular/forms';
```

```
import { AppComponent } from './app.component';
```

```
  @NgModule({
```

```
    declarations: [
```

```
      AppComponent
```

```
    ],
```

```
    imports: [
```

```
      BrowserModule, ReactiveFormsModule
```

```
    ],
```

```
    providers: [],
```

```
    bootstrap: [AppComponent]
```

```
  })
```

```
  export class AppModule { }
```

Validation rules to the Form Object.

Welcome to Angular Form Validation



Name:

Name is required.

Form value: { "name": "" }

Form status: "INVALID"

Template Driven Forms Features

- Easy to use
- Suitable for simple scenarios and fails for complex scenarios
- Similar to AngularJS
- Two way data binding(using `[(NgModel)]` syntax)
- Minimal component code
- Automatic track of the form and its data(handled by Angular)

Reactive Forms Features

- More flexible, but needs a lot of practice
- Handles any complex scenarios
- No data binding is done (immutable data model preferred by most developers)
- More component code and less HTML markup
- Reactive transformations can be made possible such as
 - Adding elements dynamically

Conclusion

Both reactive and template driven forms provide efficient ways for building forms in Angular. While the template driven syntax offers a more familiar approach, reactive forms offer a more dynamic way of building form groups and controls.

ANGULAR ROUTING AND SINGLE PAGE APPLICATION

What is the Angular 2 Router?

The router enables navigation from one view to the next as users perform application tasks.

Angular is known for single page applications (SPA)

What is single page application?

- a web application that *fits on a single web page*
- providing a more fluid *user experience similar to a desktop application*

Condition:

- all necessary code is retrieved with a single page load
- the page does not reload at any point in the process
- does not control transfer to another page

SINGLE PAGE APPLICATION

- A single-page application (SPA), is defined as a web application or web site that fits on a single web page with the goal of providing a more fluid user experience akin to a desktop application. In a SPA, either all necessary code – HTML, JavaScript, and CSS – is retrieved with a single page load or the appropriate resources are dynamically loaded and added to the page as necessary, usually in response to user actions.

SPA BENEFITS

- ❑ Runs faster
- ❑ Gives better UX
- ❑ Uses less network bandwidth

What is Angular Router?

The Angular Router enables navigation from one [view](#) to the next as users perform application tasks.

Routes tell the router which view to display when a user clicks a link or pastes a URL into the browser address bar. A typical Angular Route has two properties:

path: a string that matches the URL in the browser address bar.

component: the component that the router should create when navigating to this route.

To tell the paths and destinations to your browser, you need to import the RouterModule and Routes from `@angular/router` into your `app.module.ts` file.

```
import {RouterModule, Routes} from '@angular/router';
```

IMPORTANT TERMS

ModuleWithProviders

A wrapper (interface) around an NgModule that associates it with the providers.

Routes and Paths

In Angular, a **route** is an object (instance of Route) that provides information about which component maps to a specific path. A **path** is the fragment of a URL that determines where exactly is located the resource (or page) you want to access. You can get the path by taking off the domain name from the URL.

RouterModule

Adds router directives and providers.

IMPORTANT TERMS

RouterLink

Lets you link to specific routes in your app

```
<a routerLink="/calc">Go to Calc</a>
```

The routerLink is the selector for the RouterLink directive that turns user clicks into router navigations.

RouterLinkActive

Lets you add a CSS class to an element when the link's route becomes active.

```
<nav>
```

```
  <a routerLink="/calc" routerLinkActive="active">Go to Calc</a>
```

```
</nav>
```

IMPORTANT TERMS

forRoot()

Creates a module with all the router providers and directives. It also optionally sets up an application listener to perform an initial navigation.

The method is called `forRoot()` because you configure the router at the application's root level. The `forRoot()` method supplies the service providers and directives needed for routing, and performs the initial navigation based on the current browser URL.

RouterOutlet

Acts as a placeholder that Angular dynamically fills based on the current router state.

<router-outlet></router-outlet>

The `<router-outlet>` tells the router where to display routed views.

The RouterOutlet is one of the router directives that became available to the AppComponent because AppModule imports AppRoutingModuleModule which exported RouterModule.

IMPORTANT TERMS

We then define an array of routes which is of type `Routes` then use `RouterModule.forRoot` to export the routes so it can be injected in our application when bootstrapping.

ANGULAR 2

HOW TO CREATE ANGULAR ROUTING/SPA

Angular Router?

Step 1: create project

- ng new web-router
- ng serve --open

Step 2: Create 2 Components

- ng generate component about
- ng g c services
- ng serve --open

Step 3: Create an app.router.ts file

create a route module to store our routing information specifically. Place it within the /app folder.

app.router.ts

```
import { ModuleWithProviders } from '@angular/core';  
import { Routes, RouterModule } from '@angular/router'; //module for  
    implementing routing  
  
import { AppComponent } from './app.component';  
import { AboutComponent } from './about/about.component';  
import { ServicesComponent } from './services/services.component';
```

app.router.ts

// Create and export a variable called Router in the app.routing.ts, which would be a collection of all routes inside the Angular application.

// **pathMatch** is specifying a strict matching of path to reach the destination page.

// In the routes array, the path object is "" which means the path http://localhost:4200/, we have not provided any destination to this object, but we are redirecting this path to items path using redirectTo key.

```
export const router: Routes = [  
  { path: '', redirectTo: 'about', pathMatch: 'full' }, //default page  
  { path: 'about', component: AboutComponent },  
  { path: 'services', component: ServicesComponent }  
];
```

// RouterModule.forRoot is for creating routes for the entire application

// pass the routes array in the RouterModule using RouterModule.forRoot(routes)

```
export const routes: ModuleWithProviders = RouterModule.forRoot(router);
```

app.router.ts

In a nutshell, we're importing the necessary routing members on the first 2 lines. Then we're importing the components that currently exist within our application.

After that, we're defining a constant that contains the routes that we wish to create. "path" defines the URL; so "about"

becomes <http://someurl.com/about> -- then we specify the component associated with that view.

app.module.ts

Step 3: Import the router to app.module.ts

```
import { routes } from './app.router';
import { AppComponent } from './app.component';
import { AboutComponent } from './about/about.component';
import { ServicesComponent } from './services/services.component';
@NgModule({
  declarations: [
    AppComponent,
    AboutComponent,
    ServicesComponent
  ],
  imports: [
    BrowserModule,
    routes
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

app.component.html

Step 4:

<!--Place our rendered AppComponent -- >

Now within app.component.html, we add:

```
<div>
```

```
<router-outlet></router-outlet>
```

```
</div>
```

```
<div>
```

```
<ul class="nav navbar-nav">
```

```
<li>
```

```
<a routerLink="about">About</a>
```

```
</li>
```

```
<li>
```

```
<a routerLink="services">Services</a>
```

```
</li>
```

Angular Router

Step 6: Ensure `<base href="/">` is in index.html

If you used the angular-cli to create the project you're working with, it will automatically add this in the head tag. If it's not there, add it:

```
<base href="/">
```


COMPONENT INTERECTION

In Angular, a component can share data and information with another component by passing data or events. A component can be used inside another component, thus creating a component hierarchy. The component being used inside another component is known as the child component and the enclosing component is known as the parent component.

COMPONENT INTERECTION

Components can communicate to each other in various ways, including:

Using @Input()

Using @Output()