# NG

# Components

|  |
| --- |
| Architecting with Components |
| When building a new Angular application, we start by:   1. Breaking down an application into individual Components. 2. For each component we describe it’s own resonsibilities. 3. Once we’ve desribed the responsibilites, then we describe it’s inputs & outputs, it’s public facing interface.   Simple Example:   * Split [Application]-into--> [multiple Components] * For each component, List out the Responsibilities, Inputs and Outputs –  |  |  | | --- | --- | |  | **HeaderComponent**  **Responsibilities:** All aspects of authentication. Letting the user login/signup and logout.  Inputs: None  Outputs: LoginChanged — An output event that is fired when the users login state changes.  **SidebarComponent**  **Responsibilities:** Performing searches  **Inputs:** None  **Outputs:** SearchTermChanged — An output event that is fired when a user performs a search, $event contains the search term.  **ContentComponent**  **Responsibilities**: Showing the search results.  **Inputs**: SearchTerm the search term that we want to filter the results by.  **Outputs**: None |   Tip:   * Listing the inputs and outputs and what area this Component is responsible for helps to ensure the Components are architected correctly. * The goal is for each Component to have a well defined boundary.   **Data Flow:**   |  |  | | --- | --- | |  | The actual binding of inputs and outputs happens in HTML, in the templates of Components. The template for our root Component might end up looking like this  <**header** (loginChanged)="loggedIn = $event"></**header**>  <**sidebar** (searchTermChanged)="searchTerm = $event"></**sidebar**>  <**content** [searchTerm]="searchTerm"></**content**>  Closely looking at the diagram above an interesting fact occurs; with one way data binding, inputs go down the tree, outputs go up the tree. With one way data binding reasoning about your application becomes a lot simpler, we can trace through the flow of events in our application easily. |   **Summary**  Architecting an Angular application means understanding that it’s just a tree of Components, each Component has some inputs and outputs and should have a clear responsibility with respect to the rest of the application.  Components are composable, so we might go a step further and include a LoginButtonComponent in our HeaderComponent. But only if we would want to re-use the LoginButtonComponent independently of the HeaderComponent. |

|  |
| --- |
| Templates, Styles & View Encapsulation |
| **External –View (Template) & Styles:**  @**Component**({  selector: 'my-component',  templateUrl: 'my-component-component.html',  styleUrls: ['my-component-component.css']  })  class **MyComponent** {  }  **Inline –View (Template) & Styles:**  @**Component**({  selector: 'my-component',  template: `  <div *class*="card"></div>  `,  styles: [`  .card {  background-color: gray;  }  `]  })  class **MyComponent** {  }  **View Encapsulation:**   * Even though we changed the background color of .card and we have multiple cards on the page only the form component card was rendered with a gray background. * Normally if we change a css class the effect is seen throughout an application, something special is happening here and it’s called View Encapsulation. * Angular is inspired from Web Components, a core feature of which is the shadow DOM. * The shadow DOM lets us include styles into Web Components without letting them leak outside the component’s scope. * Angular also provides this feature for Components and we can control it with the encapsulation property. * The valid values for this config property are: * **ViewEncapsulation**.Emulated (default) * **ViewEncapsulation**.Native * **ViewEncapsulation**.None   <http://plnkr.co/edit/yipOBmmTTpymS7vVWC57?p=preview>   |  | | --- | | import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';  import { Component, NgModule, Input, Output, EventEmitter, ViewEncapsulation } from '@angular/core';  import { BrowserModule } from '@angular/platform-browser';  @**Component**({  selector: 'my',  template: `  MyComponent <small *class*="text-muted">(with NO styles & no encapsulation specified)</small>  <p *class*="text-muted"> (so it applies default 'encapsulation: ViewEncapsulation.Emulated')</p>  <br/>  <div *class*="card card-block">  <h4 *class*="card-title">Title</h4>  <div *class*="form-group">  <input *type*="text" *class*="form-control" *placeholder*="Textbox1">  </div>  <button *type*="button" *class*="btn btn-primary">Save</button>  </div>  `  }  )  class **MyComponent** {  }    @**Component**({  selector: 'my-view-encapsulation-emulated',  template: `    MyComponent <small *class*="text-muted">(with specific styles and encapsulation: ViewEncapsulation.Emulated) </small>  <br/>  <div *class*="card card-block">  <h4 *class*="card-title">Title</h4>  <div *class*="form-group">  <input *type*="text" *class*="form-control" *placeholder*="Textbox1">  </div>  <button *type*="button" *class*="btn btn-primary">Save</button>  </div>  `,  styles: [  `  .card-title {  color: blue;  }  `  ],  *// View accept all global styles (e.g. bootstrap)*  *// View overides global styles with component specific 'styles'*  *// Default (need not to specify):*  encapsulation: ViewEncapsulation.Emulated  }  )  class **MyViewEncapsulationEmulatedComponent** {  }  @**Component**({  selector: 'my-view-encapsulation-native',  template: `    MyComponent <small *class*="text-muted">(with specific styles & encapsulation: ViewEncapsulation.Native)</small>  <br/>  <div *class*="card card-block">  <h4 *class*="card-title">Title</h4>  <div *class*="form-group">  <input *type*="text" *class*="form-control" *placeholder*="Textbox1">  </div>  <div *class*="form-group">  <input *type*="text" *class*="form-control" *placeholder*="Textbox2">  </div>  <button *type*="button" *class*="btn btn-primary">Save</button>  </div>  `,  styles: [  `  .card-title {  color: green;  }  `  ],  *// Fully Native Style:*  *// View doesn't accept any global styles (e.g. bootstrap)*  *// View still accepts styles from other angular component which has 'encapsulation: ViewEncapsulation.None'*  *// View does NOT overides other ['ViewEncapsulation.None' component] styles*  encapsulation: ViewEncapsulation.Native  })  class **MyViewEncapsulationNativeComponent** {  }  @**Component**({  selector: 'my-view-encapsulation-none',  template: `    MyComponent <small *class*="text-muted">(with specific styles & encapsulation: ViewEncapsulation.None)</small>  <br/>  <div *class*="card card-block">  <h4 *class*="card-title">Title</h4>  <div *class*="form-group">  <input *type*="text" *class*="form-control" *placeholder*="Textbox1">  </div>  <div *class*="form-group">  <input *type*="text" *class*="form-control" *placeholder*="Textbox2">  </div>  <button *type*="button" *class*="btn btn-primary">Save</button>  </div>  `,  styles: [  `  .card-title {  color: red;  }  `  ],  *// View accept all global styles (e.g. bootstrap)*  *// Any styles specified to this component will get applied globally (including ViewEncapsulation.Native)*  *// Now all components has '.card-title' gets red color*  *// expect the ['ViewEncapsulation.Emulated' component] which overrides the style '.card-title'*  encapsulation: ViewEncapsulation.None  })  class **MyViewEncapsulationNoneComponent** {  }  @**Component**({  selector: 'app',  template: `  <my></my>  <my-view-encapsulation-emulated> </my-view-encapsulation-emulated><br/>  <my-view-encapsulation-native> </my-view-encapsulation-native><br/>  <my-view-encapsulation-none></my-view-encapsulation-none><br/>  `  })  class **AppComponent** {  }  @**NgModule**({  imports: [BrowserModule],  declarations: [  AppComponent,  MyComponent,  MyViewEncapsulationEmulatedComponent,  MyViewEncapsulationNativeComponent,  MyViewEncapsulationNoneComponent  ],  bootstrap: [AppComponent]  })  export class **AppModule** {  }  **platformBrowserDynamic**().**bootstrapModule**(AppModule); | |  |   Notes:   * By default styles for our components are encapsulated, that means that they don’t leak out and affect the rest of the application. * We can explicitly set the encapsulation strategy using the encapsulation property. * By default, the renderer uses ViewEncapsulation.Emulated if the view has styles, otherwise ViewEncapsulation.None. * There is also a ViewEncapsulation.Native method which uses the shadow DOM to encapsulate the view. |

# Directives

|  |
| --- |
| Directives |
| Directives are Components without a View.   * They are components without a template. * Or to put it another way, components are directives with a view.   Component = Directive functionality + View   * Everything you can do with a directive you can also do with a component. * But not everything you can do with a component you can do with a directive.   We typically associate directives to existing elements by using attribute selectors,  <elemenent *aDirective*></element>  We capitalise the name of directives when we are talking about the directive class.   * For example when we say NgFor we mean the class which defines the NgFor directive.   When we are talking about either an instance of a directive or the attribute   * we use to associate a directive to an element we lowercase the first letter. * So ngFor refers to both the instance of a directive and the \_attribut\_e name used to associate a directive with an element.   **Directives**   * Built-in Directives * Custom Directives   **Built-in Directives:**   * NgFor * NgIf * NgSwitch * NgClass * NgNonBindable   Note:   * NgIf, NgFor and NgSwitch are structural directives * Structural Directives are directives which change the structure of the DOM by adding or removing elements.   … |

|  |
| --- |
| Built-in Directives |
| NgFor : **Syntax:** \*ngFor="let <value> of <collection>"   * <value> is a variable name of your choosing, * <collection> is a property on your component which holds a collection,   + usually an array but anything that can be iterated over in a for-of loop.   <ul>  <li \**ngFor*="let person of people; let i = index">  {{ i + 1 }} - {{ person.name }}  </li>  </ul>  <ul \**ngFor*="let group of peopleByCountry">  <li>{{ group.country }}</li>  <ul>  <li \**ngFor*="let person of group.people">  {{ person.name }}  </li>  </ul>  </ul> NgIf: Syntax: \*ngIf="<condition>"   * [hidden]='false' 🡪 simply hides the element * \*ngIf='false' 🡪 removes the element completely from the DOM   <ul \**ngFor*="let person of people">  <li \**ngIf*="person.age < 30">  {{ person.name }} ({{ person.age }})  </li>  </ul>  We can’t have two structural directives, --- directives starting with a \* attached to the same element.  The below code would not work:  <ul \**ngFor*="let person of people" \**ngIf*="person.age < 30">  <li>{{ person.name }}</li>  </ul> NgSwitch: Without NgSwitch:  <ul \**ngFor*="let person of people">  <li \**ngIf*="person.country ==='UK'"  *class*="text-success">{{ person.name }} ({{ person.country }})  </li>  <li \**ngIf*="person.country === 'USA'"  *class*="text-primary">{{ person.name }} ({{ person.country }})  </li>  <li \**ngIf*="person.country === 'HK'"  *class*="text-danger">{{ person.name }} ({{ person.country }})  </li>  <li \**ngIf*="person.country !== 'HK' && person.country !== 'UK' && person.country !== 'USA'"  *class*="text-warning">{{ person.name }} ({{ person.country }})  </li>  </ul>  With NgSwitch:    <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>  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.  NgStyle: **Syntax:** [ngStyle]="{'background-color':'green'}"  <div [*ngStyle*]="{'background-color':'green'}"></<div>  <div [*ngStyle*]="{'background-color':person.country === 'UK' ? 'green' : 'red' }"></<div>  <div [*ngStyle*]="{'font-size.px':24}"></<div>  <div [*style*.*color*]="'red'"></<div>  <div [*style*.*font-size*.*px*]="24"></<div> NgClass: **Syntax:** [ngClass]="{'text-success':true}"  <div [*ngClass*]="{'text-success':true}"></<div>  <div [*ngClass*]="{'text-success':person.country === 'UK'}"></<div>  <div [*class*.*text-success*]="true"></<div>  <div [*class*.*text-success*]="person.country === 'UK'"></<div> NgNonBindable: Without NgNonBindable:   |  |  | | --- | --- | | <div>  To render the name variable we use this syntax  <pre>{{ name }}</pre>  </div> | ../../../Desktop/Screen%20Shot%202018-02-16%20at%203.29.55%20PM.png |  * Normally Angular will try to find a variable called {{ name }} on the component * and print out the value of the name variable instead of just printing out {{ name }}.   With NgNonBindable:   * To make angular ignore an element we simply add the ngNonBindable directive to the element  |  |  | | --- | --- | | <div>  To render the name variable we use this syntax  <pre *ngNonBindable*>{{ name }}</pre>  </div> | ../../../Desktop/Screen%20Shot%202018-02-16%20at%203.31.46%20PM.png |  Structural Directives (in long form / long syntax):  * 3 built in structural directives, NgIf, NgFor and NgSwitch * Structural Directives are directives which change the structure of the DOM by adding or removing elements.   Long form vs \* (structural directives) :  <ng-template [*ngIf*]="!data.hide">  <p *class*="card-text"> {{ data.punchline }}</p>  </ng-template>  <p *class*="card-text" \**ngIf*="!data.hide">  {{ data.punchline }}  </p>  <ng-template *ngFor* *let-j* [*ngForOf*]="jokes">  <joke [*joke*]="j"></joke>  </ng-template>  <joke \**ngFor*="let j of jokes" [*joke*]="j">  </joke>   * We use the <ng-template> tag to define the element we want to insert into the DOM. * We can prepend the directive name with \*   + to skip having to define a <ng-template>   + and have the directive use the element it’s attached to as the template.   … |

