ANGULAR 4

Agenda

- Introduction of Angular
- Components
- Directives
- Data binding
- Pipes
- Services
- Routing
- Modules
- Reactive Extensions
- Testing

Introduction of Angular

- Features of Angular
- Development environment setup.
- Project setup
- Structure of Angular project.
- How to install packages.
- How Angular application start.

Features

- Angular is an open source JavaScript framework that is used to build single page based web applications.
- Developed by Google
- One framework - Mobile & Desktop.
- Use modern web platform capabilities to deliver app-like experiences.
- Build native mobile apps with strategies from Ionic Framework, NativeScript, and React Native.
- Create desktop-installed apps across Mac, Windows, and Linux

Development environment setup

- Install Node.js: Node.js is an open-source, cross-platform JavaScript run-time environment
  - Url: https://nodejs.org/en/download/
- Checking Node version: node –v
- Installing Angular-cli:

  - npm install --global @angular/cli@1.0.0

  Or

  - npm install --global @angular/cli

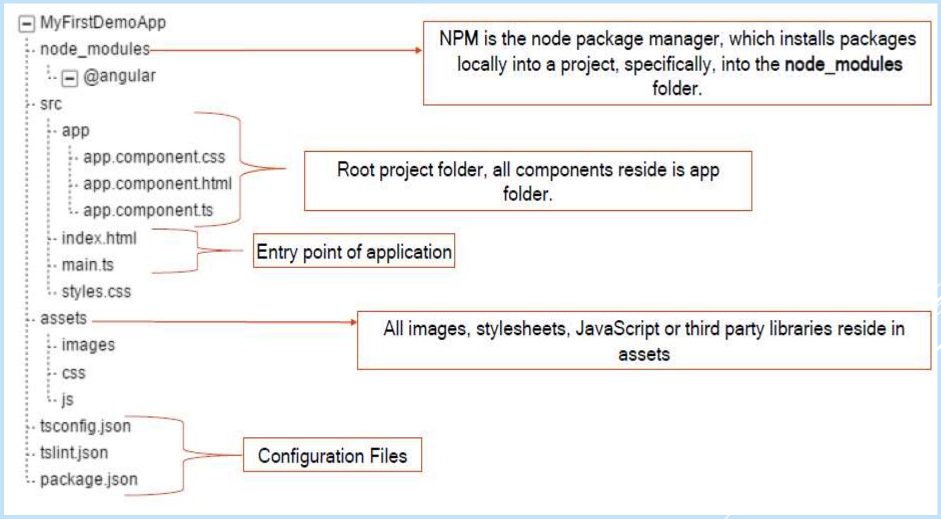
Project setup

- Open command prompt/terminal.
- Change your working directory

  - cd <directory-path>
- Create a new project new Angular-cli

  - ng new <project-name>

Structure of the Project

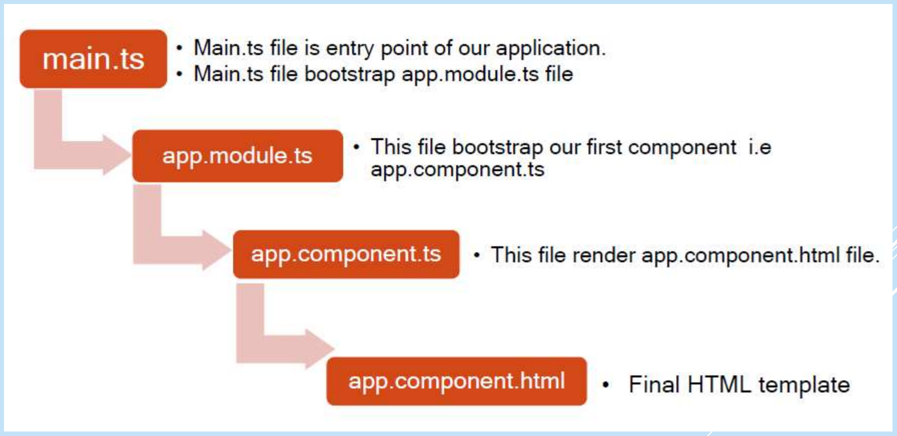

MyFirstDemoApp
node_modules
@angular
src
app
app.component.css
app.component.html
app.component.ts
index.html
main.ts
styles.css
assets
images
css
js
tsconfig.json
tslint.json
package.json
NPM is the node package manager, which installs packages locally into a project, specifically, into the **node_modules** folder.
Root project folder, all components reside is app folder.
Entry point of application
All images, stylesheets, JavaScript or third party libraries reside in assets
Configuration Files

Installing Packages & Serving app

- Installing any additional/third party required packages
  - npm install <package-name > --save
- To compile and execute app
  - ng serve –o // will run on default port:4200

  change port number
  - ng serve –o –port <port-number>

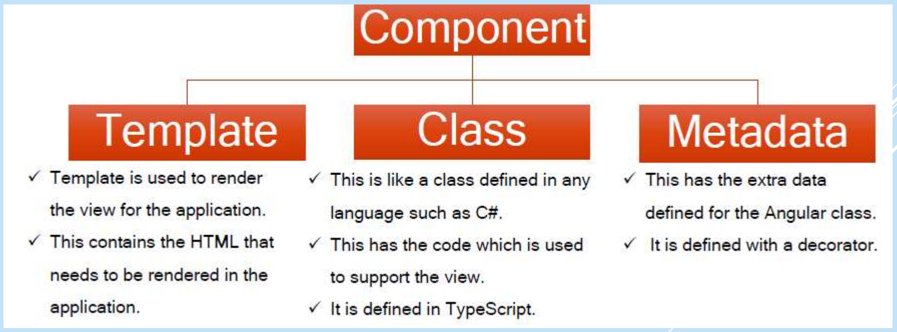
How Angular application start



- Main.ts file is entry point of our application.
- Main.ts file bootstrap app.module.ts file
- This file bootstrap our first component i.e app.component.ts
- This file render app.component.html file.
- Final HTML template

Component

- An Angular component is responsible for managing a template and providing it with the data and logic it needs.
- A component consists of the following:

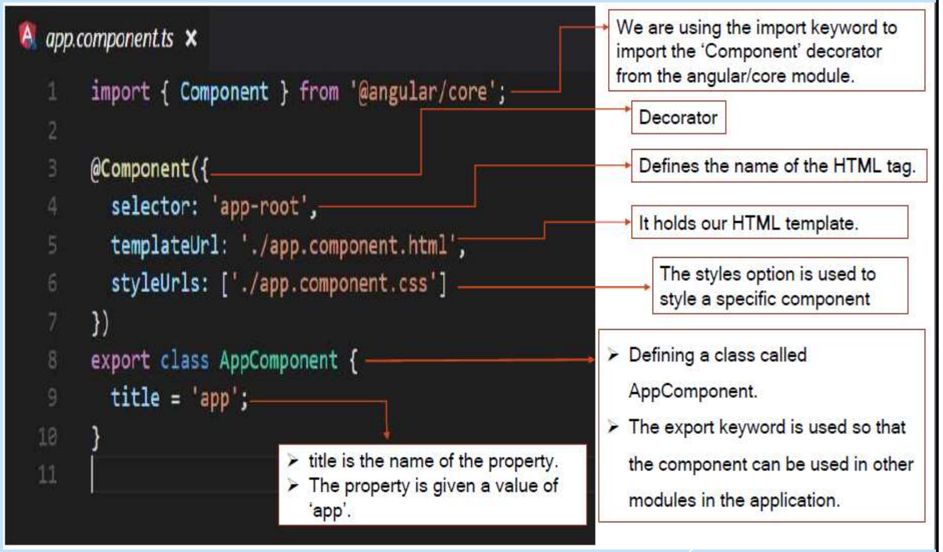


Template
- ✓ Template is used to render the view for the application.
- ✓ This contains the HTML that needs to be rendered in the application.