# Reactive Programming with RxJS

|  |
| --- |
| Reactive Programming Introduction |
| Reactive Programming = Streams + Operations    **What are Streams?**:   * Streams are a sequence of values over time * For example a number that goes up by 1 every second might have a stream   + [0, 1, 2, 3, 4] * Another stream might be a sequence of x and y positions of mouse click events, like so:   + [ (12,34), (345,22), (1,993)] * We could have a stream to represent each keypress (while user filling form)   + [J, A, G, A, D, E, E, S, H] * Or we could have a stream which contains a JSON representation of the whole form as the user enters data, like so:   [  { "name": "J" },  { "name": "Ja" },  { "name": "Jag" },  { "name": "Jaga" }  ]   * We could have a stream for:   + The x,y position of the mouse as it moves around the screen in a HTML5 game.   + The data returned from a real-time websockets connection.   + The chat windows opened by this user in a browser.   Mostly, “ Everything we do with a web application can be thought of as a stream”  **What is Reactive Programming?**   * Reactive Programming = Streams + Operations   Reactive programming is the idea that you can create your entire program   * just by defining the different **streams** and the **operations** that are performed on those streams   To explain this lets convert a simple imperative function into a reactive one.   * Imperative programming is a programming paradigm that you probably have been using so far in your career,   + it’s by far the most common and it’s involves executing statements that change a programs state,   + i.e. call functions that change the values of variables  |  |  | | --- | --- | |  | We have a function called add and some state variables, A, B and C.   * To add A and B together and change the state of C to be the sum we call the function add.   Later on the value of B changes to 4.   * First we need a way of simply knowing that B has changed, that’s hard enough to figure out by itself. * Secondly we need to know that because B has changed we need to recalculate C.   In a web application our inputs are constantly changing over time, via user interaction or network events   * Most of the logic we end up writing is, Just to figure out   + 1. what functions need to be called   + 2. and in what order functions needs to be called   + for each of these changes to our inputs.   **Note:**   * Applications can be thought of as just a   + huge pile of variables (which we call application state)   + as well logic to decide which functions to call, and in what order,     - --- when any of those variables change. * Calling those functions then also changes the values of variables, for which we need additional logic to figure out what other functions to call…​ it’s endless! |  |  |  | | --- | --- | |  | With reactive programming   * we stop thinking about variables, * instead we think in terms of streams * and how those streams are connected together.   we convert the variables A, B and C into 🡺 streams.   * A is now not an individual value at one point in time,   + it’s a stream of values over time. * The function add() we think of an an operation we perform * which connects the   + { output of streams A and B} to the {input of stream C}   Now if we push some numbers onto stream A and B, the add operation is automatically called, calculating the sum of 3 and pushing it onto stream C.  If stream C was connected to another stream via another operation, that operation would then be called automatically as well.  **Tip:**  An analogy which works for me is to think about reactive programming as **plumbing**.   * We decide which pipes we need in our application, * we decide how those pipes are connected together * and then we turn on the water and sit back.   With reactive programming   * we don’t call functions * we just define how our application is plumbed together * and start pushing values onto streams * and let the plumbing and operations handle the rest.   So if later on the value of B changes, we simply push the new value onto the stream B and then let the plumbing handle the rest |   … |
|  |

|  |
| --- |
| RxJS |
| ReactiveX – an API for asynchronous programming with observable streams   * RxJS is the JavaScript implementation of the **ReactiveX –API** * RxJava is the Java implementation of the ReactiveX –API * Rx.NET is the .NET implementation of the ReactiveX –API   RxJS stands for \*R\*eactive E\*x\*tensions for \*J\*ava\*S\*cript,   * and its a library that gives us an implementation of Observables for JavaScript.   Observables might become a core part of the JavaScript language (ES 7) in the future,   * so we can think of RxJS as a filler / placeholder for when that arrives.   let obs1 = Rx.Observable  .**operator1**();  .**operator2**();  .**operator3**();  .**operator4**();  .**operator5**();  obs1.**subscribe**(value => {  **console**.**log**("Subscriber: " + value)  });  let obs1 = Rx.Observable  .**interval**(1000)  .**take**(3)  .**map**((v) => **Date**.**now**());  obs1.**subscribe**(value => {  **console**.**log**("Subscriber: " + value)  });  … |

|  |
| --- |
| RxJS with Angular |
| There are a few places in Angular where reactive programming and observables are in use.  **EventEmitter**: Under the hood this works via Observables.  **HTTP**: We’ve not covered this yet but HTTP requests in Angular are all handled via Observables.  **Forms**: Reactive forms in Angular expose an observable, a stream of all the input fields in the form combined.  Reactive Form Example:  import {NgModule, Component} from '@angular/core';  import {BrowserModule} from '@angular/platform-browser';  import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';  import {ReactiveFormsModule, FormGroup, FormControl, Validators, FormBuilder} from "@angular/forms";  import 'rxjs/Rx';  @**Component**({  selector: 'form-app',  template: `    <form *[formGroup]*="fg1" *(ngSubmit)*="**onSubmit**()">  <textarea *class*="form-control" *placeholder*="Comments:" *formControlName*="comment"></textarea>  <p *class*="text-muted"> {{ 100 - fg1.value.comment.length }} characters left </p>    <input *type*="text"  *class*="form-control"  *placeholder*="Name:"  *formControlName*="name">    <input *type*="email"  *class*="form-control"  *placeholder*="Email:"  *formControlName*="email">    <p *class*="text-muted"> {{ fg1.value.lastUpdateTS }} </p>  <button *type*="submit"  *class*="btn btn-primary"  *[disabled]*="!fg1.valid">Submit  </button>    </form>  `  })  class **FormAppComponent** {  *// instances of Form Group*  fg1: **FormGroup**;    *// individual instances of Controls and Rules (user input is valid or invalid)*  comment = new **FormControl**("", Validators.required);  name = new **FormControl**("", Validators.required);  email = new **FormControl**("", [  Validators.required,  Validators.**pattern**("[^ @]\*@[^ @]\*")  ]);    */\* Observable Solution \*/*  constructor(fb1: **FormBuilder**) {    *// Using FormBuilder, Group the individual Form Controls :*  this.fg1 = fb1.**group**({  "comment": this.comment,  "name": this.name,  "email": this.email  });      *// FormGroup instance exposes an observable (valueChanges),*  *// - a stream of all the input fields combined into a object*  *// We can subscribe to that observable*  this.fg1.valueChanges  *// Processing only valid form values*  .**filter**(data => this.fg1.valid)  *// Cleaning form data (comment textarea values)*  .**map**(data => {  data.comment = data.comment.**replace**(/<(?:.|\n)\*?>/gm, '');  return data  })  *// Adding new proeprties to the form*  .**map**(data => {  data.lastUpdateTS = new **Date**();  return data  })  .**subscribe**( data => **console**.**log**(JSON.**stringify**(data)));    }  */\* None Observable Solution \*/*  */\**    *constructor(fb1: FormBuilder) {*  *this.fg1 = fb1.group({*  *"comment": this.comment,*  *"name": this.name,*  *"email": this.email*  *});*  *this.fg1.valueChanges*  *.subscribe( data => {*  *if (this.fg1.valid) {*  *data.comment = data.comment.replace(/<(?:.|\n)\*?>/gm, '');*  *data.lastUpdateTS = new Date();*  *console.log(JSON.stringify(data))*  *}*  *});*  *}*  *\*/*  **onSubmit**() {  **console**.**log**("Form submitted!");  }  }  @**Component**({  selector: 'app',  template: `  <form-app></form-app>  `  })  class **AppComponent** {  }  @**NgModule**({  imports: [BrowserModule, ReactiveFormsModule],  declarations: [AppComponent, FormAppComponent],  bootstrap: [AppComponent],  })  class **AppModule** {  }  **platformBrowserDynamic**().**bootstrapModule**(AppModule);  … |

|  |
| --- |
| Pipe |
| Built-in Pipes  import { NgModule, Component } from '@angular/core';  import { BrowserModule } from '@angular/platform-browser';  import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';  @**Component**({  selector: 'pipe-builtins',  template: `  <div *class*="card card-block">  <h4 *class*="card-title">Currency</h4>  <div *class*="card-text">  *<!-- CA$1,234.56 -->*  <p>{{ 1234.56 | currency:"CAD" }}</p>    *<!-- CAD1,234.56 -->*  <p>{{ 1234.56 | currency:'CAD':'code'}}</p>    *<!-- CA$1,234.56 -->*  <p>{{ 1234.56 | currency:'CAD':'symbol'}}</p>    *<!-- $1,234.56 -->*  <p>{{ 1234.56 | currency:'CAD':'symbol-narrow'}}</p>  </div>  </div>  <div *class*="card card-block">  <h4 *class*="card-title">Date</h4>  <div *class*="card-text">  *<!-- 12:05 AM -->*  <p>{{ dateVal | date: 'shortTime' }}</p>  *<!-- Sunday, February 18, 2018 -->*  <p>{{ dateVal | date: 'fullDate' }}</p>  *<!-- 12:05 AM -->*  <p>{{ dateVal | date: 'shortTime' }}</p>  *<!-- 18/2/2018 -->*  <p>{{ dateVal | date: 'd/M/y' }}</p>  </div>  </div>  <div *class*="card card-block">  <div *class*="card-text">  <h4 *class*="card-title">DecimalPipe</h4>  *<!-- 003.14 -->*  <p>{{ 3.14159265 | number: '3.1-2' }}</p>  *<!-- 3.1416 -->*  <p>{{ 3.14159265 | number: '1.4-4' }}</p>  </div>  </div>  <div *class*="card card-block">  <h4 *class*="card-title">JsonPipe</h4>  <div *class*="card-text">  *<!-- [object Object] -->*  <p>{{ jsonVal }}</p>    *<!-- { "moo": "foo", "goo": { "too": "new" } } -->*  <p>{{ jsonVal | json }}</p>  </div>  </div>  <div *class*="card card-block">  <h4 *class*="card-title">LowerCasePipe</h4>  <div *class*="card-text">  *<!-- asim -->*  <p>{{ 'ASIM' | lowercase }}</p>  </div>  </div>  <div *class*="card card-block">  <h4 *class*="card-title">UpperCasePipe</h4>  <div *class*="card-text">  *<!-- ASIM -->*  <p>{{ 'asim' | uppercase }}</p>  </div>  </div>  <div *class*="card card-block">  <h4 *class*="card-title">PercentPipe</h4>  <div *class*="card-text">  *<!-- 12% -->*  <p>{{ 0.123456 | percent }}</p>  *<!-- 12.35% -->*  <p>{{ 0.123456 | percent: '2.1-2' }}</p>  *<!-- 0,000,000,012.3456% -->*  <p>{{ 0.123456 | percent : "10.4-4" }}</p>  </div>  </div>  <div *class*="card card-block">  <h4 *class*="card-title">SlicePipe</h4>  <div *class*="card-text">  *<!-- [...] | slice: startIndex : endIndex -->*  *<!-- 2,3 -->*  <p>{{ [1,2,3,4,5,6] | slice:1:3 }}</p>  *<!-- 3,4,5,6 -->*  <p>{{ [1,2,3,4,5,6] | slice:2 }}</p>  *<!-- 3,4,5 -->*  <p>{{ [1,2,3,4,5,6] | slice:2:-1 }}</p>  *<!-- '-' negative means index starts from end -->*    *<!--*  *\*3*  *\*4*  *\*5*  *-->*  <ul>  <li *\*ngFor*="let v of [1,2,3,4,5,6] | slice:2:-1">  {{v}}  </li>  </ul>    </div>  </div>  `  })  class **PipeBuiltinsComponent** {  private dateVal: **Date** = new **Date**();  private jsonVal: **Object** = { moo: 'foo', goo: { too: 'new' } };  }  @**Component**({  selector: 'app',  template: `  <pipe-builtins></pipe-builtins>  `  })  class **AppComponent** {  }  @**NgModule**({  imports: [BrowserModule],  declarations: [AppComponent,  PipeBuiltinsComponent  ],  bootstrap: [AppComponent],  })  class **AppModule** {  }  **platformBrowserDynamic**().**bootstrapModule**(AppModule);  AsyncPipe  import {NgModule, Component, OnDestroy} from '@angular/core';  import {BrowserModule} from '@angular/platform-browser';  import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';  import { Observable } from 'rxjs/Rx';  @**Component**({  selector: 'async-pipe',  template: `  <div *class*="card card-block">  <h4 *class*="card-title">AsyncPipe</h4>    *<!-- 'Promise' with AsyncPipe -->*  <p *class*="card-text">{{ promise | async }} </p>  *<!-- 'Promise' with AsyncPipe -->*  <p *class*="card-text">{{ promiseData }} </p>  *<!-- 'Observable' with AsyncPipe -->*  <p *class*="card-text">{{ observable | async }}</p>    *<!-- 'Observable' without AsyncPipe -->*  <p *class*="card-text">{{ observableData }}</p>  </div>  `  })  class **AsyncPipeComponent** implements **OnDestroy** {    promise: **Promise**<string>;  promiseData: string;    observable: **Observable**<number>;  subscription: **Object** = null;  observableData: number;  constructor() {  *// 'Promise' with AsyncPipe*  this.promise = this.**getPromise**();    *// 'Promise' without AsyncPipe*  this.**getPromise**().**then**(v => this.promiseData = v);    *// 'Observable' with AsyncPipe*  this.observable = this.**getObservable**();    *// 'Observable' without AsyncPipe*  this.subscription = this.**getObservable**()  .**subscribe**((v) => this.observableData = v);  }      **getPromise**() {  return new **Promise**((resolve, reject) => {  **setTimeout**(() => **resolve**("Promise complete!"), 3000);  });  }  **getObservable**() {  return Observable  .**interval**(1000)  .**take**(10)  .**map**((v) => v \* v);  }    **ngOnDestroy**() {  if (this.subscription) {  *// AsyncPipe, automatically unsubscribes the observables*  this.subscription.**unsubscribe**();  }  }  }  @**Component**({  selector: 'app',  template: `  <async-pipe></async-pipe>  `  })  class **AppComponent** {  imageUrl: string = "";  }  @**NgModule**({  imports: [BrowserModule],  declarations: [AppComponent,  AsyncPipeComponent  ],  bootstrap: [AppComponent],  })  class **AppModule** {  }  **platformBrowserDynamic**().**bootstrapModule**(AppModule);  Custom Pipes  import { NgModule, Component, Pipe } from '@angular/core';  import { BrowserModule } from '@angular/platform-browser';  import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';  import { Observable } from 'rxjs/Rx';  @**Pipe**({  name: "defaultImg"  })  class **DefaultPipe** {  **transform**(value: string, fallback: string, forceHttps: boolean = false): string {  let image = "";  if (value) {  image = value;  } else {  image = fallback;  }  if (forceHttps) {  if (image.**indexOf**("https") == -1) {  image = image.**replace**("http", "https");  }  }  return image;  }  }  @**Component**({  selector: 'app',  template: `  <img *[src]*="imageUrl | defaultImg: 'https://png.icons8.com/user.png' : true"/>    *<!-- Syntax: -->*  *<!-- <element1 [myProp]="param1 | pipeName: param2 : param3 "> </element1> -->*    *<!-- something like this [doesnt work, just for understanding] -->*  *<!-- <img [src]="defaultPipe.transform(imageUrl,'https://png.icons8.com/user.png', true )" -->*      `  })  class **AppComponent** {  imageUrl: string = "";  }  @**NgModule**({  imports: [BrowserModule],  declarations: [  AppComponent,  DefaultPipe  ],  bootstrap: [AppComponent],  })  class **AppModule** {  }  **platformBrowserDynamic**().**bootstrapModule**(AppModule);  … |

|  |
| --- |
|  |
| Forms:  * Model Driven Approach * Template Driven Approach  Model Driven Approach :  |  | | --- | | import { NgModule, Component, Pipe, OnInit } from '@angular/core';  import { ReactiveFormsModule, FormsModule, FormGroup, FormControl, Validators, FormBuilder } from '@angular/forms';  import { BrowserModule } from '@angular/platform-browser';  import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';  @**Component**({  selector: 'model-form',  template: `  <form *novalidate* *[formGroup]*="myFormGroup" *(ngSubmit)*="**onSubmit**()">  <fieldset *formGroupName*="name">  <div *class*="form-group" *[ngClass]*="{  'has-danger': firstName.invalid && (firstName.dirty || firstName.touched),  'has-success': firstName.valid && (firstName.dirty || firstName.touched)  }">  <label>First Name</label>  <input *type*="text" *class*="form-control" *formControlName*="firstName">  <div *class*="form-control-feedback" *\*ngIf*="firstName.errors && (firstName.dirty || firstName.touched)">  <p *\*ngIf*="firstName.errors.required">First Name is required</p>  </div>  <pre>Valid? {{ myFormGroup.controls.name.controls.firstName.valid }}</pre>  <pre>Dirty? {{ myFormGroup.controls.name.controls.firstName.dirty }}</pre>  *<!-- or -->*  <pre>Valid? {{ firstName.valid }}</pre>  <pre>Dirty? {{ firstName.dirty }}</pre>  </div>  <div *class*="form-group" *[ngClass]*="{  'has-danger': lastName.invalid && (lastName.dirty || lastName.touched),  'has-success': lastName.valid && (lastName.dirty || lastName.touched)  }">  <label>Last Name</label>  <input *type*="text" *class*="form-control" *formControlName*="lastName">  <div *class*="form-control-feedback" *\*ngIf*="lastName.errors && (lastName.dirty || lastName.touched)">  <p *\*ngIf*="lastName.errors.required">Last Name is required</p>  </div>  </div>  </fieldset>  <div *class*="form-group" *[ngClass]*="{  'has-danger': email.invalid && (email.dirty || email.touched),  'has-success': email.valid && (email.dirty || email.touched)  }">  <label>Email</label>  <input *type*="email" *class*="form-control" *formControlName*="email">  <div *class*="form-control-feedback" *\*ngIf*="email.errors && (email.dirty || email.touched)">  <p *\*ngIf*="email.errors.required">Email is required</p>  <p *\*ngIf*="password.errors.pattern">The email address must contain at least the @ character</p>  </div>  <pre>Dirty? {{ myFormGroup.controls.email.dirty }}</pre>  <pre>Pristine? {{ myFormGroup.controls.email.pristine }}</pre>  *<!-- Touched: user focused on the control and then focused on something else -->*  <pre>Touched? {{ myFormGroup.controls.email.touched }}</pre>  <pre>Untouched? {{ myFormGroup.controls.email.untouched }}</pre>  <pre>Valid? {{ myFormGroup.controls.email.valid }}</pre>  <pre>Invalid? {{ myFormGroup.controls.email.invalid }}</pre>  </div>  <div *class*="form-group" *[ngClass]*="{  'has-danger': password.invalid && (password.dirty || password.touched),  'has-success': password.valid && (password.dirty || password.touched)  }">  <label>Password</label>  <input *type*="password" *class*="form-control" *formControlName*="password">  <div *class*="form-control-feedback" *\*ngIf*="password.errors && (password.dirty || password.touched)">  <p *\*ngIf*="password.errors.required">Password is required</p>  <p *\*ngIf*="password.errors.minlength">Password must be 8 characters long, we need another {{password.errors.minlength.requiredLength - password.errors.minlength.actualLength}}  characters </p>  </div>  </div>    <pre>{{ password.errors | json }}</pre>  <div *class*="form-group" *[ngClass]*="{  'has-danger': language.invalid && (language.dirty || language.touched),  'has-success': language.valid && (language.dirty || language.touched)  }">  <label>Language</label>  <select *class*="form-control" *formControlName*="language">  <option *value*="">Please select a language</option>  <option *\*ngFor*="let lang of langs" *[value]*="lang">{{lang}}  </option>  </select>  </div>  <button *type*="submit" *class*="btn btn-primary">Submit</button>  <pre>{{myFormGroup.value | json}}</pre>  </form>  `  })  class **ModelFormComponent** implements **OnInit** {  langs: string[] = [  'English',  'French',  'German',  ];  myFormGroup: **FormGroup**;  firstName: **FormControl**;  lastName: **FormControl**;  email: **FormControl**;  password: **FormControl**;  language: **FormControl**;  **ngOnInit**() {  this.**createFormControls**();  this.**createForm**();  }  **createFormControls**() {  *// Angular has built-in validators in the Validators module*  *// required, minLength, maxLength and pattern*  this.firstName = new **FormControl**('', Validators.required);  this.lastName = new **FormControl**('', Validators.required);  this.email = new **FormControl**('', [  Validators.required,  Validators.**pattern**("[^ @]\*@[^ @]\*")  ]);  this.password = new **FormControl**('', [  Validators.required,  Validators.**minLength**(8)  ]);  this.language = new **FormControl**('');  }  **createForm**() {  *// Form Group:*  this.myFormGroup = new **FormGroup**({  name: new **FormGroup**({  firstName: this.firstName,  lastName: this.lastName,  }),  email: this.email,  password: this.password,  language: this.language  });  }  **onSubmit**() {  if (this.myFormGroup.valid) {  **console**.**log**("Form Submitted!", this.myFormGroup.value);  *// Reset Form:*  this.myFormGroup.**reset**();  }  }  }  @**Component**({  selector: 'app',  template: `<model-form></model-form>`  })  class **AppComponent** {  }  @**NgModule**({  imports: [  BrowserModule,  FormsModule,  ReactiveFormsModule],  declarations: [  AppComponent,  ModelFormComponent  ],  bootstrap: [  AppComponent  ],  })  class **AppModule** {  }  **platformBrowserDynamic**().**bootstrapModule**(AppModule); |  Template Driven Approach:  |  | | --- | | import { NgModule, Component, OnInit, ViewChild } from '@angular/core';  import { FormsModule,FormGroup,FormControl } from '@angular/forms';  import { BrowserModule } from '@angular/platform-browser';  import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';  class **Signup** {  constructor(public firstName: string = '',  public lastName: string = '',  public email: string = '',  public password: string = '',  public language: string = '') {  }  }  @**Component**({  selector: 'template-form',  template: `  <form *novalidate* *(ngSubmit)*="**onSubmit**()" *#myFormTemplateRef*="ngForm">  <fieldset *ngModelGroup*="name">  <div *class*="form-group" *[ngClass]*="{  'has-danger': firstName.invalid && (firstName.dirty || firstName.touched),  'has-success': firstName.valid && (firstName.dirty || firstName.touched)  }">  <label>First Name</label>  <input *type*="text" *class*="form-control" *name*="firstName"  *[(ngModel)]*="model.firstName" *required* *#firstName*="ngModel">  <div *class*="form-control-feedback" *\*ngIf*="firstName.errors && (firstName.dirty || firstName.touched)">  <p *\*ngIf*="firstName.errors.required">First name is required</p>  </div>    <pre>Valid? {{ myFormTemplateRef.form.controls.name.controls.firstName.valid }}</pre>  <pre>Dirty? {{ myFormTemplateRef.form.controls.name.controls.firstName.dirty }}</pre>  *<!-- or -->*  <pre>Valid? {{ firstName.valid }}</pre>  <pre>Dirty? {{ firstName.dirty }}</pre>  </div>  <div *class*="form-group" *[ngClass]*="{  'has-danger': lastName.invalid && (lastName.dirty || lastName.touched),  'has-success': lastName.valid && (lastName.dirty || lastName.touched)  }">  <label>Last Name</label>  <input *type*="text" *class*="form-control" *name*="lastName"  *[(ngModel)]*="model.lastName" *required* *#lastName*="ngModel">  <div *class*="form-control-feedback" *\*ngIf*="lastName.errors && (lastName.dirty || lastName.touched)">  <p *\*ngIf*="lastName.errors.required">Last name is required</p>  </div>  </div>  </fieldset>  <div *class*="form-group" *[ngClass]*="{  'has-danger': email.invalid && (email.dirty || email.touched),  'has-success': email.valid && (email.dirty || email.touched)  }">  <label>Email</label>  <input *type*="email" *class*="form-control" *name*="email"  *[(ngModel)]*="model.email" *required* *pattern*="[^ @]\*@[^ @]\*" *#email*="ngModel">  <div *class*="form-control-feedback" *\*ngIf*="email.errors && (email.dirty || email.touched)">  <p *\*ngIf*="email.errors.required">Email is required</p>  <p *\*ngIf*="email.errors.pattern">Email must contain at least the @ character</p>  </div>  </div>  <div *class*="form-group" *[ngClass]*="{  'has-danger': password.invalid && (password.dirty || password.touched),  'has-success': password.valid && (password.dirty || password.touched)  }">  <label>Password</label>  <input *type*="password" *class*="form-control" *name*="password"  *[(ngModel)]*="model.password" *required* *minlength*="8" *#password*="ngModel">    <div *class*="form-control-feedback" *\*ngIf*="password.errors && (password.dirty || password.touched)">  <p *\*ngIf*="password.errors.required">Password is required</p>  <p *\*ngIf*="password.errors.minlength">Password must be at least 8 characters long</p>  </div>  </div>  <div *class*="form-group">  <label>Language</label>  <select *class*="form-control" *name*="language" *[(ngModel)]*="model.language">  <option *value*="">Please select a language</option>  <option *\*ngFor*="let lang of langs" *[value]*="lang">{{lang}}  </option>  </select>  </div>  <button *type*="submit" *class*="btn btn-primary" *[disabled]*="myFormTemplateRef.invalid">Submit  </button>  <pre>{{myFormTemplateRef.value | json}}</pre>  </form>  `  })  class **TemplateFormComponent** {  *// Domain Model:*  model: **Signup** = new **Signup**();  @**ViewChild**('myFormTemplateRef') form: any;  langs: string[] = [  'English',  'French',  'German',  ];  **onSubmit**() {  if (this.form.valid) {  **console**.**log**("Form Submitted!", this.form.value);  **console**.**log**(this.form.value);  **console**.**log**(this.model);  this.form.**reset**();  }  }  }  @**Component**({  selector: 'app',  template: `<template-form></template-form>`  })  class **AppComponent** {  }  @**NgModule**({  imports: [  BrowserModule,  FormsModule  ],  declarations: [  AppComponent,  TemplateFormComponent  ],  bootstrap: [  AppComponent  ],  })  class **AppModule** {  }  **platformBrowserDynamic**().**bootstrapModule**(AppModule); |  (Reactive) Model Driven Approach:  |  | | --- | | import { NgModule, Component, OnInit } from '@angular/core';  import { ReactiveFormsModule, FormControl } from '@angular/forms';  import { BrowserModule } from '@angular/platform-browser';  import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';  import 'rxjs/Rx';  @**Component**({  selector: 'reactive-model-form',  template: `    <input *type*="search"  *class*="form-control"  *placeholder*="Please enter search term"  *[formControl]*="searchField">  <hr/>  <ul>  <li *\*ngFor*="let search of searches">{{ search }}</li>  </ul>  `  })  class **ReactiveModelFormComponent** implements **OnInit** {  searchField: **FormControl**;  searches: string[] = [];  **ngOnInit**() {  this.searchField = new **FormControl**();  this.searchField.valueChanges  .**debounceTime**(400)  .**distinctUntilChanged**()  .**subscribe**(term => {  this.searches.**push**(term);  });  }  }  @**Component**({  selector: 'app',  template: `<reactive-model-form></reactive-model-form>`  })  class **AppComponent** {  }  @**NgModule**({  imports: [  BrowserModule,  ReactiveFormsModule  ],  declarations: [  AppComponent,  ReactiveModelFormComponent  ],  bootstrap: [  AppComponent  ],  })  class **AppModule** {  }  **platformBrowserDynamic**().**bootstrapModule**(AppModule); |   … |