Class
- ✓ This is like a class defined in any language such as C#.
- ✓ This has the code which is used to support the view.
- ✓ It is defined in TypeScript.

Metadata
- ✓ This has the extra data defined for the Angular class.
- ✓ It is defined with a decorator.

Component continue…



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

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

We are using the import keyword to import the 'Component' decorator from the angular/core module.

Decorator

Defines the name of the HTML tag.

It holds our HTML template.

The styles option is used to style a specific component

- Defining a class called AppComponent.
- The export keyword is used so that the component can be used in other modules in the application.

- title is the name of the property.
- The property is given a value of 'app'.

Creating Component

- ng generate command is used to generate a component using Angular-cli
  - ng generate component <component-name>

  Short hand syntax
  - ng g c <component-name>
- Angular component have 3 basic parts:
  - <component-name>.html (the html code of the component)
  - <component-name>.ts (the logic of the component)
  - <component-name>.css (css style of the component)

Directives

There are three kinds of directives in Angular:

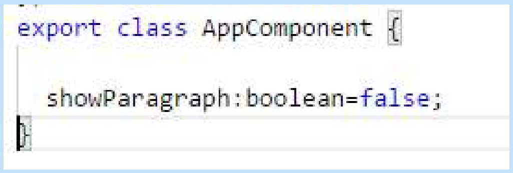
- Components—directives with a template.
- Structural directives—change the DOM layout by adding and removing DOM elements.
- Attribute directives—change the appearance or behavior of an element, component, or another directive.

Structural directives

- Structural directives are easy to recognize. An asterisk (*) precedes the directive attribute name.
- Angular has the following built-in structural directives:
    - ngIf
    - ngFor
    - ngSwitch

*ngIf

- It takes a boolean expression and makes an entire chunk of the DOM appear or disappear.

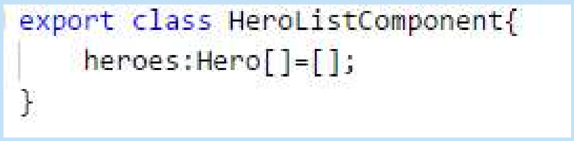


```
export class AppComponent {

  showParagraph:boolean=false;
}
```

```
<p *ngIf="showParagraph">
  Expression is false and ngIf is false.
  This paragraph is not in the DOM.
</p>
```

*ngFor

- It is a *repeater* directive — a way to present a list of items. You define a block of HTML that defines how a single item should be displayed.

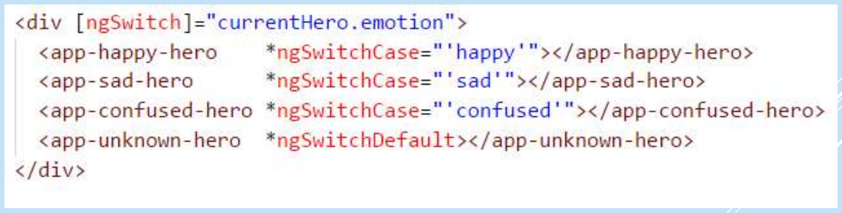


```
export class HeroListComponent{
    heroes:Hero[]=[];
}
```

```
<div *ngFor="let hero of heroes">{{hero.name}}</div>
```

*ngSwitch

- It is like JavaScript switch statement. It can display *one* element from among several possible elements, based on a *switch condition*. Angular puts only the *selected* element into the DOM.

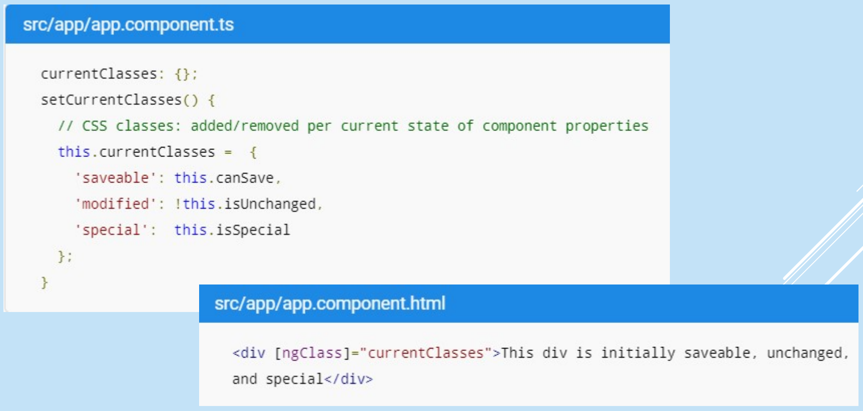

```
<div [ngSwitch]="currentHero.emotion">
  <app-happy-hero    *ngSwitchCase="'happy'"></app-happy-hero>
  <app-sad-hero      *ngSwitchCase="'sad'"></app-sad-hero>
  <app-confused-hero *ngSwitchCase="'confused'"></app-confused-hero>
  <app-unknown-hero  *ngSwitchDefault></app-unknown-hero>
</div>
```

Attribute directives

- Attribute directives listen to and modify the behavior of other HTML elements, attributes, properties, and components. They are usually applied to elements as if they were HTML attributes, hence the name.
- Angular has the following built-in attribute directives:
  - ngClass
  - ngStyle

[ngClass]

- You typically control how elements appear by adding and removing CSS classes dynamically. You can bind to the ngClass to add or remove several classes simultaneously.



src/app/app.component.ts

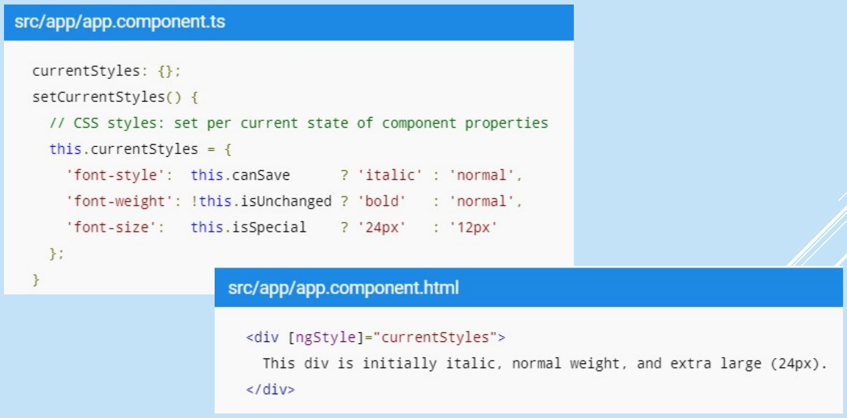
```
currentClasses: {};
setCurrentClasses() {
  // CSS classes: added/removed per current state of component properties
  this.currentClasses =  {
    'saveable': this.canSave,
    'modified': !this.isUnchanged,
    'special':  this.isSpecial
  };
}
```

src/app/app.component.html

```
<div [ngClass]="currentClasses">This div is initially saveable, unchanged,
and special</div>
```

[ngStyle]

- You can set inline styles dynamically, based on the state of the component. With ngStyle you can set many inline styles simultaneously.