Dependency Injection & Providers

|  |
| --- |
| Dependency Injection & Providers |
| Dependency Injection:   * Injecting dependent objects   class **MailChimpService** extends **EmailService** { }  class **EmailSender** {  emailService: **EmailService**;  constructor() {    this.emailService = new **MailChimpService**("APIKEY12345678910");  }  **sendEmail**(mail: **Mail**) {  this.emailService.**sendEmail**(mail);  }  }  emailSender = new **EmailSender**();  emailSender.**sendEmail**(mail);  *// 1. Tightly coupled:*  *// - MailChimpService is hardcoded here*  *// - Incase, if i wanted to use another email provider?*  *// - Incase, if MailChimp service provider asked to send username & password instead of API key*  *// 2. Hard to Test:*  *// - Since 'MailChimpService' is real api service, We cannot change that to 'MockService'*  class **MailChimpService** extends **EmailService** { }  class **EmailSender** {  emailService: **EmailService**;  constructor(emailService: **EmailService**) {  this.emailService = emailService;  }  **sendEmail**(mail: **Mail**) {  this.emailService.**sendEmail**(mail);  }  }  emailSender = new **EmailSender**(new **MailChimpService**());  emailSender.**sendEmail**(mail);  **1. Flexible/Easier to re-use**  We can re-use the EmailSender class but with a different email service.  For example if we wanted to use SendGridService instead of MailChimpService.  emailSender = new **EmailSender**(new **SendGridService**());  emailSender.**sendEmail**(mail);  **2. Easier to test**  MockedEmailService extends EmailService {  mailSent: boolean = false;    **sendEmail**(mail: Mail) {  this.mailSent = true;  }  }    let mockService = **MockedEmailService**()  emailSender = new **EmailSender**(mockService);    if (mockService.mailSent === true) {  *// test passed*  }  3. Easier to maintain   * Since the **EmailSender** class is not responsible for creating concrete instances of the email service * Incase, the MailChimpService required some new configuration then the EmailSender class isn’t affected.  Inversion of Control :  * This ‘idea’ of **moving the responsibility** of creating concrete instances of dependency’s to something else is called Inversion of Control, or IoC.   Inversion of Control:   * One of the Implementation is : Dependency Injection ((Design Pattern) * The specific design pattern for implementing IoC above is called Dependency Injection,   + we injected the dependencies of EmailSender in the constructor.   Dependency injection is an important application design pattern it’s used not only in Angular but throughout software development as a whole.  **Angular & DI :**   * Angular has its own dependency injection framework * and we really can’t build an Angular application without it. * everyone just calls it DI.   **DI framework** in Angular consists of 4 concepts working together:  **Token**   * This uniquely identifies something that we want injected. A dependancy of our code.   **Dependancy**   * The actual code we want injected.   **Provider**   * This is a map between a token and a list of dependancies.   **Injector**   * This is a function which when passed a token returns a dependancy (or a list of dependencies) == Summary   … |

|  |
| --- |
| Injectors |
| import {ReflectiveInjector} from '@angular/core';  import {OpaqueToken} from '@angular/core';  *// ##1. Simple Injector Example*  {  **console**.**log**("Simple Injector Example");  class **MandrillService** {  }  class **SendGridService** {  }  *// 1. Created 'injector' and configured list of dependencies*  let injector = ReflectiveInjector.**resolveAndCreate**([  MandrillService,  SendGridService  ]);  *// 2. Resolve a dependecy*  *// We pass in a token, the class name 'MandrillService', into our injector and ask it to resolve to a dependency.*  *// - In this case it simply returns an instance of MandrillService.*  let emailService = injector.**get**(MandrillService);  **console**.**log**(emailService);  *// The injector doesn’t return the class, but an instance of the class instantiated with 'new'*  *// emailService = new MandrillService()*    *// Injector Caching Example*  {  **console**.**log**("Injector Caching Example");  let emailService1 = injector.**get**(MandrillService);  let emailService2 = injector.**get**(MandrillService);  *// Both are Same Instance*  **console**.**log**(emailService1 === emailService2); *// true*  *// The dependencies returned from injectors are cached.*  *// -So multiple calls to the same injector for the same token will return the same instance*  *// Note:*  *// A different injector for the same token might return a different instance of a dependency*  *// - but the same injector will always return the same instance.*  }  *// Injector Caching Caching State Sharing Example*  {  **console**.**log**("Injector Caching Caching State Sharing Example");  let emailService1 = injector.**get**(MandrillService);  emailService1.foo = "moo";  let emailService2 = injector.**get**(MandrillService);  **console**.**log**(emailService2.foo); *// moo*  }  }  *// ##2. Child Injector Returns Different Instance*  {  **console**.**log**("Child Injector Returns Different Instance");  class **EmailService** {  }    let injector = ReflectiveInjector.**resolveAndCreate**([EmailService]);  let childInjector = injector.**resolveAndCreateChild**([EmailService]);    let emailService1 = injector.**get**(EmailService);  let emailService2 = childInjector.**get**(EmailService);    *// Both are NOT Same Instance*  **console**.**log**(emailService1 === emailService2); *// true*    *// I’ve mentioned previously that different injectors return different instances of dependencies,*  *// this is also true even if the injector is a child injector*  }  *// ##3. Child Injector Forwards Request to Parent*  {  **console**.**log**("Child Injector Forwards Request to Parent");  class **EmailService** {  }    let injector = ReflectiveInjector.**resolveAndCreate**([EmailService]);  let childInjector = injector.**resolveAndCreateChild**([]);    let emailService1 = injector.**get**(EmailService);  let emailService2 = childInjector.**get**(EmailService);    *// Both are Same Instance*  **console**.**log**(emailService1 === emailService2); *// true*  *// Child injectors forward requests to their parent injector, if they can’t resolve the token locally.*  *// We request the token EmailService from the childInjector,*  *// it can’t find that token locally*  *// so it asks it’s parent injector which returns the instance it had cached from a previous direct reqeust.*  }    … |

|  |
| --- |
| Provider |
| Provider configurations   1. useClass 2. useExisting 3. useValue 4. useFactory  1. useClass : **Short Form:**  let injector = ReflectiveInjector.**resolveAndCreate**([  MandrillService,  SendGridService  ]);  **Long Form:** (both shares same functionality)  let injector = ReflectiveInjector.**resolveAndCreate**([  { provide: MandrillService, useClass: MandrillService },  { provide: SendGridService, useClass: SendGridService },  ]);  **Switching dependencies:**   * If we wanted to re-use our application and move from **Mandrill** to **SendGrid** without using DI * we would have to search through all the code for where we have requested **MandrillService** to be injected and replace with **SendGridService**. * A better solution is   import { ReflectiveInjector } from '@angular/core';  class **MandrillService** {};  class **SendGridService** {};  *// The token is "EmailService" and the dependency is the class 'MandrillService'*  let injector = ReflectiveInjector.**resolveAndCreate**([  { provide: "EmailService", useClass: MandrillService }  ]);  let emailService = injector.**get**("EmailService");  **console**.**log**(emailService); *// new MandrillService()*    To move from **Mandrill** to **SendGrid,** we just have to change once place  let injector = ReflectiveInjector.**resolveAndCreate**([  { provide: "EmailService", useClass: SendGridService }  ]); 2. useExisting : import { ReflectiveInjector } from '@angular/core';  class **MandrillService** {};  class **SendGridService** {};  class **GenericEmailService** {};  let injector = ReflectiveInjector.**resolveAndCreate**([  { provide: GenericEmailService, useClass: GenericEmailService },  { provide: MandrillService, useExisting: GenericEmailService },  { provide: SendGridService, useExisting: GenericEmailService }  ]);  let emailService1 = injector.**get**(SendGridService);  **console**.**log**(emailService1); *// GenericEmailService {}*  let emailService2 = injector.**get**(MandrillService);  **console**.**log**(emailService2); *// GenericEmailService {}*  let emailService3 = injector.**get**(GenericEmailService);  **console**.**log**(emailService3); *// GenericEmailService {}*  **console**.**log**(emailService1 === emailService2); *// true*  **console**.**log**(emailService2 === emailService3); *// true*  … 3. useValue :… import { ReflectiveInjector } from '@angular/core';  let injector1 = ReflectiveInjector.**resolveAndCreate**([  { provide: "APIKey", useValue: 'XYZ1234ABC' }  ]);  let apiKey = injector1.**get**("APIKey");  **console**.**log**(apiKey); *// "XYZ1234ABC"*  let injector2 = ReflectiveInjector.**resolveAndCreate**([  { provide: "Config",  useValue: {  'APIKey': 'XYZ1234ABC',  'APISecret': '555-123-111'  }  }  ]);  let config = injector2.**get**("Config");  **console**.**log**(config); *// Object {APIKey: "XYZ1234ABC", APISecret: "555-123-111"}*  *// Configuration should be 'read-only', But it is NOT*  config['APIKey'] = 'New Key';  **console**.**log**(config); *// Object {APIKey: "New Key", APISecret: "555-123-111"}*  *// When you use 'object' -- we should manually make it immutable (using Object.freeze)*  let injector3 = ReflectiveInjector.**resolveAndCreate**([  { provide: "Config",  useValue: **Object**.**freeze**({  'APIKey': 'XYZ1234ABC',  'APISecret': '555-123-111'  })  }  ]); 4.useFactory : import { ReflectiveInjector } from '@angular/core';  class **MandrillService** {};  class **SendGridService** {};  const isProd = true;  let injector = ReflectiveInjector.**resolveAndCreate**([  {  provide: "EmailService",  **useFactory**: () => {  if (isProd) {  return new **MandrillService**();  } else {  return new **SendGridService**();  }  }  },  ]);  let emailService1 = injector.**get**("EmailService");  **console**.**log**(emailService1); *// MandrillService {}*  …  … |

|  |
| --- |
| Tokens |
| import { ReflectiveInjector } from '@angular/core';  import { InjectionToken } from '@angular/core';  *// Using 'String Token':*  {  class **MandrillService** { };  class **SendGridService** { };  let injector = ReflectiveInjector.**resolveAndCreate**([  { provide: "EmailService", useClass: MandrillService }  ]);  let emailService = injector.**get**("EmailService");  **console**.**log**(emailService); *// new MandrillService()*  }  *// Using 'Type Token':*  {  class **EmailService** { };  class **MandrillService** extends **EmailService** { };  class **SendGridService** extends **EmailService** { };  let injector = ReflectiveInjector.**resolveAndCreate**([  { provide: EmailService, useClass: MandrillService }  ]);  let emailService = injector.**get**(EmailService); *// new MandrillService()*  **console**.**log**(emailService);  }  *// Using 'String Token' (Fail Case):*  {  **console**.**log**("String Token (Fail Case) Example");  class **MandrillService** {  }  class **SendGridService** {  }  let MandrillServiceToken = "EmailService";  let SendGridServiceToken = "EmailService";  let injector = ReflectiveInjector.**resolveAndCreate**([  { provide: SendGridServiceToken, useClass: SendGridService },  { provide: MandrillServiceToken, useClass: MandrillService },  ]);  let emailService1 = injector.**get**(SendGridServiceToken); *// new MandrillService()*  let emailService2 = injector.**get**(MandrillServiceToken); *// new MandrillService()*  **console**.**log**(emailService1 === emailService2);  }  *// Using 'InjectionToken':*  {  **console**.**log**("InjectionToken");  class **MandrillService** {  }  class **SendGridService** {  }  const MandrillServiceToken = new **InjectionToken**<string>("EmailService");  const SendGridServiceToken = new **InjectionToken**<string>("EmailService");  let injector = ReflectiveInjector.**resolveAndCreate**([  { provide: SendGridServiceToken, useClass: SendGridService },  { provide: MandrillServiceToken, useClass: MandrillService },  ]);  let emailService1 = injector.**get**(SendGridServiceToken); *// new SendGridService()*  let emailService2 = injector.**get**(MandrillServiceToken); *// new MandrillService()*  **console**.**log**(emailService1 === emailService2); *// false*  }  … |