src/app/app.component.ts

```
currentStyles: {};
setCurrentStyles() {
  // CSS styles: set per current state of component properties
  this.currentStyles = {
    'font-style':  this.canSave      ? 'italic' : 'normal',
    'font-weight': !this.isUnchanged ? 'bold'   : 'normal',
    'font-size':   this.isSpecial    ? '24px'   : '12px'
  };
}
```

src/app/app.component.html

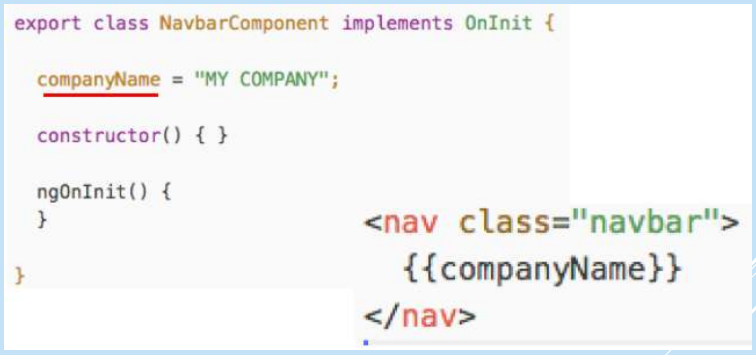
```
<div [ngStyle]="currentStyles">
  This div is initially italic, normal weight, and extra large (24px).
</div>
```

Data binding

- Data binding is communication between business logic and views.
- There are primarily 3 ways of data binding in Angular
  - One way from business logic(component.ts) to View
  - One way from View to business logic(component.ts)
  - Two-way data binding

String Interpolation

- Allows us to read primitive or object values from component properties in the template (html file)

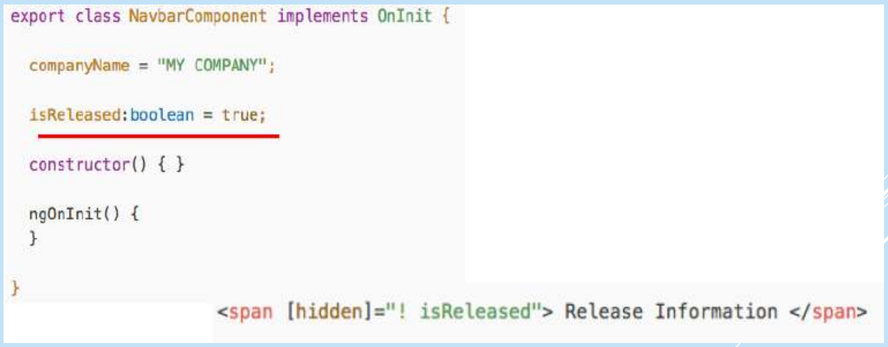


```
export class NavbarComponent implements OnInit {

  companyName = "MY COMPANY";

  constructor() { }

  ngOnInit() {
  }

}
```

```
<nav class="navbar">
  {{companyName}}
</nav>
```

Property binding

- Angular executes the expression and assigns it to a property of an HTML element, a component, or a directive.

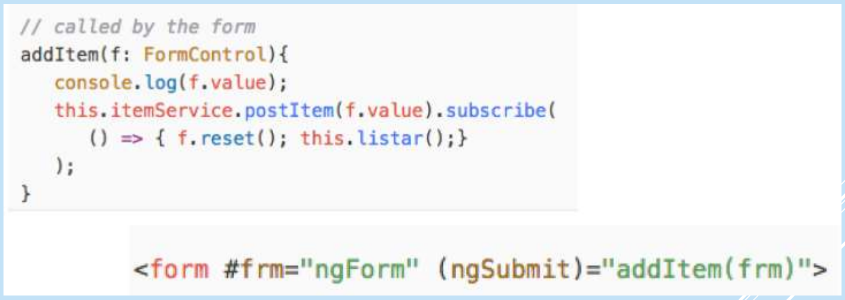


```
export class NavbarComponent implements OnInit {

  companyName = "MY COMPANY";

  isReleased:boolean = true;

  constructor() { }

  ngOnInit() {
  }

}
```

```
<span [hidden]="! isReleased"> Release Information </span>
```

Event binding

- A component method responds to an event raised by an element, component, or directive.

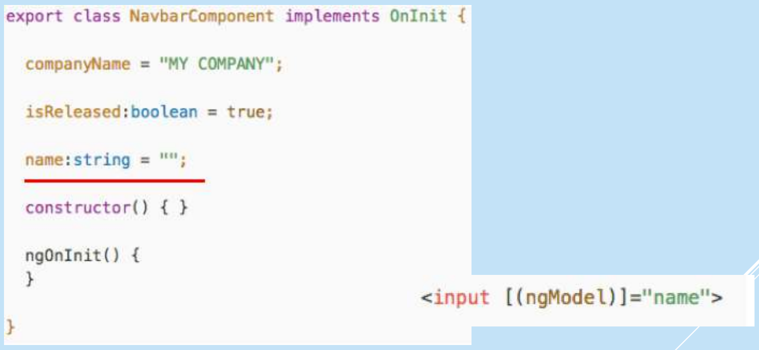


```
// called by the form
addItem(f: FormControl){
   console.log(f.value);
   this.itemService.postItem(f.value).subscribe(
      () => { f.reset(); this.listar();}
   );
}
```

```
<form #frm="ngForm" (ngSubmit)="addItem(frm)">
```

Two-way data binding

- Its takes a combination of setting a specific element property and listening for an element change event.
- Implemented using ngModel directive



```
export class NavbarComponent implements OnInit {

  companyName = "MY COMPANY";

  isReleased:boolean = true;

  name:string = "";

  constructor() { }

  ngOnInit() {
  }
}
```

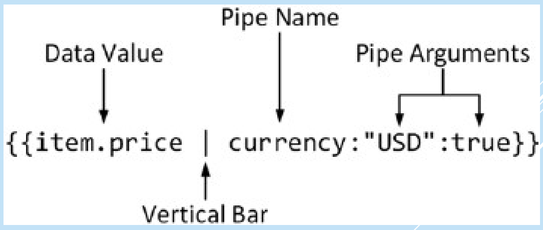
```
<input [(ngModel)]="name">
```

Pipes

- Pipes are small fragments of code that transform data values so they can be displayed to the user in templates.
- Pipes allow transformation logic to be defined in self-contained classes so that it can be applied consistently throughout an application.
- The @Pipe decorator is applied to a class and used to specify a name by which the pipe can be used in a template.