|  |
| --- |
| Configuring Dependency Injection in Angular |
| Injector Tree   |  |  | | --- | --- | |  | * Angular app will have a **“tree of injectors”** mirroring the **“component tree”** * We have a top level parent injector which is attached to our **NgModule**. * Then we have child injectors descending in a hierarchy matching the component tree. * So a parent component will have a child injector stemming from NgModule. * A child component of parent component will have a child injector stemming from Parent. |   **Many ways we can inject:**   |  | | --- | | *// The NgModule decorator has a property called 'providers' which accepts a list of providers*  *// exactly the same as we would pass to the ReflectiveInjector via the resolveAndCreate function*  @**NgModule**({  ...  providers: [EmailService, JobService]  })  class **AppModule** { }  *// This creates a top level parent injector and configures it with two class providers, EmailService and JobService.*  *// -or-*  *// We can also configure our 'Components' and 'Directives' the same way using a property called providers*  @**Component**({  selector: 'my-comp',  template: `...`,  providers: [EmailService]  })  class **MyComponent** { }  *// This creates a child injector who’s parent injector is the injector on the parent component.*  *// If there is no parent component then the parent injector is the top level NgModule injector.*  *// -or-*  *// viewProviders is a special injector that resolves dependencies only for this components view children*  *// and doesn’t act as a parent injector for any content children*  @**Component**({  selector: 'my-comp',  template: `...`,  viewProviders: [EmailService]  })  class **MyComponent** { } |   **DI Example:**   |  | | --- | | import {NgModule, Component, Injectable, Inject, TypeDecorator} from '@angular/core';  import {BrowserModule} from '@angular/platform-browser';  import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';  class **OtherService** {  constructor() {  };  }  *// This version doesn't work as Angular doesn't know it should be injecting otherService*  *// class SimpleService {*  *// otherService: OtherService;*  *// constructor(otherService: OtherService) {*  *// this.otherService = otherService;*  *// };*  *// }*  *// This version works but we have to decorate every parameter to our constructor with @Inject*  *// class SimpleService {*  *// otherService: OtherService;*  *//*  *// constructor(@Inject(OtherService) otherService: OtherService) {*  *// this.otherService = otherService;*  *// };*  *//*  *// }*  *// This works because @Injectable automatically injects every parameter to the constructor as long as that parameter has a type*  @**Injectable**()  class **SimpleService** {  otherService: **OtherService**;  constructor(otherService: **OtherService**) {  this.otherService = otherService;  };  }  *// This DOESN'T work because the otherService parameter doesn't have a type*  *// @Injectable*  *// class SimpleService {*  *// otherService: OtherService;*  *//*  *// constructor(otherService: any) {*  *// this.otherService = otherService;*  *// };*  *// }*  @**Component**({  selector: 'simple',  template: `<p>Simple is as simple does</p>`,  })  class **SimpleComponent** {  constructor(private simpleService: **SimpleService**) {  }  }  @**Component**({  selector: 'app',  template: '<simple></simple>'  })  class **AppComponent** {  }  @**NgModule**({  imports: [BrowserModule],  declarations: [AppComponent, SimpleComponent],  bootstrap: [AppComponent],  providers: [OtherService, SimpleService]  })  class **AppModule** {  }  **platformBrowserDynamic**().**bootstrapModule**(AppModule); |  NgModule.providers VS Component.providers VS Component.viewProviders : **NgModule.providers:**   |  | | --- | |  | | import { NgModule, Component, Injectable } from '@angular/core';  import { FormsModule } from '@angular/forms';  import { BrowserModule } from '@angular/platform-browser';  import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';  class **SimpleService** {  value: string;  }  @**Component**({  selector: 'child',  template: `  <div *class*="child">  <p>Child</p>  {{ service.value }}  </div>  `,  styles: [`  .child {  background-color: #239CDE;  padding: 10px;  }  `],  })  class **ChildComponent** {  *// Getting 'new SimpleService()' instance from 'ParentComponent'*  *// 'ParentComponent' gets 'SimpleService instance' from NgModule'*  constructor(private service: **SimpleService**) { }  }  @**Component**({  selector: 'parent',  template: `  <div *class*="parent">  <p>Parent</p>  <form *novalidate*>            <div *class*="form-group">            <input *type*="text"  *class*="form-control"  *name*="value"  *[(ngModel)]*="service.value">        </div>  </form>  <ng-content></ng-content>  </div>  `,  styles: [`  .parent {  background-color: #D1E751;  padding: 10px;  }  `],  })  class **ParentComponent** {  *// Getting 'new SimpleService()' instance from 'NgModule'*  constructor(private service: **SimpleService**) { }  }  @**Component**({  selector: 'app',  template: `  <div *class*="row">      <div *class*="col-xs-6">          <parent><child></child></parent>      </div>      <div *class*="col-xs-6">          <parent><child></child></parent>      </div>  </div>  `  })  class **AppComponent** {  }  @**NgModule**({  imports: [BrowserModule, FormsModule],  declarations: [AppComponent, ParentComponent, ChildComponent],  bootstrap: [AppComponent],  providers: [SimpleService]  })  class **AppModule** {  }  **platformBrowserDynamic**().**bootstrapModule**(AppModule); | | **Output:**    // Since everthing gets same instance, it all shares the same value across all the components |   **Component.providers:**   |  | | --- | |  | | import { NgModule, Component, Injectable } from '@angular/core';  import { FormsModule } from '@angular/forms';  import { BrowserModule } from '@angular/platform-browser';  import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';  class **SimpleService** {  value: string;  }  @**Component**({  selector: 'child',  template: `  <div *class*="child">  <p>Child</p>  {{ service.value }}  </div>  `,  styles: [`.child { background-color: #239CDE;padding: 10px;}`],  })  class **ChildComponent** {  *// Getting 'new SimpleService()' instance from 'ParentComponent'*  constructor(private service: **SimpleService**) { }  }  @**Component**({  selector: 'parent',  template: `  <div *class*="parent">  <p>Parent</p>  <form *novalidate*>            <div *class*="form-group">            <input *type*="text"  *class*="form-control"  *name*="value"  *[(ngModel)]*="service.value">        </div>  </form>  <ng-content></ng-content>  </div>  `,  styles: [`.parent {background-color: #D1E751; padding: 10px;}`],  *// Since ParentComponent itself has own provider 'SimpleService',*  *// each Parent component gets new instance of 'SimpleService'*  *// Anyway each parent's child component shares the same value*  providers: [SimpleService]  })  class **ParentComponent** {  *// Getting 'new SimpleService()' instance from 'ParentComponent' itself*  constructor(private service: **SimpleService**) { }  }  @**Component**({  selector: 'app',  template: `  <div *class*="row">      <div *class*="col-xs-6">          <parent><child></child></parent>      </div>      <div *class*="col-xs-6">          <parent><child></child></parent>      </div>  </div>  `  })  class **AppComponent** {  }  @**NgModule**({  imports: [BrowserModule, FormsModule],  declarations: [AppComponent, ParentComponent, ChildComponent],  bootstrap: [AppComponent],  providers: [SimpleService]  })  class **AppModule** {  }  **platformBrowserDynamic**().**bootstrapModule**(AppModule); | | **Output:** |   **Component.viewProviders:**   |  | | --- | |  | | import { NgModule, Component, Injectable } from '@angular/core';  import { FormsModule } from '@angular/forms';  import { BrowserModule } from '@angular/platform-browser';  import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';  class **SimpleService** {  value: string;  }  @**Component**({  selector: 'child',  template: `  <div *class*="child">  <p>Child</p>  {{ service.value }}  </div>  `,  styles: [`.child { background-color: #239CDE;padding: 10px;}`],  })  class **ChildComponent** {  *// Getting 'new SimpleService()' instance from 'ParentComponent'*  constructor(private service: **SimpleService**) { }  }  @**Component**({  selector: 'parent',  template: `  <div *class*="parent">  <p>Parent</p>  <form *novalidate*>            <div *class*="form-group">            <input *type*="text"  *class*="form-control"  *name*="value"  *[(ngModel)]*="service.value">        </div>  </form>    *<!-- 'ng-content' cannot get the same Parent's 'SimpleService instance' -->*  <ng-content></ng-content>    *<!-- 'child' can get the same Parent's 'SimpleService instance' -->*  <child></child>    </div>  `,  styles: [`.parent {background-color: #D1E751; padding: 10px;}`],  *// Since ParentComponent has viewProviders 'SimpleService',*  *// each Parent component gets new instance of 'SimpleService'*  *// each parent's child component shares the same value*  *// But it cannot share the same instance of 'SimpleService' to <ng-content>'s child*  viewProviders: [SimpleService]  })  class **ParentComponent** {  *// Getting 'new SimpleService()' instance from 'NgModule'*  constructor(private service: **SimpleService**) { }  }  @**Component**({  selector: 'app',  template: `  <div *class*="row">      <div *class*="col-xs-6">          <parent><child></child></parent>      </div>      <div *class*="col-xs-6">          <parent><child></child></parent>      </div>  </div>  `  })  class **AppComponent** {  }  @**NgModule**({  imports: [BrowserModule, FormsModule],  declarations: [AppComponent, ParentComponent, ChildComponent],  bootstrap: [AppComponent],  providers: [SimpleService]  })  class **AppModule** {  }  **platformBrowserDynamic**().**bootstrapModule**(AppModule); | | **Output:** |   **Summary:**   * We can configure the DI framework in Angular in three main ways. * We can configure a **provider** on the **NgModule**, on a **component** or **directives** **providers** property and on a **components** **viewProviders** property. * Deciding where to configure your provider and understanding the different is key in understanding how to architect an Angular application. * If we want an instance of a dependency to be shared globally and share state across the application we configure it on the **NgModule**. * If we want a separate instance of a dependency to be shared across each instance of a component and it’s children we configure it on the components **providers** property. * If we want a separate instance of a dependency to be shared across each instance of a component and only it’s view children we configure it on the components **viewProviders** property.   … |

# HTTP

|  |
| --- |
| 1. Core HTTP API |
| import { NgModule, Component } from '@angular/core';  import { HttpModule, Http, URLSearchParams, Headers, RequestOptions } from '@angular/http';  import { BrowserModule } from '@angular/platform-browser';  import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';  import 'rxjs/add/operator/toPromise';  import 'rxjs/Rx';  @**Component**({  selector: 'app',  template: `  <div *class*="row">  <div *class*="m-t-1">    <button *class*="btn btn-primary" *(click)*="**doGET**()">GET</button>    <button *class*="btn btn-primary" *(click)*="**doPOST**()">POST</button>    <button *class*="btn btn-primary" *(click)*="**doPUT**()">PUT</button>    <button *class*="btn btn-primary" *(click)*="**doDELETE**()">DELETE</button>      </div>  </div>  <div *class*="row">  <div *class*="m-t-1">    <button *class*="btn btn-secondary" *(click)*="**doGETAsPromise**()">As Promise</button>    <button *class*="btn btn-secondary" *(click)*="**doGETAsPromiseError**()">Error as Promise</button>    <button *class*="btn btn-secondary" *(click)*="**doGETAsObservableError**()">Error as Observable</button>      </div>  </div>  <div *class*="row">  <div *class*="m-t-1">    <button *class*="btn btn-danger" *(click)*="**doGETWithHeaders**()">With Headers</button>      </div>  </div>  `  })  class **AppComponent** {  apiRoot: string = "http://httpbin.org";  constructor(private http: **Http**) {  }  **doGET**() {  **console**.**log**("GET");  let url = `${this.apiRoot}/get`;  *// Query Params:*  let urlSearchParams = new **URLSearchParams**();  urlSearchParams.**set**('param1', 'value1');  urlSearchParams.**set**('param2', 'value2');  *// Make HTTP request:*  this.http.**get**(url, { search: urlSearchParams }).**subscribe**(res => {  *// Prints: whatever was returned in the HTTP response body from the server*  *// It might be normal 'string' or 'string has json format '*  **console**.**log**(res.**text**());  *// To convert the JSON formatted string into an object*  **console**.**log**(res.**json**());  });  }  **doPOST**() {  **console**.**log**("POST");  let url = `${this.apiRoot}/post`;  *// Query Params:*  let urlSearchParams = new **URLSearchParams**();  urlSearchParams.**set**('param1', 'value1');  urlSearchParams.**set**('param2', 'value2');  *// Payload:*  let payload = { key1: 'value1', key2: 'value2' };  this.http.**post**(url, payload, { search: urlSearchParams }).**subscribe**(  res => **console**.**log**(res.**json**())  );  }  **doPUT**() {  **console**.**log**("PUT");  let url = `${this.apiRoot}/put`;  *// Query Params:*  let urlSearchParams = new **URLSearchParams**();  urlSearchParams.**set**('param1', 'value1');  urlSearchParams.**set**('param2', 'value2');  *// Payload:*  let payload = { key1: 'value1', key2: 'value2' };  this.http.**put**(url, payload, { search: urlSearchParams }).**subscribe**(  res => **console**.**log**(res.**json**())  );  }  **doDELETE**() {  **console**.**log**("DELETE");  let url = `${this.apiRoot}/delete`;  *// Query Params:*  let urlSearchParams = new **URLSearchParams**();  urlSearchParams.**set**('param1', 'value1');  urlSearchParams.**set**('param2', 'value2');  this.http.**delete**(url, { search: urlSearchParams }).**subscribe**(  res => **console**.**log**(res.**json**())  );  }  **doGETAsPromise**() {  **console**.**log**("GET AS PROMISE");  let url = `${this.apiRoot}/get`;  this.http.**get**(url)  .**toPromise**()  .**then**(res => **console**.**log**(res.**json**()));  }  **doGETAsPromiseError**() {  **console**.**log**("GET AS PROMISE ERROR");  let url = `${this.apiRoot}/post`;  this.http.**get**(url)  .**toPromise**()  .**then**(  res => **console**.**log**(res.**json**()),  errMsg => **console**.**error**(`Error: ${errMsg.status} ${errMsg.statusText}`)  );  */\* -or-*  *this.http.get(url)*  *.toPromise()*  *.then(*  *function (res) {*  *console.log(res.json())*  *},*  *function (errMsg) {*  *console.error(`Error: ${errMsg.status} ${errMsg.statusText}`)*  *}*  *);*  *\*/*  }  **doGETAsObservableError**() {  **console**.**log**("GET AS OBSERVABLE ERROR");  let url = `${this.apiRoot}/post`;  this.http.**get**(url).**subscribe**(  res => **console**.**log**(res.**json**()),  errMsg => **console**.**error**(`Error: ${errMsg.status} ${errMsg.statusText}`)  );  }  **doGETWithHeaders**() {  **console**.**log**("GET WITH HEADERS");  let url = `${this.apiRoot}/get`;  *// Query Params:*  let urlSearchParams = new **URLSearchParams**();  urlSearchParams.**set**('param1', 'value1');  urlSearchParams.**set**('param2', 'value2');  *// with 'RequestOptions' (headers):*  let headers: **Headers** = new **Headers**();  headers.**append**('Authorization', **btoa**('username:password'));  let reqOptions: **RequestOptions** = new **RequestOptions**();  reqOptions.headers = headers;  reqOptions.search = urlSearchParams;    this.http.**get**(url, reqOptions).**subscribe**(  res => **console**.**log**(res.**json**()),  errMsg => **console**.**error**(`Error: ${errMsg.status} ${errMsg.statusText}`)  );  }  }  *// Importing 'HttpModule' into our NgModule*  *// configures our NgModules injector with all the providers needed to use Http in our app.*  @**NgModule**({  imports: [BrowserModule, HttpModule],  declarations: [AppComponent],  bootstrap: [AppComponent]  })  class **AppModule** {  }  **platformBrowserDynamic**().**bootstrapModule**(AppModule); |

|  |
| --- |
| HTTP Example with Promises |
| import { NgModule, Component, Injectable } from '@angular/core';  import { BrowserModule } from '@angular/platform-browser';  import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';  import { HttpModule, Http, Response } from '@angular/http';  import 'rxjs/Rx';  class **SearchItem** {  constructor(public track: string,  public artist: string,  public link: string,  public thumbnail: string,  public artistId: string) {  }  }  @**Injectable**()  export class **ItunesSearchService** {  apiRoot: string = 'https://itunes.apple.com/search';  results: **SearchItem**[];  loading: boolean;  constructor(private http: **Http**) {  this.results = [];  this.loading = false;  }  **search**(searchKeyword: string) {  let promise = new **Promise**((resolve, reject) => {  let apiURL = `${this.apiRoot}?term=${searchKeyword}&media=music&limit=20`;  *// 'http.get()' returns Observables*  *// converting the Observables into 'Promise' –using 'toPromise()'*  this.http.**get**(apiURL)  .**toPromise**()  .**then**(  res => {  *// Success:*  *// this.results = res.json().results;*  this.results = res.**json**().results.**map**(item => {  *// transform response:*  return new **SearchItem**(  item.trackName,  item.artistName,  item.trackViewUrl,  item.artworkUrl30,  item.artistId  );  });  **resolve**();  },  msg => {  *// Error:*  **reject**(msg);  }  );  });  return promise;  }  }  @**Component**({  selector: 'app',  template: `  <form *class*="form-inline">  <div *class*="form-group">  <input *type*="search" *class*="form-control" *placeholder*="Enter search string" *#search*>  </div>  <button *type*="button" *class*="btn btn-primary" *(click)*="**doSearch**(search.value)">Search</button>    </form>  <hr/>  <div *class*="text-center">  <p *class*="lead" *\*ngIf*="loading">Loading...</p>  </div>  <ul *class*="list-group">  <li *class*="list-group-item" *\*ngFor*="let track of itunesSearchService.results">  <img *src*="{{track.thumbnail}}">  <a *target*="\_blank" *href*="{{track.link}}">{{ track.track }}  </a>  </li>  </ul>  `  })  class **AppComponent** {  private loading: boolean = false;  constructor(private itunesSearchService: **ItunesSearchService**) {  }  **doSearch**(searchKeyword: string) {  this.loading = true;  this.itunesSearchService.**search**(searchKeyword)  .**then**(\_ => this.loading = false);  }  }  @**NgModule**({  imports: [  BrowserModule,  HttpModule,  ],  declarations: [AppComponent],  bootstrap: [AppComponent],  providers: [ItunesSearchService]  })  class **AppModule** {  }  **platformBrowserDynamic**().**bootstrapModule**(AppModule); |

|  |
| --- |
| HTTP Example with Observables |
| |  | | --- | | import { NgModule, Component, Injectable } from '@angular/core';  import { BrowserModule } from '@angular/platform-browser';  import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';  import { HttpModule, Http, Response } from '@angular/http';  import { ReactiveFormsModule, FormControl, FormsModule } from '@angular/forms';  import { Observable } from 'rxjs';  import 'rxjs/add/operator/map';  import 'rxjs/add/operator/debounceTime';  import 'rxjs/add/operator/distinctUntilChanged';  import 'rxjs/add/operator/switchMap';  import 'rxjs/add/operator/do';  class **SearchItem** {  constructor(public track: string,  public artist: string,  public link: string,  public thumbnail: string,  public artistId: string) {  }  }  @**Injectable**()  export class **ItunesSearchService** {  apiRoot: string = 'https://itunes.apple.com/search';  constructor(private http: **Http**) {  }  **search**(searchKeyword: string): **Observable**<**SearchItem**[]> {  let apiURL = `${this.apiRoot}?term=${searchKeyword}&media=music&limit=20`;  *// 'http.get()' returns Observable*  return this.http.**get**(apiURL)  *// This 'map' is not a normal JavaScript fn*  *// It is RxJS 'map' operator*  *// which helps us convert 'response' into array of 'SearchItem' model*  .**map**(res => {  */\**  *let searchResults = res.json().results;*  *let transFormedSearchResults = searchResults.map(item => {*  *return new SearchItem(*  *item.trackName,*  *item.artistName,*  *item.trackViewUrl,*  *item.artworkUrl30,*  *item.artistId*  *);*  *});*  *return transFormedSearchResults;*  *\*/*  return res.**json**().results.**map**(item => {  *// This 'map' is a normal JavaScript fn*  return new **SearchItem**(  item.trackName,  item.artistName,  item.trackViewUrl,  item.artworkUrl30,  item.artistId  );  });    });  }  }  @**Component**({  selector: 'app',  template:  `  <form *class*="form-inline">  <div *class*="form-group">  <input *type*="search" *class*="form-control" *placeholder*="Enter search string"  *[formControl]*="searchField">  </div>  </form>  <div *class*="text-center">  <p *class*="lead" *\*ngIf*="loading">Loading...</p>  </div>  <ul *class*="list-group">  <li *class*="list-group-item" *\*ngFor*="let track of results | async">  <img *src*="{{track.thumbnail}}">  <a *target*="\_blank" *href*="{{track.link}}">{{ track.track }}  </a>  </li>  </ul>  `  })  class **AppComponent** {  private loading: boolean = false;  private results: **Observable**<**SearchItem**[]>;  private searchField: **FormControl**;  constructor(private itunesSearchService: **ItunesSearchService**) {  }  **ngOnInit**() {  this.searchField = new **FormControl**();  this.results = this.searchField.valueChanges  .**debounceTime**(400)  .**distinctUntilChanged**()  .**do**(\_ => this.loading = true)  .**switchMap**(searchKeyword => {  return this.itunesSearchService.**search**(searchKeyword)  })  .**do**(\_ => this.loading = false)  }  }  @**NgModule**({  imports: [  BrowserModule,  ReactiveFormsModule,  FormsModule,  HttpModule  ],  declarations: [AppComponent],  bootstrap: [AppComponent],  providers: [ItunesSearchService]  })  class **AppModule** {  }  **platformBrowserDynamic**().**bootstrapModule**(AppModule); |  Why switchMap() here ?  |  | | --- | | **ngOnInit**() {  this.searchField = new **FormControl**();    *// Two Observables*  *// 1. (input textbox stream..)*  *// 2. (http resp stream...)*  *// to solve this, we need 'two subscribes'*  this.searchField.valueChanges  .**debounceTime**(400)  .**distinctUntilChanged**()  *// returns: Observable<string>*  *// Transformation:*  .**map**(searchKeyword => {  *// input (searchKeyword): string*  *// returns: Observable<SearchItem[]>*  this.itunesSearchService.**search**(searchKeyword)  })  *// returns: Observable<Observable<SearchItem[]>>*  .**subscribe**(value1 => {  *// input (value1): Observable<SearchItem[]>*  value1.**subscribe**(value2 => {  *// input (value2): SearchItem[]*  **console**.**log**(value2)  })  });    */\**  *Observable<string>*  *.*  *.--map()*  *.*  *V*  *Observable<Observable<SearchItem[]>>*  *.*  *.--subscribe(value1)*  *.*  *V*  *Observable<SearchItem[]>*  *.*  *.--subscribe(value2)*  *.*  *SearchItem[]*  *\*/*    *// This is a common problem with Observables*  *// so there is a better way, 'switch' operator*  */\**  *'switch' expects a stream of Observables,*  *- when it get an Observable pushed onto it’s input stream*  *- it unsubscribes from any previous Observables*  *- and subscribes to the new one*  *- and then emits any values from that Observable onto it’s output stream.*  *it coverts Observable<Observable<SearchItem[]>> into Observable<SearchItem[]>*  *which is exactly the problem we are solving with the two subscribes above.*  *\*/*  this.searchField.valueChanges  .**debounceTime**(400)  .**distinctUntilChanged**()  .**map**(searchKeyword => this.itunesSearchService.**search**(searchKeyword))  .**switch**()  .**subscribe**(value => **console**.**log**(value));  *// Using switch with map is such a common occurrence*  *// so there is a combined operator called 'switchMap'*  this.results = this.searchField.valueChanges  .**debounceTime**(400)  .**distinctUntilChanged**()  .**switchMap**(searchKeyword => this.itunesSearchService.**search**(searchKeyword))  } |   … |

|  |
| --- |
| JSONP Example with Observables |
| **CORS Problem:**   * This is a security measure implemented in all browsers that stops you from using an API in a potentially unsolicited way and most APIs protected by it. * Because of CORS if we tried to make a request to the iTunes API url with the http client library the browser would issue a CORS error. * JSONP is a method of performing API requests which go around the issue of CORS.   **JSONP Solution:**   * We ask browser to treat that API as a javascript file, like this   + <script *src*="https://itunes.apple.com/search?term=love&media=music&limit=20"></script> * The browser then just downloads the javascript file   + since browsers don’t check for CORS when downloading javascript files it a works around the issue of CORS. * Imagine the response from the server was {hello:'world'}, we don’t just want to download the file because:   + The browser would download the json and try to execute it,   + since JSON isn’t javascript **nothing** would actually **happen**. * We want to do something with the JSON once it’s downloaded, ideally call a function passing it the json data.   + So APIs that support JSONP return something that looks like javascript,   + for example   + process\_response({hello:'world'}); * Of course we need a function called process\_response ready and waiting in our application   + so it can be called but if we are using a framework that supports JSONP these details are handled for us   **We can only use JSONP when:**   * The API itself supports JSONP. * It needs to return the JSON response wrapped in a function and it usually lets us pass in the function name we want it to use as one of the query params. * We can only use it for GET requests, it doesn’t work for PUT/POST/DELETE and so on.  |  | | --- | | import { NgModule, Component, Injectable } from '@angular/core';  import { BrowserModule } from '@angular/platform-browser';  import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';  import { JsonpModule, Jsonp, Response } from '@angular/http';  import { ReactiveFormsModule, FormControl, FormsModule } from '@angular/forms';  import { Observable } from 'rxjs';  import 'rxjs/add/operator/map';  import 'rxjs/add/operator/debounceTime';  import 'rxjs/add/operator/distinctUntilChanged';  import 'rxjs/add/operator/switchMap';  import 'rxjs/add/operator/do';  class **SearchItem** {  constructor(public track: string,  public artist: string,  public link: string,  public thumbnail: string,  public artistId: string) {  }  }  @**Injectable**()  export class **SearchService** {  apiRoot: string = 'https://itunes.apple.com/search';  constructor(private jsonp: **Jsonp**) {  }  **search**(term: string) {  */\**  *- The iTunes API supports JSONP, we just need to tell it what name to use via the callback query parameter.*  *- We passed it the special string JSONP\_CALLBACK.*  *- Angular will replace JSONP\_CALLBACK with an automatically generated function name every time we make a request*  *\*/*  let apiURL = `${this.apiRoot}?term=${term}&media=music&limit=20&callback=JSONP\_CALLBACK`;  return this.jsonp.**request**(apiURL)  .**map**(res => {  return res.**json**().results.**map**(item => {  return new **SearchItem**(  item.trackName,  item.artistName,  item.trackViewUrl,  item.artworkUrl30,  item.artistId  );  });  });  }  }  @**Component**({  selector: 'app',  template: `  <form *class*="form-inline">  <div *class*="form-group">  <input *type*="search"  *class*="form-control"  *placeholder*="Enter search string"  *[formControl]*="searchField">  </div>  </form>  <div *class*="text-center">  <p *class*="lead" *\*ngIf*="loading">Loading...</p>  </div>  <ul *class*="list-group">  <li *class*="list-group-item"  *\*ngFor*="let track of results | async">  <img *src*="{{track.thumbnail}}">  <a *target*="\_blank"  *href*="{{track.link}}">{{ track.track }}  </a>  </li>  </ul>  `  })  class **AppComponent** {  private loading: boolean = false;  private results: **Observable**<**SearchItem**[]>;  private searchField: **FormControl**;  constructor(private itunes: **SearchService**) {  }  **ngOnInit**() {  this.searchField = new **FormControl**();  this.results = this.searchField.valueChanges  .**debounceTime**(400)  .**distinctUntilChanged**()  .**do**(\_ => this.loading = true)  .**switchMap**(term => this.itunes.**search**(term))  .**do**(\_ => this.loading = false)  }  }  @**NgModule**({  imports: [  BrowserModule,  ReactiveFormsModule,  FormsModule,  JsonpModule  ],  declarations: [AppComponent],  bootstrap: [AppComponent],  providers: [SearchService]  })  class **AppModule** {  }  **platformBrowserDynamic**().**bootstrapModule**(AppModule); |   … |

# Routing :

|  |
| --- |
| Routing Overview |
| **URL defines the current state of your application:**  **State**: is a computer science terms and means "all the stored information, at a given instant in time, to which program has access".   * An address in a URL can’t store that much information but it can store enough so that perhaps we can re-fetch some and re-calculate the rest to get to the same state as before. * Giving the a URL to someone else should enable them to bring up the same state in their browser.   **Server Side Routing:**  In traditional applications built with Server Side Routing when you change the URL in your browser, the browser makes a request to the server to return some HTML which it will display.    **Client Side Routing:**  When the URL changes in the browser we want our local application that’s running in the browser (the client) to handle the change, we don’t want the request sent to the server.  When we first navigate to a new site the server returns the html, javascript and css needed to render that page.   * All further changes to the URL are handled locally by the client application. * Typically the client application will make one or more API requests to get the information it needs to show the new page.   There is only ever a single page returned from the server, all further modifications of the page are handled by the client and that’s why it’s called a (SPA)**Single Page Application**.    The advantages of an SPA are:   * Can be faster.   + Instead of making a time-consuming request to a far away server every time the URL changes the client app updates the page much faster. * Less bandwidth required.   + We don’t send over a big html page for every URL change, instead we might just call a smaller API which returns just enough data to render the change in the page. * Convenience.   + Now a single developer can build most of the functionality of a site instead of splitting the effort between a front end and server side developer.   Angular has a couple of modules which let us implement our application as an SPA, the concept as a whole in Angular is called the Component Router and in this section you will learn how to build an SPA in Angular using the component router.  … |