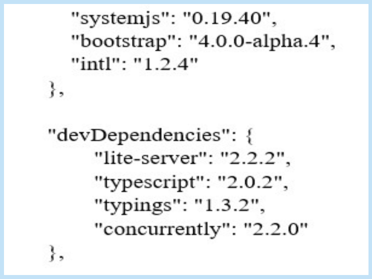
Using Pipes

- Angular has various built-in pipes:
  - Number pipe
  - Currency pipe
  - Percent pipe
  - Date pipe
  - Uppercase or lowercase pipe
  - Json pipe
  - Slice pipe
  - Async pipe



Installing the Internationalization Polyfill

- Some of the built-in pipes that Angular provides rely on the Internationalization API, which provides location-sensitive string comparisons and formatting for numbers, dates, and times.
- The most complete polyfill for the Internationalization API is Intl.js



```
    "systemjs": "0.19.40",
    "bootstrap": "4.0.0-alpha.4",
    "intl": "1.2.4"
  },

  "devDependencies": {
      "lite-server": "2.2.2",
      "typescript": "2.0.2",
      "typings": "1.3.2",
      "concurrently": "2.2.0"
  },
```

Creating a Custom Pipe

```
import { Pipe } from "@angular/core";

@Pipe({
    name: "addTax"
})
export class PaAddTaxPipe {

    defaultRate: number = 10;

    transform(value: any, rate?: any): number {
        let valueNumber = Number.parseFloat(value);
        let rateNumber = rate == undefined ?
            this.defaultRate : Number.parseInt(rate);
        return valueNumber + (valueNumber * (rateNumber / 100));
    }
}
```

Registering & applying a Custom Pipe

- Pipes are registered using the declarations property of the Angular module.

```
@NgModule({
    imports: [BrowserModule, FormsModule, ReactiveFormsModule],
    declarations: [ProductComponent, PaAttrDirective, PaModel,
        PaStructureDirective, PaIteratorDirective,
        PaCellColor, PaCellColorSwitcher, ProductTableComponent,
        ProductFormComponent, PaToggleView, PaAddTaxPipe],
```

- Once a custom pipe has been registered, it can be used in data binding expressions.

```
<td style="vertical-align:middle">
    {{item.price | addTax:(taxRate || 0) }}
</td>
```

Combining Pipes

- Pipes are chained together using the vertical bar character, and the names of the pipes are specified in the order that data should flow.

```
<td style="vertical-align:middle">
      {{item.price | addTax:(taxRate || 0) | currency:"USD":true                    }}
</td>
```

Using the Built-in Pipes

- Formatting Numbers

```
<td style="vertical-align:middle">{{item.price | number:"3.2-2" }}</td>
```

- Formatting Currency Values

```
{{item.price | currency:"USD":true:"2.2-2" }}
```

- Formatting Percentages

```
<option value="10">{{ 0.1 | percent }}</option>
```

- Formatting Dates

```
<div>Date formatted from object: {{ dateObject | date }}</div>
```

Using the Built-in Pipes

- Changing String Case

```
<td  style="vertical-align:middle">{{item.name | uppercase }}</td>
<td  style="vertical-align:middle">{{item.category | lowercase }}</td>
```

- Serializing Data as JSON

```
<div>{{ getProducts() | json }}</div>
```

- Slicing Data Arrays

```
<tr *paFor="let item of getProducts() | slice:0:(itemCount || 1);
```

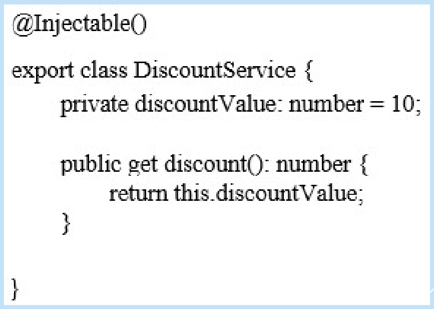
Services

- *Services* are objects that provide common functionality to support other building blocks in an application, such as directives, components, and pipes.
- Services are used through a process called *dependency injection*.
- Services are classes to which the @Injectable decorator has been applied.

Creating Service

- Services can easily be created from Angular-cli using following command:
  - ng generate service <service-name>

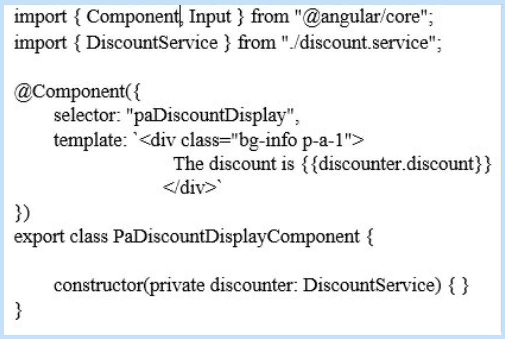
  Short hand syntax:
  - ng g s <service-name>



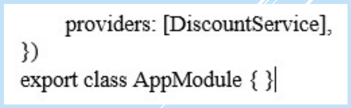
```
@Injectable()
export class DiscountService {
    private discountValue: number = 10;

    public get discount(): number {
        return this.discountValue;
    }

}
```

Consuming Service

- To consume service in a component, it needs to be injected inside component constructor.
- Register the service in Angular module



```
import { Component, Input } from "@angular/core";
import { DiscountService } from "./discount.service";

@Component({
    selector: "paDiscountDisplay",
    template: `<div class="bg-info p-a-1">
                    The discount is {{discounter.discount}}
               </div>`
})
export class PaDiscountDisplayComponent {

    constructor(private discounter: DiscountService) { }
}
```



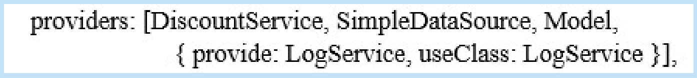
```
    providers: [DiscountService],
})
export class AppModule { }
```

Using Service Providers

- When using dependency injection, the objects that are used to resolve dependencies are created by *service providers*, known more commonly as *providers*.

Using the Class Provider

- This provider is the most commonly used. This is the provider that is targeted by adding the class names to the module’s providers property.
- This is a shorthand syntax. There is also a literal syntax that achieves the same result.



```
providers: [DiscountService, SimpleDataSource, Model,
                    { provide: LogService, useClass: LogService }],
```

Using the Value Provider

- The value provider is used when you need to take responsibility for creating the service objects yourself, rather than leaving it to the class provider. This can also be useful when services are simple types, such as string or number values.

```
let logger = new LogService();
```

App.module.ts

```
providers: [DiscountService, SimpleDataSource, Model,
        { provide: LogService, useValue: logger }],
```

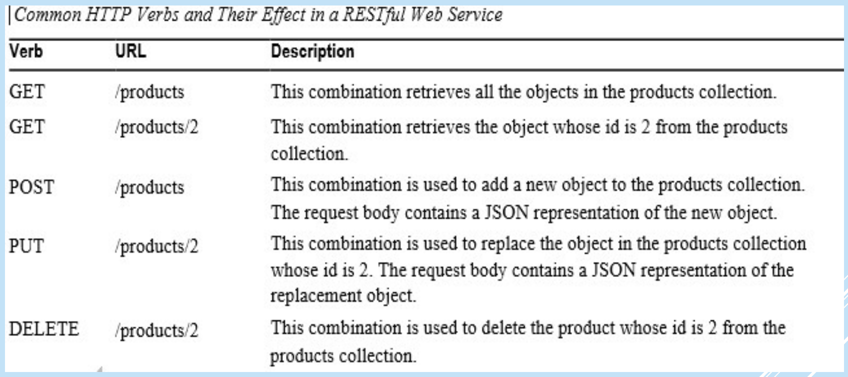
Making Asynchronous HTTP Requests

- Asynchronous HTTP requests allow Angular applications to interact with web services so that persistent data can be loaded into the application and changes can be sent to the server and saved.
- Requests are made using the Http class, which is delivered as a service through dependency injection.
- Angular HTTP feature requires the use of Reactive Extensions Observable objects.

Understanding Restful Web Services

- The most common approach for delivering data to an application is to use the *Representational State Transfer* pattern, known as REST, to create a data web service.
- The core premise of a RESTful web service is to embrace the characteristics of HTTP so that request methods—also known as *verbs*—specify an operation for the server to perform, and the request URL specifies one or more data objects to which the operation will be applied.