|  |
| --- |
| Routes (with Parameterized, Optional Params, Nested) |
| import { NgModule, Component, Injectable } from '@angular/core';  import { BrowserModule } from '@angular/platform-browser';  import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';  import { Routes, RouterModule, Router, ActivatedRoute, CanActivate, CanActivateChild, CanDeactivate, ActivatedRouteSnapshot, RouterStateSnapshot } from "@angular/router";  *//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Components \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**  @**Component**({  selector: 'route4child2',  template:  `  <h4> Route4Child2Component </h4>  `  })  class **Route4Child2Component** {  }  @**Component**({  selector: 'route4child1',  template:  `  <h4> Route4Child1Component </h4>    <br/>    *<!-- if we want route to start from root, we must specify '/' -->*  *<!-- [routerLink]="['/route1']" ==> /#/route1 -->*    <a *class*="nav-link" *[routerLink]*="['/route1']">Go to Route1</a>    `  })  class **Route4Child1Component** {  }  @**Component**({  selector: 'route4',  template:  `  <h3> Route4Component </h3>  <hr/>  <ul *class*="nav nav-pills">  <li *class*="nav-item">  <a *class*="nav-link" *[routerLinkActive]*="['active']" *[routerLink]*="['./route4child1']">Route4child1</a>  </li>  <li *class*="nav-item">  <a *class*="nav-link" *[routerLinkActive]*="['active']" *[routerLink]*="['./route4child2']">Route4child2</a>  </li>  </ul>  <br/>  <router-outlet></router-outlet>  `  })  class **Route4Component** {  }    @**Component**({  selector: 'route3',  template:  `  <h3> Route3Component </h3>  `  })  class **Route3Component** {    }    @**Component**({  selector: 'route2',  template:  `  <h3> Route2Component </h3>  `  })  class **Route2Component** {    constructor(private route: **ActivatedRoute**) {  this.route.params.**subscribe**(params => {  **console**.**log**(params);  **console**.**log**(params['param1']);  **console**.**log**(params['param2']);  });  }    }  @**Component**({  selector: 'route1',  template:  `  <h3> Route1Component </h3>  `  })  class **Route1Component** {  }    @**Component**({  selector: 'app-header',  template: `  <nav *class*="navbar navbar-light bg-faded">  <ul *class*="nav navbar-nav">  *<!-- We can also use 'routerLinkActive' directive in routerLink's parent element -->*  <li *class*="nav-item" *[routerLinkActive]*="['active']">    *<!-- if u didnt specify slashes or specify './' it is (relative path) -->*  *<!-- [routerLink]="['route1']" or [routerLink]="['./route1']" same -->*  <a *class*="nav-link" *[routerLink]*="['route1']">Route1</a>    </li>  <li *class*="nav-item" *[class.active]*="router.**isActive**('route2')">  <a *class*="nav-link" *(click)*="**gotoRoute2**()">Route2</a>  </li>  <li *class*="nav-item" *[class.active]*="router.**isActive**('route3')">  <a *class*="nav-link" *(click)*="**gotoRoute3**()">Route3</a>  </li>  <li *class*="nav-item" *[routerLinkActive]*="['active']">  <a *class*="nav-link" *[routerLink]*="['route4', 101]">Route4</a>  </li>  </ul>  </nav>  <br/>  `  })  class **HeaderComponent** {  constructor(private router: **Router**) {  }  **gotoRoute2**(){    var param1 = 101;  var param2 = 'jagan';  *// this will navigate without Parameter*  *// ---> /#/route2*  *// this.router.navigate(['route2']);*    *// this will navigate as Parameterized Way with only param1*  *// ---> /#/route2/101*  *// this.router.navigate(['route2', param1]);*  *// this will navigate as Parameterized Way with param1, param2*  *// ---> /#/route2/101/jagan*  this.router.**navigate**(['route2', param1, param2]);      *// this will navigate as 'Optional Params'*  *// ---> /#/route2;param1=101;param2=jagan*  *//this.router.navigate(['route2', { param1: 101, param2: 'jagan'}]);*  }    **gotoRoute3**(){  *// this will navigate as 'Optional Params'*  *// ---> /#/route2;key1=value1;key2=value2*  this.router.**navigate**(['route3', { key1: 'value1', key2: 'value2'}]);  }  }  @**Component**({  selector: 'app',  template: `  <app-header></app-header>    <router-outlet></router-outlet>  `  })  class **AppComponent** {  }  *//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Routes \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**  const routes: **Routes** = [    *// if path has '' empty (pathMatch:'full') expects exact value*  { path: '', redirectTo: 'route1', pathMatch: 'full' },  *// 1. Simple Route*  { path: 'route1', component: Route1Component},    *// 2. Parameterized Route: //TODO: param1*  *// this.router.navigate(['route2']); -----> /#/route2*  { path: 'route2', component: Route2Component},  *// this.router.navigate(['route2', 'param1']); -----> /#/route2/param1/param2*  { path: 'route2/:param1', component: Route2Component},  *// this.router.navigate(['route2', 'param1', 'param2']); -----> /#/route2/param1*  { path: 'route2/:param1/:param2', component: Route2Component},    *// 3. Optional Parameters: (No need to define anything)*  *// this.router.navigate(['route3', { key1: 'value1' }]); -----> /#/route3;key1=value1*  { path: 'route3', component: Route3Component},    *// 4. Nested Route (with Parameterized)*  {  path: 'route4/:param1',  component: Route4Component,    *// Nested Route: '/#/route4/:param1/route4child1' or '/#/route4/:param1/route4child2'*  children: [  *// Nested Route (Default): E.g. '/#/route4/101' ---redirectTo--> '/#/route4/101/route4child1'*  { path: '', redirectTo: 'route4child1', pathMatch: 'full' },  { path: 'route4child1', component: Route4Child1Component },  { path: 'route4child2', component: Route4Child2Component }  ]  },      *// fallback: catch all route by using the path \*\*,*  *// normally we might show an error 404 page*  { path: '\*\*', component: Route1Component }  ];  *//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Bootstrap \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**  @**NgModule**({  imports: [  BrowserModule,  RouterModule.**forRoot**(routes, { useHash: true })  ],  declarations: [  AppComponent,  HeaderComponent,  Route1Component,  Route2Component,  Route3Component,  Route4Component,  Route4Child1Component,  Route4Child2Component,  ],  bootstrap: [AppComponent],  providers: []  })  class **AppModule** {  }  **platformBrowserDynamic**().**bootstrapModule**(AppModule); |

|  |
| --- |
| Router Guards |
| **4 types of Guards:**  **CanActivate**   * Checks to see if a user can visit a route.   **CanActivateChild**   * Checks to see if a user can visit a routes children.   **CanDeactivate**   * Checks to see if a user can exit a route.   **Resolve**   * Performs route data retrieval before route activation. * need lazy loading modules   **CanLoad**   * Checks to see if a user can route to a module that lazy loaded. * needs lazy loading modules   **Note:**   * A Route can implement zero or any number of Guards. * Guards return either **true** if the user can access a route or **false** if they can’t.   + They can also return an **Observable** or **Promise** , after they resolve (success or error blocks) it should return **true** or **false**   **Guard Function Parameters :**  To help determining whether or not a guard should accept or deny access the guard function can be passed certain arguments:   * **component**: **Component** this is the component itself. * **route**: **ActivatedRouteSnapshot**   + this is the future route that will be activated if the guard passes, we can use it’s params property to extract the route params. * **state**: **RouterStateSnapshot**   + this is the future RouterState if the guard passes, we can find the URL we are trying to navigate to from the url property.  |  | | --- | | import { NgModule, Component, Injectable } from '@angular/core';  import { BrowserModule } from '@angular/platform-browser';  import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';  import { Routes, RouterModule, Router, ActivatedRoute, CanActivate, CanActivateChild, CanDeactivate, ActivatedRouteSnapshot, RouterStateSnapshot } from "@angular/router";  import 'rxjs/add/operator/toPromise';  import { Observable } from 'rxjs/Observable';  import 'rxjs/add/observable/of';  *//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Services \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**  @**Injectable**()  class **UserService** {    *// Fake 'login' decision*  isUserLoggedIn: boolean = false;    **loginPromise**(): **Promise** {  return new **Promise**((resolve, reject) => {  **resolve**(this.isUserLoggedIn);  });  }      **loginObservable**(): **Observable** {  return Observable.**of**(this.isUserLoggedIn);  }  }  *//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Components \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**  @**Component**({  selector: 'route3',  template:  `  <h3> Route3Component </h3>  `  })  class **Route3Component** {    *// this fn gets called from*  **isOkToLeave**() {  return false;  }  }  @**Component**({  selector: 'route2child2',  template:  `  <h4> Route2Child2Component </h4>  `  })  class **Route2Child2Component** {  }  @**Component**({  selector: 'route2child1',  template:  `  <h4> Route2Child1Component </h4>    `  })  class **Route2Child1Component** {  }  @**Component**({  selector: 'route1',  template:  `  <h3> Route2Component </h3>  <hr/>  <ul *class*="nav nav-pills">  <li *class*="nav-item">  <a *class*="nav-link" *[routerLinkActive]*="['active']" *[routerLink]*="['./route2child1']">Route2child1</a>  </li>  <li *class*="nav-item">  <a *class*="nav-link" *[routerLinkActive]*="['active']" *[routerLink]*="['./route2child2']">Route2child2</a>  </li>  </ul>  <br/>  <router-outlet></router-outlet>  `  })  class **Route2Component** {  }    @**Component**({  selector: 'route1',  template:  `  <h3> Route1Component </h3>  `  })  class **Route1Component** {  }    @**Component**({  selector: 'app-header',  template: `  <nav *class*="navbar navbar-light bg-faded">  <ul *class*="nav navbar-nav">  <li *class*="nav-item" *[routerLinkActive]*="['active']">  <a *class*="nav-link" *[routerLink]*="['route1']">Route1</a>  </li>  <li *class*="nav-item" *[routerLinkActive]*="['active']">  <a *class*="nav-link" *[routerLink]*="['route2']">Route2</a>  </li>  <li *class*="nav-item" *[routerLinkActive]*="['active']">  <a *class*="nav-link" *[routerLink]*="['route3']">Route3</a>  </li>  </ul>  </nav>  <br/>  `  })  class **HeaderComponent** {  constructor(private router: **Router**) {  }  }  @**Component**({  selector: 'app',  template: `  <app-header></app-header>    <router-outlet></router-outlet>  `  })  class **AppComponent** {  }  *//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Guards \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**  *// We implemented 'CanDeactivate' interface with '<Generics>'*  *// so that, it can pass the appropriate component instance as part of canDeactivate fn*  class **UnsearchedTermGuard** implements **CanDeactivate**<**SearchComponent**> {  **canDeactivate**(component: **SearchComponent**,  route: **ActivatedRouteSnapshot**,  state: **RouterStateSnapshot**): boolean {      **console**.**log**("UnsearchedTermGuard");  **console**.**log**(state.url);  return component.**isOkToLeave**() || window.**confirm**("Are you sure?");  }  }  *// We implemented 'CanDeactivate' interface with '<Generics>'*  *// so that, it can pass the appropriate component instance as part of canDeactivate fn*  class **Route3ComponentFnCheckGuard** implements **CanDeactivate**<**Route3Component**> {    **canDeactivate**(component: **Route3Component**,  route: **ActivatedRouteSnapshot**,  state: **RouterStateSnapshot**): boolean {      **console**.**log**("Route3ComponentFnCheckGuard");  **console**.**log**(state.url);  return component.**isOkToLeave**() || window.**confirm**("Are you sure?");  }  }  class **AlwaysAuthChildrenGuard** implements **CanActivateChild** {  **canActivateChild**() {  **console**.**log**("AlwaysAuthChildrenGuard");  return true;  }  }  @**Injectable**()  class **OnlyLoggedInUsersGuard** implements **CanActivate** {  constructor(private userService: **UserService**) {  };  **canActivate**() {  **console**.**log**("OnlyLoggedInUsers");    *// 1. canActivate fn--can return boolean*  *// return true;*    *// 2. canActivate fn--can return Promise (that returns boolean)*  */\**  *return this.userService.loginPromise()*  *.then((isLoggedIn)=>{*  *return isLoggedIn;*  *}).catch((err)=>{*  *return false;*  *});*  *\*/*    *// 3. canActivate fn--can return Observable (that returns boolean)*  return this.userService.**loginObservable**();  }  }  class **AlwaysAuthGuard** implements **CanActivate** {  **canActivate**() {  **console**.**log**("AlwaysAuthGuard");  return true;  }  }  *//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Routes \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**  const routes: **Routes** = [  { path: '', redirectTo: 'route1', pathMatch: 'full' },    *// Whenever user navigates to the Route1Component route*  *// 'alwaysAuthGuard.canActivate()' will be called*  { path: 'route1', component: Route1Component, canActivate: [AlwaysAuthGuard]},    {  path: 'route2',  component: Route2Component,    *// Whenever user navigates to the Route2Component route*  *// 1. 'alwaysAuthGuard.canActivate()' will be called*  *// 2. 'onlyLoggedInUsersGuard.canActivate()' will be called*  *// --UseCase: CanActivate guard is to check user has permissions to view that page/route*  canActivate: [AlwaysAuthGuard, OnlyLoggedInUsersGuard],    *// Whenever user navigates to the Route2Component child routes (Route2Child1Component or Route2Child2Component )*  *// "alwaysAuthChildrenGuard.canActivate()" will be called*  canActivateChild: [AlwaysAuthChildrenGuard],  children: [  { path: '', redirectTo: 'route2child1', pathMatch: 'full' },  { path: 'route2child1', component: Route2Child1Component },  { path: 'route2child2', component: Route2Child2Component }  ]  },    *// Whenever user navigates to the Route3Component route*  *// 'alwaysAuthGuard.canDeactivate()' will be called*  { path: 'route3', component: Route3Component, canDeactivate: [Route3ComponentFnCheckGuard] },  ];  *//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Bootstrap \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**  @**NgModule**({  imports: [  BrowserModule,  RouterModule.**forRoot**(routes, { useHash: true })  ],  declarations: [  AppComponent,  HeaderComponent,  Route1Component,  Route2Component,  Route2Child1Component,  Route2Child2Component,  Route3Component  ],  bootstrap: [AppComponent],  providers: [  UserService,  OnlyLoggedInUsersGuard,  AlwaysAuthGuard,  AlwaysAuthChildrenGuard,  Route3ComponentFnCheckGuard  ]  })  class **AppModule** {  }  **platformBrowserDynamic**().**bootstrapModule**(AppModule); |   … |