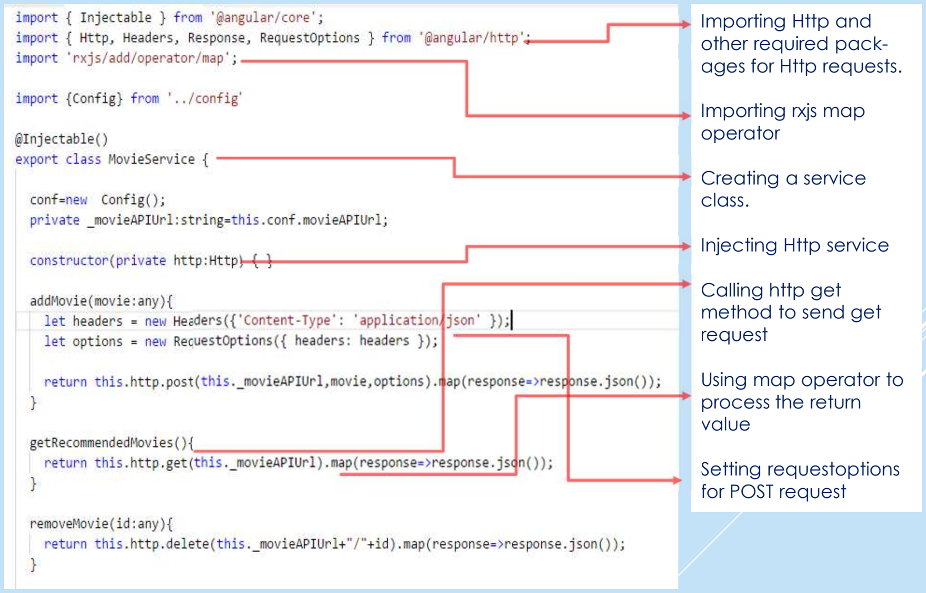
Understanding Restful Web Services



Common HTTP Verbs and Their Effect in a RESTful Web Service

Verb	URL	Description
GET	/products	This combination retrieves all the objects in the products collection.
GET	/products/2	This combination retrieves the object whose id is 2 from the products collection.
POST	/products	This combination is used to add a new object to the products collection. The request body contains a JSON representation of the new object.
PUT	/products/2	This combination is used to replace the object in the products collection whose id is 2. The request body contains a JSON representation of the replacement object.
DELETE	/products/2	This combination is used to delete the product whose id is 2 from the products collection.

Making HTTP Requests to REST service



```
import { Injectable } from '@angular/core';
import { Http, Headers, Response, RequestOptions } from '@angular/http';
import 'rxjs/add/operator/map';

import {Config} from '../config'

@Injectable()
export class MovieService {

  conf=new  Config();
  private _movieAPIUrl:string=this.conf.movieAPIUrl;

  constructor(private http:Http) { }

  addMovie(movie:any){
    let headers = new Headers({'Content-Type': 'application/json' });
    let options = new RequestOptions({ headers: headers });

    return this.http.post(this._movieAPIUrl,movie,options).map(response=>response.json());
  }

  getRecommendedMovies(){
    return this.http.get(this._movieAPIUrl).map(response=>response.json());
  }

  removeMovie(id:any){
    return this.http.delete(this._movieAPIUrl+"/"+id).map(response=>response.json());
  }
```

Importing Http and other required packages for Http requests.

Importing rxjs map operator

Creating a service class.

Injecting Http service

Calling http get method to send get request

Using map operator to process the return value

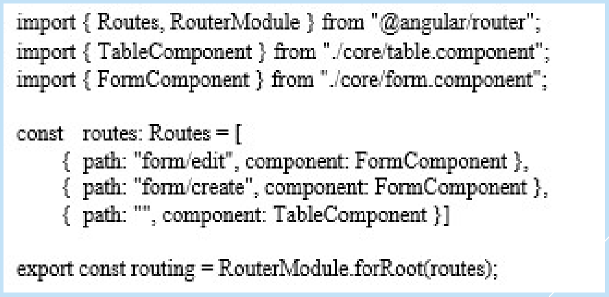
Setting requestoptions for POST request

Routing

- Routing is a functionality that helps your application to become a Single Page Application
- It redirect the user to another component without reload the page.
- Routing uses the browser's URL to manage the content displayed to the user.
- Routing allows the structure of an application to be kept apart from the components and templates in the application. Changes to the structure of the application are made in the routing configuration rather than in individual components and directives.

Creating a Routing Configuration

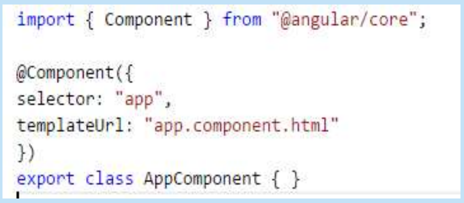
- The first step when applying routing is to define the *routes*, which are mappings between URLs and the components that will be displayed to the user. Routing configurations are conventionally defined in a file called app.routing.ts, defined in the app folder.



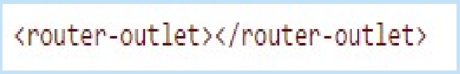
```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";

const   routes: Routes = [
      { path: "form/edit", component: FormComponent },
      { path: "form/create", component: FormComponent },
      { path: "", component: TableComponent }]

export const routing = RouterModule.forRoot(routes);
```

Creating the routing component

- When using routing, the root component is dedicated to managing the navigation between different parts of the application.



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

@Component({
selector: "app",
templateUrl: "app.component.html"
})
export class AppComponent { }
```

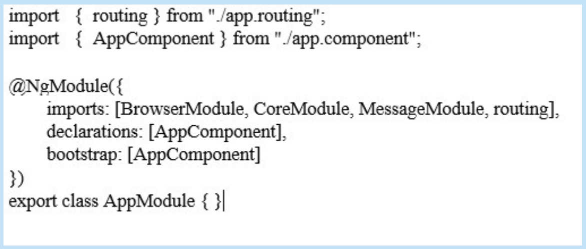


```
<router-outlet></router-outlet>
```

- For the purposes of routing, it is the router-outlet element—known as the outlet—that is important because it tells Angular that this is where the component matched by the routing configuration should be displayed.

Updating the Root Module

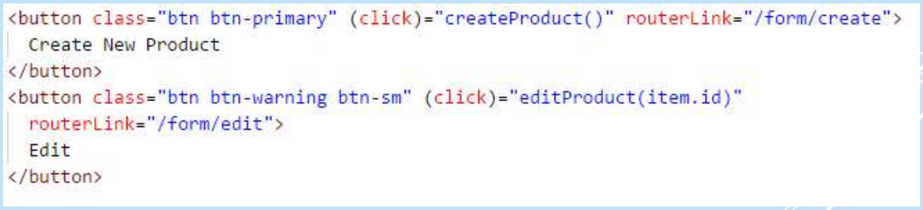
- The next step is to update the root module so that the routes defined inside app.routing.ts file can be used in the application.



```
import   {  routing } from "./app.routing";
import   {  AppComponent } from "./app.component";

@NgModule({
        imports: [BrowserModule, CoreModule, MessageModule, routing],
        declarations: [AppComponent],
        bootstrap: [AppComponent]
})
export class AppModule { }
```

Adding Navigation Links

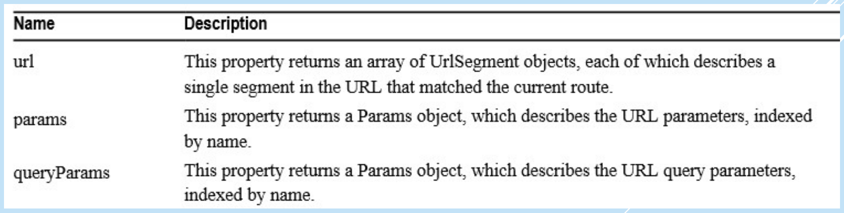
- Add links to the application that will change the browser's URL and, in doing so, trigger a routing change that will display a different component to the user.



```
<button class="btn btn-primary" (click)="createProduct()" routerLink="/form/create">
  Create New Product
</button>
<button class="btn btn-warning btn-sm" (click)="editProduct(item.id)"
  routerLink="/form/edit">
  Edit
</button>
```

Handling Route Changes in Components

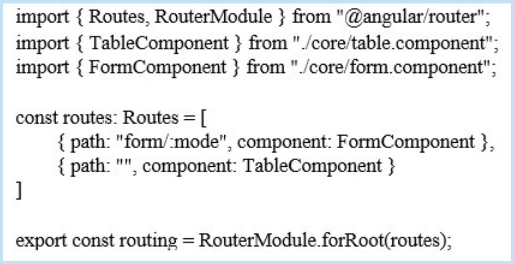
- Angular provides a service ActivatedRoute , that components can receive to get details of the current route.
- The snapshot property returns an instance of the ActivatedRouteSnapshot class, which provides information about the route that led to the current component being displayed to the user using the properties described in Table



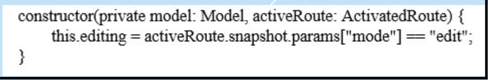
Name	Description
url	This property returns an array of UrlSegment objects, each of which describes a single segment in the URL that matched the current route.
params	This property returns a Params object, which describes the URL parameters, indexed by name.
queryParams	This property returns a Params object, which describes the URL query parameters, indexed by name.

Using Route Parameters

- Angular routes can be more flexible and include route *parameters*, which allow any value for a segment to match the corresponding segment in the navigated URL. This means routes that target the same component with similar URLs can be consolidated into a single route.



```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "./core/table.component";
import { FormComponent } from "./core/form.component";

const routes: Routes = [
      { path: "form/:mode", component: FormComponent },
      { path: "", component: TableComponent }
]

export const routing = RouterModule.forRoot(routes);
```



```
constructor(private model: Model, activeRoute: ActivatedRoute) {
      this.editing = activeRoute.snapshot.params["mode"] == "edit";
}
```

Using Multiple Route Parameters

- This cycle of consolidating and then expanding routes is typical of most development projects as you increase the amount of information that is included in routed URLs to add functionality to the application.

```
const   routes: Routes = [
      {  path: "form/:mode/:id", component: FormComponent },
      {  path: "form/:mode", component: FormComponent },
      {  path: "", component: TableComponent }]
```

```
constructor(private model: Model, activeRoute: ActivatedRoute) {
      this.editing = activeRoute.snapshot.params["mode"] == "edit";
      let id = activeRoute.snapshot.params["id"];
      if (id != null) {
            Object.assign(this.product, model.getProduct(id) || new Product());
      }
}
```

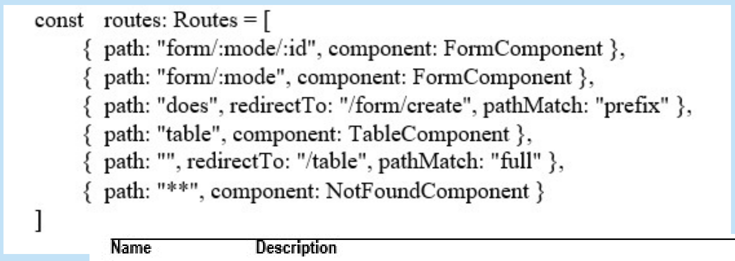
Using Wildcards

- The Angular routing system supports a special path, denoted by two asterisks (the ** characters), that allows routes to match any URL.
- The basic use for the wildcard path is to deal with navigation that would otherwise create a routing error.

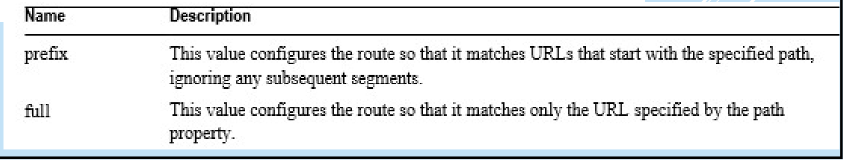
```
const   routes: Routes = [
      {  path: "form/:mode/:id", component: FormComponent },
      {  path: "form/:mode", component: FormComponent },
      {  path: "does", redirectTo: "/form/create", pathMatch: "prefix" },
      {  path: "table", component: TableComponent },
      {  path: "", redirectTo: "/table", pathMatch: "full" },
      {  path: "**", component: NotFoundComponent }
]
```

Using Redirections in Routes

- Routes do not have to select components; they can also be used as aliases that redirect the browser to a different URL.
- Redirections are defined using the redirectTo property in a route



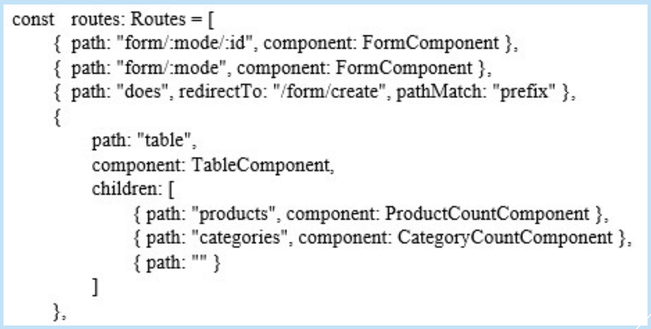
```
const   routes: Routes = [
        { path: "form/:mode/:id", component: FormComponent },
        { path: "form/:mode", component: FormComponent },
        { path: "does", redirectTo: "/form/create", pathMatch: "prefix" },
        { path: "table", component: TableComponent },
        { path: "", redirectTo: "/table", pathMatch: "full" },
        { path: "**", component: NotFoundComponent }
]
```



Name	Description
prefix	This value configures the route so that it matches URLs that start with the specified path, ignoring any subsequent segments.
full	This value configures the route so that it matches only the URL specified by the path property.

Creating Child Routes

- Child routes allow components to respond to part of the URL by embedding router-outlet elements in their templates, creating more complex arrangements of content.



```
const   routes: Routes = [
        { path: "form/:mode/:id", component: FormComponent },
        { path: "form/:mode", component: FormComponent },
        { path: "does", redirectTo: "/form/create", pathMatch: "prefix" },
        {
              path: "table",
              component: TableComponent,
              children: [
                    { path: "products", component: ProductCountComponent },
                    { path: "categories", component: CategoryCountComponent },
                    { path: "" }
              ]
        },
```

Creating Child Routes

- Child routes are defined using the children property, which is set to an array of routes defined in the same way as the top-level routes.
- When Angular uses the entire URL to match a route that has children, there will be a match only if the URL to which the browser navigates contains segments that match both the top-level segment and the segments specified by one of the child routes.

Angular animations

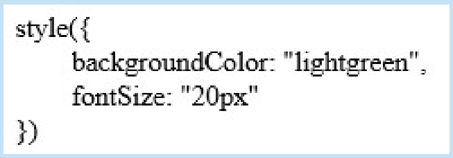
- The animations system can change the appearance of HTML elements to reflect changes in the application state.
- Used judiciously, animations can make applications more pleasant to use.
- Animations are defined using functions defined in the @angular/core module, registered using the animations property in the @Component decorator and applied using a data binding.

Adding the Animations Polyfill

- The Angular animations feature relies on a JavaScript API that is supported only by recent browsers, and a polyfill is required to allow older browsers to work.

Defining Style Groups

- The heart of the animation system is the style group, which is a set of CSS style properties and values that will be applied to an HTML element. Style groups are defined using the style function, which accepts a JavaScript object literal that provides a map between property names and values.