|  |
| --- |
| Routing Strategies |
| **Client Side SPA’s we have two strategies:**   * HashLocationStrategy * PathLocationStrategy    HashLocationStrategy:  * To enable HashLocationStrategy in an Angular application * RouterModule.forRoot(routes, {useHash: true}) * To enable HashLocationStrategy in an Angular application * The # part of the url is called the **Hash Fragment**. * It’s normally used so that people can link to a particular section in a HTML page, specifically anchor tags   + <http://somedomain.com/page#routing-strategies>   + browser opens somedomain.com/page and then scroll down so that the <a name="routing-strategies"></a> top of the page * However it has another very important characteristic in that anything past the # in a URL never gets sent to the server.   + So if your URL was <https://codecraft.tv/contact/#/foo/moo/loo>   + then the browser makes a GET request to <https://codecraft.tv/contact/> only   + #/foo/moo/loo part of the URL is never sent.   + Anyway it is part of the URL so it can be bookmarked and sent to other people  PathLocationStrategy:  * This is the default strategy in Angular so we need to do nothing to enable it. * It takes advantage of a relatively new HTML5 API called pushstate (from the HTML5 history API). * When we navigate to different route (using pushstate)   + the URL changes to new route     - <http://localhost:4040/artist/1234/tracks>   + Also browser doesn’t send the request to the server     - /artist/1234/tracks * One big downside is,   + if we then reloaded the page or open the bookmarked page the browser would make a request to the server   + and the browser does not have the request the page from the server (Server will throw error)   + Otherwise Server Side should (ignore) additional routes   or  **Using base href**   * <base href='/my/app'/> * Or you can provide it to the DI framework it via the symbol APP\_BASE\_HREF. * The value of the base href gets prepended to every navigation request we make, * so if we ask to navigate to ['moo','foo'] with the above href the URL would change to /my/app/moo/foo   **Angular Universal:**   * What if we could build an app that rendered the page on the server side and returned that to the client, and from that point on let the client handle the routing? * What if we reloaded the page at **localhost:4040/artist/1234/tracks** and the server at **localhost:4040** rendered the page. * The server called the iTunes APIs and generated the HTML for the tracks page,   + it returned it to the browser and the browser just displayed it.   + Then if the user clicked on search the client application takes over and handles the routing on the client side. * That is something called Angular Universal   + it’s the ability to run Angular in both the browser and the server side.   + Big benefit of Angular Universal is that pages can be cached on the server side and applications will then load much faster.   … |

|  |
| --- |
| Custom Form Validators (Model Driven Form) |
| import {NgModule,Component,Pipe,OnInit} from '@angular/core';  import {ReactiveFormsModule,FormsModule,FormGroup,FormControl,Validators,FormBuilder} from '@angular/forms';  import {BrowserModule} from '@angular/platform-browser';  import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';  */\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Custom Validator \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/*  class **CodeCraftValidators** {  static **emailDomain**(requiredDomain) {    *// Note: this 'fn' returns another 'fn'*  return function (control: **FormControl**) {  let email = control.value;  if (email && email.**indexOf**("@") != -1) {  let [\_, domain] = email.**split**("@");  if (domain !== requiredDomain) {  return {  emailDomain: {  valid: false,  parsedDomain: domain  }  }  }  }  return null;  }  }  }  */\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Model Form \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/*  @**Component**({  selector: 'model-form',  template: `    <form *novalidate* *[formGroup]*="myform" *(ngSubmit)*="**onSubmit**()">  <div *class*="form-group" *[ngClass]*="{  'has-danger': email.invalid && (email.dirty || email.touched),  'has-success': email.valid && (email.dirty || email.touched)  }">  <input *type*="email" *class*="form-control"  *placeholder*="Email"  *formControlName*="email">    <div *class*="form-control-feedback" *\*ngIf*="email.errors && (email.dirty || email.touched)">  <p *\*ngIf*="email.errors.required">Email is required</p>  <p *\*ngIf*="email.errors.pattern">The email address must contain at least the @ character</p>  <p *\*ngIf*="email.errors.emailDomain">Email must be on the codecraft.tv domain</p>  </div>  </div>    <button *type*="submit" *[disabled]*="myform.invalid">Submit</button>    <pre>{{myform.value | json}}</pre>  </form>  `  })  class **ModelFormComponent** implements **OnInit** {    myform: **FormGroup**;  email: **FormControl**;    **ngOnInit**() {    *// 1. create FormControls :*  this.email = new **FormControl**('', [  Validators.required,  Validators.**pattern**("[^ @]\*@[^ @]\*"),    *// this 'emailDomain fn' returns a 'fn' --that returns 'null' or '{obj}'*  *// if 'null', it doesnt have any errors*  *// if '{obj}, it has errors*  CodeCraftValidators.**emailDomain**('codecraft.tv')  ]);      *// 2. create FormGroup :*  this.myform = new **FormGroup**({  email: this.email  });  }      **onSubmit**() {  if (this.myform.valid) {  **console**.**log**("Form Submitted!", this.myform.value);  this.myform.**reset**();  }  }  }  */\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* App \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/*  @**Component**({  selector: 'app',  template: `<model-form></model-form>`  })  class **AppComponent** {  }  @**NgModule**({  imports: [  BrowserModule,  FormsModule,  ReactiveFormsModule],  declarations: [  AppComponent,  ModelFormComponent  ],  bootstrap: [  AppComponent  ]  })  class **AppModule** {  }  **platformBrowserDynamic**().**bootstrapModule**(AppModule); |

|  |
| --- |
| Custom Form Validators (Template Driven Form) |
| import {NgModule,Component,OnInit,ViewChild,Directive,Inject,Input,} from '@angular/core';  import {NG\_VALIDATORS,FormsModule,FormGroup,FormControl,ValidatorFn,Validators} from '@angular/forms';  import {BrowserModule} from '@angular/platform-browser';  import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';  class **Signup** {  constructor(public email: string = '') {}  }  */\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Custom Validator \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/*  class **CodeCraftValidators** {  static **emailDomain**(requiredDomain) {    *// Note: this 'fn' returns another 'fn'*  return function (control: **FormControl**) {  **console**.**log**("here");  let email = control.value;  if (email && email.**indexOf**("@") != -1) {  let [\_, domain] = email.**split**("@");  if (domain !== requiredDomain) {  return {  emailDomain: {  parsedDomain: domain,  requiredDomain: requiredDomain  }  }  }  }  return null;  }  }  }  @**Directive**({    *// If input controls have 'emailDomain' attribute and 'ngModel' attribute*  *// then use this validator*  selector: '[emailDomain][ngModel]',    providers: [  {  *// list of all validators available in angular*  provide: NG\_VALIDATORS,    *// Our Custom Validtor Fn*  *// it internally calls emailDomainValidator.validate()*  useClass: EmailDomainValidator,    *// Multi providers return multiple dependencies as a list for a given token.*  *// so we are going to get list of validators returned as a result*  *// append 'emailDomainValidator' with list of dependencies that are associated with 'NG\_VALIDATORS token'*  multi: true  }  ]  })  class **EmailDomainValidator** {  private valFn = ValidatorFn;  *// we are getting 'requiredDomain' from 'NgModule' --> providers --> useValue*  constructor(@**Inject**('RequiredDomain') requiredDomain: string) {    *// emailDomain fn returns another 'fn' and we are setting that to 'valFn'*  this.valFn = CodeCraftValidators.**emailDomain**(requiredDomain)  }  **validate**(control: **FormControl**) {  return this.**valFn**(control);  }  }  */\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Model Form \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/*  @**Component**({  selector: 'template-form',  template: `  <form *novalidate* *(ngSubmit)*="**onSubmit**()" *#f*="ngForm">  <div *class*="form-group" *[ngClass]*="{  'has-danger': email.invalid && (email.dirty || email.touched),  'has-success': email.valid && (email.dirty || email.touched)  }">    <input *type*="email" *class*="form-control" *name*="email" *placeholder*="Email"  *[(ngModel)]*="model.email"  *required*  *pattern*="[^ @]\*@[^ @]\*"  *emailDomain*  *#email*="ngModel">  <div *class*="form-control-feedback" *\*ngIf*="email.errors && (email.dirty || email.touched)">  <p *\*ngIf*="email.errors.required">Email is required</p>  <p *\*ngIf*="email.errors.pattern">Email must contain at least the @ character</p>  *<!--<p \*ngIf="email.errors.emailDomain">Email must be on the codecraft.tv domain</p>-->*  <p *\*ngIf*="email.errors.emailDomain">Email must be on the {{ email.errors.emailDomain.requiredDomain }} domain</p>  </div>  </div>  <button *type*="submit" *class*="btn btn-primary" *[disabled]*="f.invalid">Submit  </button>  <pre>{{f.value | json}}</pre>  </form>  `  })  class **TemplateFormComponent** {  model: **Signup** = new **Signup**();  @**ViewChild**('f') form: any;  **onSubmit**() {  if (this.form.valid) {  **console**.**log**("Form Submitted!", this.form.value, this.model);  this.form.**reset**();  }  }  }  */\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* App \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/*  @**Component**({  selector: 'app',  template: `<template-form></template-form>`  })  class **AppComponent** {  }  @**NgModule**({  imports: [  BrowserModule,  FormsModule  ],  declarations: [  AppComponent,  TemplateFormComponent,  EmailDomainValidator  ],  bootstrap: [  AppComponent  ],  providers: [  {provide: 'RequiredDomain', useValue: 'codecraft.tv'}  ]  })  class **AppModule** {  }  **platformBrowserDynamic**().**bootstrapModule**(AppModule); |

|  |
| --- |
| Custom Form Validators (Template Driven Form) (Bindable) |
| import {NgModule,Component,OnInit,ViewChild,Directive,Inject,Input,} from '@angular/core';  import {NG\_VALIDATORS,FormsModule,FormGroup,FormControl,ValidatorFn,Validators} from '@angular/forms';  import {BrowserModule} from '@angular/platform-browser';  import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';  class **Signup** {  constructor(public email: string = '') {}  }  */\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Custom Validator \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/*  class **CodeCraftValidators** {  static **emailDomain**(requiredDomain) {    *// Note: this 'fn' returns another 'fn'*  return function (control: **FormControl**) {  **console**.**log**("here");  let email = control.value;  if (email && email.**indexOf**("@") != -1) {  let [\_, domain] = email.**split**("@");  if (domain !== requiredDomain) {  return {  emailDomain: {  parsedDomain: domain,  requiredDomain: requiredDomain  }  }  }  }  return null;  }  }  }  @**Directive**({    *// If input controls have 'emailDomain' attribute and 'ngModel' attribute*  *// then use this validator*  selector: '[emailDomain][ngModel]',    providers: [  {  *// list of all validators available in angular*  provide: NG\_VALIDATORS,    *// Our Custom Validtor Fn*  *// it internally calls emailDomainValidator.validate()*  useExisting: EmailDomainValidator,    *// Multi providers return multiple dependencies as a list for a given token.*  *// so we are going to get list of validators returned as a result*  *// append 'emailDomainValidator' with list of dependencies that are associated with 'NG\_VALIDATORS token'*  multi: true  }  ]  })  class **EmailDomainValidator** {  @**Input**('emailDomain') emailDomain: string;  private valFn = Validators.nullValidator;  **ngOnChanges**(): void {  if (this.emailDomain) {  *// emailDomain fn returns another 'fn' and we are setting that to 'valFn'*  this.valFn = CodeCraftValidators.**emailDomain**(this.emailDomain)  } else {  *// nullValidator fn returns another 'fn' that returns always 'null'*  this.valFn = Validators.nullValidator;  }  }  **validate**(control: **FormControl**) {  return this.**valFn**(control);  }  }  */\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Model Form \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/*  @**Component**({  selector: 'template-form',  template: `  <form *novalidate* *(ngSubmit)*="**onSubmit**()" *#f*="ngForm">  <div *class*="form-group" *[ngClass]*="{  'has-danger': email.invalid && (email.dirty || email.touched),  'has-success': email.valid && (email.dirty || email.touched)  }">    <input *type*="email" *class*="form-control" *name*="email" *placeholder*="Email"  *[(ngModel)]*="model.email"  *required*  *pattern*="[^ @]\*@[^ @]\*"  *[emailDomain]*="'codecraft.tv'"  *#email*="ngModel">  <div *class*="form-control-feedback" *\*ngIf*="email.errors && (email.dirty || email.touched)">  <p *\*ngIf*="email.errors.required">Email is required</p>  <p *\*ngIf*="email.errors.pattern">Email must contain at least the @ character</p>  *<!--<p \*ngIf="email.errors.emailDomain">Email must be on the codecraft.tv domain</p>-->*  <p *\*ngIf*="email.errors.emailDomain">Email must be on the {{ email.errors.emailDomain.requiredDomain }} domain</p>  </div>  </div>  <button *type*="submit" *class*="btn btn-primary" *[disabled]*="f.invalid">Submit  </button>  <pre>{{f.value | json}}</pre>  </form>  `  })  class **TemplateFormComponent** {  model: **Signup** = new **Signup**();  @**ViewChild**('f') form: any;  **onSubmit**() {  if (this.form.valid) {  **console**.**log**("Form Submitted!", this.form.value, this.model);  this.form.**reset**();  }  }  }  */\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* App \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/*  @**Component**({  selector: 'app',  template: `<template-form></template-form>`  })  class **AppComponent** {  }  @**NgModule**({  imports: [  BrowserModule,  FormsModule  ],  declarations: [  AppComponent,  TemplateFormComponent,  EmailDomainValidator  ],  bootstrap: [  AppComponent  ],  providers: [  {provide: 'RequiredDomain', useValue: 'codecraft.tv'}  ]  })  class **AppModule** {  }  **platformBrowserDynamic**().**bootstrapModule**(AppModule); |

|  |
| --- |
|  |
| … |

|  |
| --- |
|  |
| … |

|  |
| --- |
|  |
| … |

|  |
| --- |
|  |
| … |

|  |
| --- |
|  |
| … |

|  |
| --- |
|  |
| … |

|  |
| --- |
|  |
| … |

|  |
| --- |
|  |
| … |