```
style({
        backgroundColor: "lightgreen",
        fontSize: "20px"
})
```

Defining Element States

- Angular needs to know when it needs to apply a set of styles to an element. This is done by defining an element state, which provides a name by which the set of styles can be referred to.
- Element states are created using the state function, which accepts the name and the style set that should be associated with it.

```
state("selected", style({
        backgroundColor: "lightgreen",
        fontSize: "20px"
})),
```

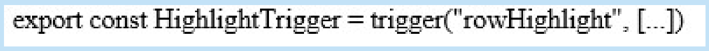
Defining State Transitions

- When an HTML element is in one of the states created using the state function, Angular will apply the CSS properties in the state's style group.
- The transition function is used to tell Angular how the new CSS properties should be applied.

```
transition("selected => notselected", animate("200 ms")),
transition("notselected => selected", animate("400 ms"))
```

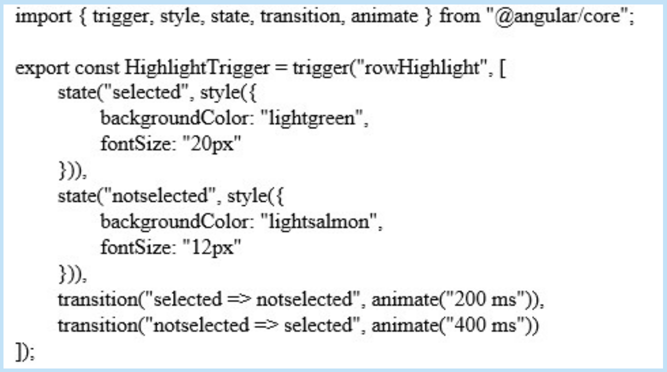
Defining the Trigger

- The final piece of plumbing is the animation trigger, which packages up the element states and transitions and assigns a name that can be used to apply the animation in a component. Triggers are created using the trigger function.



```
export const HighlightTrigger = trigger("rowHighlight", [...])
```

Defining the animation



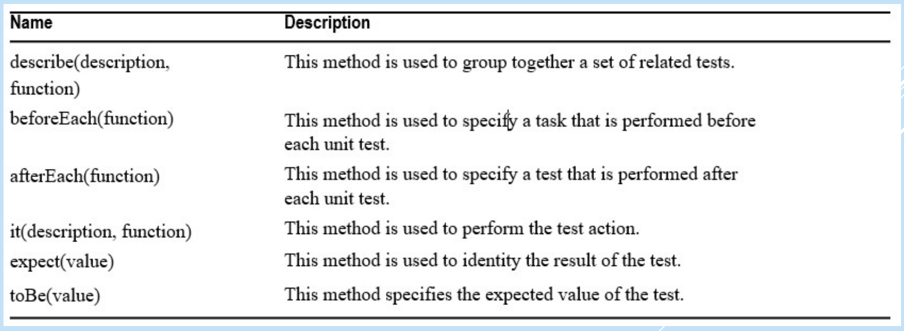
```
import { trigger, style, state, transition, animate } from "@angular/core";

export const HighlightTrigger = trigger("rowHighlight", [
        state("selected", style({
                backgroundColor: "lightgreen",
                fontSize: "20px"
        })),
        state("notselected", style({
                backgroundColor: "lightsalmon",
                fontSize: "12px"
        })),
        transition("selected => notselected", animate("200 ms")),
        transition("notselected => selected", animate("400 ms"))
]);
```

Angular Unit Testing

- Angular components and directives require special support for testing so that their interactions with other parts of the application infrastructure can be isolated and inspected.
- Isolated unit tests are able to assess the basic logic provided by the class that implements a component or directive but do not capture the interactions with host elements, services, templates, and other important Angular features.
- Angular provides a test bed that allows a realistic application environment to be created and then used to perform unit tests.

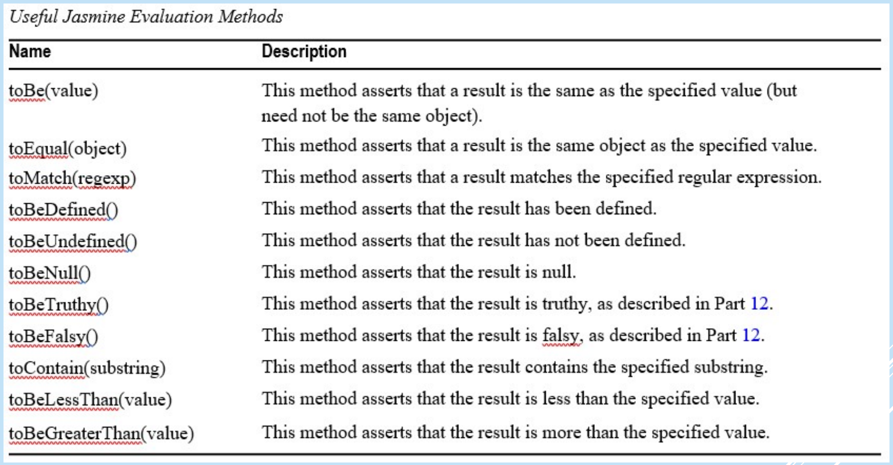
Working with Jasmine

- The API that Jasmine provides chains together JavaScript methods to define unit tests.
- Table below describes the most useful functions for Angular testing.



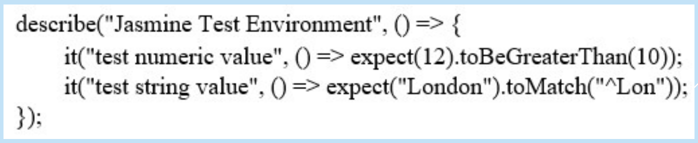
Name	Description
describe(description, function)	This method is used to group together a set of related tests.
beforeEach(function)	This method is used to specify a task that is performed before each unit test.
afterEach(function)	This method is used to specify a test that is performed after each unit test.
it(description, function)	This method is used to perform the test action.
expect(value)	This method is used to identity the result of the test.
toBe(value)	This method specifies the expected value of the test.

Working with Jasmine



Useful Jasmine Evaluation Methods

Name	Description
toBe(value)	This method asserts that a result is the same as the specified value (but need not be the same object).
toEqual(object)	This method asserts that a result is the same object as the specified value.
toMatch(regexp)	This method asserts that a result matches the specified regular expression.
toBeDefined()	This method asserts that the result has been defined.
toBeUndefined()	This method asserts that the result has not been defined.
toBeNull()	This method asserts that the result is null.
toBeTruthy()	This method asserts that the result is truthy, as described in Part 12.
toBeFalsy()	This method asserts that the result is falsy, as described in Part 12.
toContain(substring)	This method asserts that the result contains the specified substring.
toBeLessThan(value)	This method asserts that the result is less than the specified value.
toBeGreaterThan(value)	This method asserts that the result is more than the specified value.



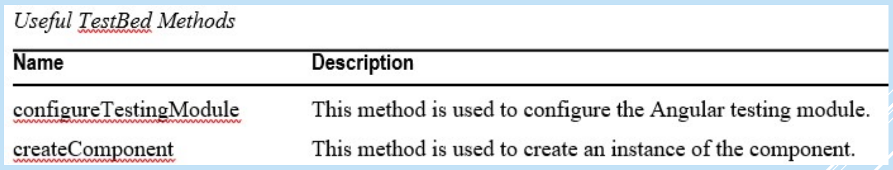
```
describe("Jasmine Test Environment", () => {
    it("test numeric value", () => expect(12).toBeGreaterThan(10));
    it("test string value", () => expect("London").toMatch("^Lon"));
});
```

Testing an Angular Component

- The building blocks of an Angular application can't be tested in isolation because they depend on the underlying features provided by Angular and by the other parts of the project, including the services , directives, templates, and modules it contains.
- As a consequence, testing a building block such as a component means using testing utilities that are provided by Angular to re-create enough of the application to let the component function so that tests can be performed against it.

Working with the TestBed Class

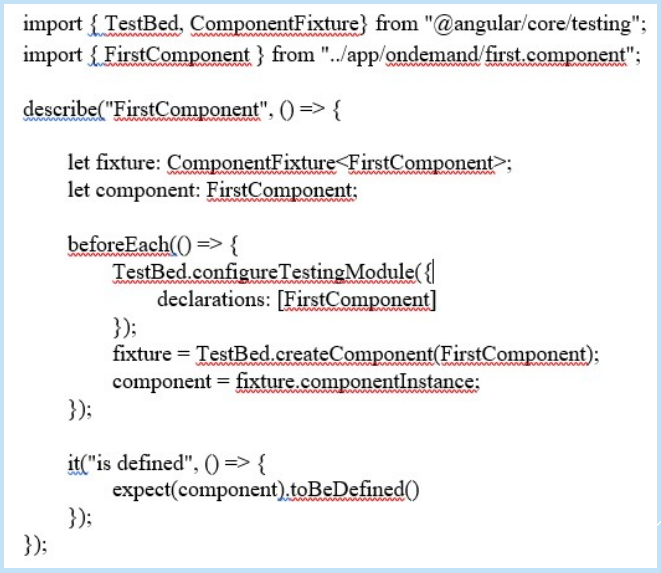
- At the heart of Angular unit testing is a class called TestBed, which is responsible for simulating the Angular application environment so that tests can be performed.



Useful TestBed Methods

Name	Description
configureTestingModule	This method is used to configure the Angular testing module.
createComponent	This method is used to create an instance of the component.

Working with the TestBed Class



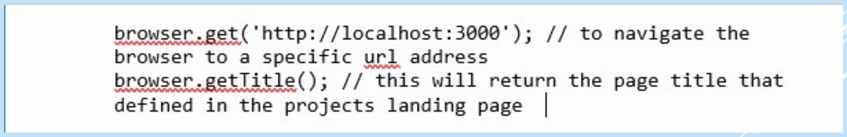
```
import { TestBed, ComponentFixture} from "@angular/core/testing";
import { FirstComponent } from "../app/ondemand/first.component";

describe("FirstComponent", () => {

    let fixture: ComponentFixture<FirstComponent>;
    let component: FirstComponent;

    beforeEach(() => {
        TestBed.configureTestingModule({
            declarations: [FirstComponent]
        });
        fixture = TestBed.createComponent(FirstComponent);
        component = fixture.componentInstance;
    });

    it("is defined", () => {
        expect(component).toBeDefined()
    });
});
```

An overview of Protractor

- Protractor is an end-to-end testing tool that runs using Node.js and is available as an npm package.
- End-to-end testing is testing an application against all the interconnected moving parts and layers of an application.
- With end-to-end testing, the focus is on how the application or a module, as a whole, works, such as confirming that the click of a button triggers x, y, and z actions.
- Protractor offers the ability to select a specific DOM element, share the data with that element, simulate the click of a button, and interact with an application in the same way as a user would.

Core of Protractor

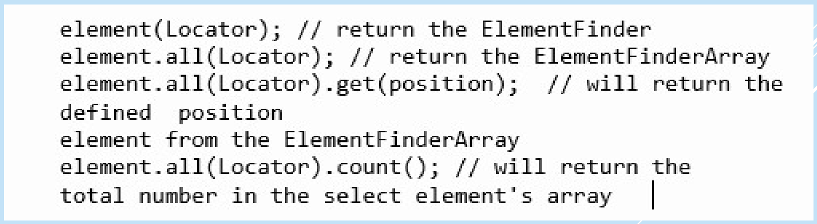
- Protractor provides some global functions; some are from its core API, and some are from WebDriver.
- **Browser**: Protractor provides the global function browser, which is a global object from WebDriver that is mostly used to interact with the application browser where the application is running during the e2e test process.



```
browser.get('http://localhost:3000'); // to navigate the
browser to a specific url address
browser.getTitle(); // this will return the page title that
defined in the projects landing page
```

Core of Protractor

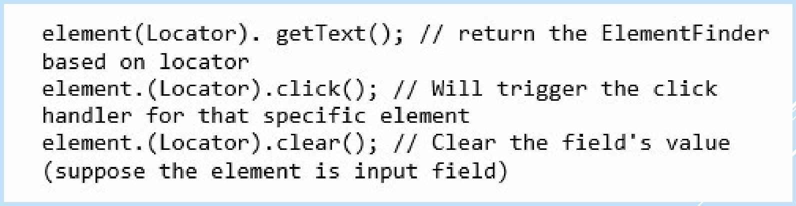
- **Element**: This is a global function provided by Protractor; it's basically used to find a single element based on the locator, but it supports multiple element selection as well, by chaining another method, .all as element.all, which also takes Locator and returns ElementFinderArray.



```
element(Locator); // return the ElementFinder
element.all(Locator); // return the ElementFinderArray
element.all(Locator).get(position);  // will return the
defined  position
element from the ElementFinderArray
element.all(Locator).count(); // will return the
total number in the select element's array
```

Core of Protractor

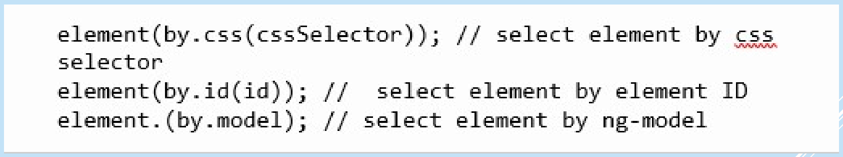
- **Action**: As we have seen, the element method will return a selected DOM element object, but we need to interact with a DOM and the actions for doing that job come with some built-in methods. The DOM will not contact the browser unit with any action method calls.



```
element(Locator). getText(); // return the ElementFinder
based on locator
element.(Locator).click(); // Will trigger the click
handler for that specific element
element.(Locator).clear(); // Clear the field's value
(suppose the element is input field)
```

Core of Protractor

- **Locator**: This actually informs Protractor how to find a certain element in the DOM element. Protractor exports Locator as a global factory function, which will be used with a global by object.



```
element(by.css(cssSelector)); // select element by css
selector
element(by.id(id)); //  select element by element ID
element.(by.model); // select element by ng-model
```