Angular

Web framework
Notes by rajeeshvengalapurath@gmail.com

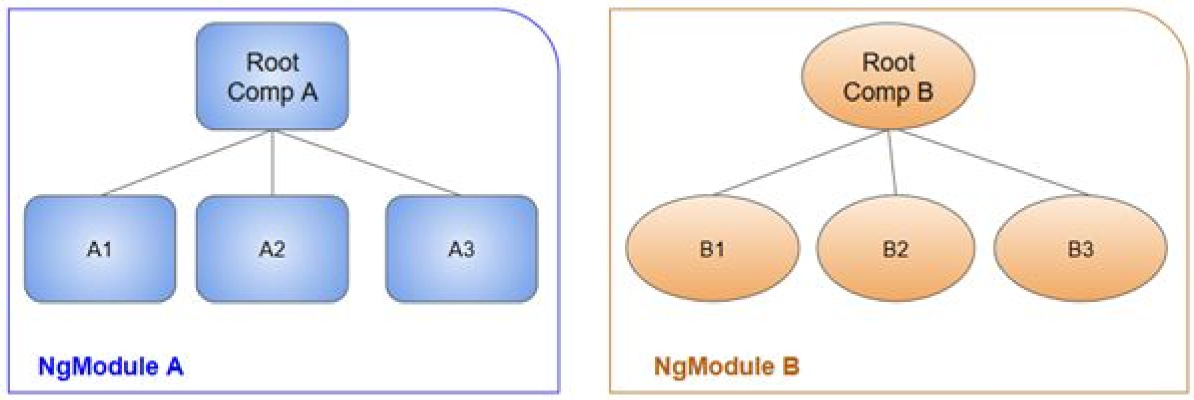
Introduction

- Angular is a platform and framework for building **single-page client applications** in HTML and TypeScript
- It **implements core** and optional functionality as a set of **TypeScript libraries** that you import into your apps.
- The basic building blocks are **NgModules**, which provide a compilation context for components.
- NgModules collect related code into functional sets
- An **Angular app** is defined by a **set of NgModules**
- An **app** always has **at least a root module** that **enables bootstrapping**, and typically has many more feature modules.
- Components define views, which are sets of **screen elements** that Angular can choose among and modify according to your program logic and data
- Components use **services**, which provide specific functionality not directly related to views. Service providers can be injected into components as dependencies, making your code modular, reusable, and efficient
- Both **components** and **services** are **simply classes**, with decorators that mark their type and provide metadata that tells Angular how to use them
- An app's components typically define many views, **arranged hierarchically**

Modules

- NgModules provide a **compilation context** for their components
  - **Compilation context** is a term for a **grouping of the TypeScript files** that will parse and analyze to determine what is valid and what is not valid.It's group of something(files, components) **which will be compiled**
- NgModules are containers for a cohesive block of code dedicated to an application domain, a workflow, or a closely related set of capabilities
- the **root module**, which is conventionally named **AppModule** and resides in a file named **app.module.ts**
- Angular NgModules differ from and complement JavaScript (ES2015) modules
- An NgModule can associate its components with related code, such as services, to form functional units
- Every Angular app has a **root module**, conventionally named **AppModule,** which provides the bootstrap mechanism that launches the application
- Like JavaScript modules, NgModules can **import functionality** from other NgModules
- Allow their own functionality to be **exported** and used by other NgModules

    - For example, to use the router service in your app, you import the Router NgModule.
- An NgModule is defined by a class decorated with **@NgModule()**
- The **@NgModule() decorator is a function** that takes **a single metadata object**, whose properties describe the module
    - **declarations**: The components, directives, and pipes that belong to this NgModule.
    - **exports**: The subset of declarations that should be visible and usable in the component templates of other NgModules.
    - **imports**: Other modules whose exported classes are needed by component templates declared in this NgModule.
    - **providers**: Creators of services that this NgModule contributes to the global collection of services; they become accessible in all parts of the app. (You can also specify providers at the component level, which is often preferred.)
    - **bootstrap**: The main application view, called the root component, which hosts all other app views. Only the root NgModule should set the bootstrap property.
- A root NgModule always has a root component that is created during bootstrap, but any NgModule can include any number of additional components
- The components that belong to an NgModule share a compilation context.
-



src/app/app.module.ts

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

Eager Loading

Feature modules under Eager Loading would be loaded before the application starts. This is the default module-loading strategy.

Lazy Loading

Suitable for big-size web application.
Feature modules under Lazy Loading would be loaded on demand after the application starts. It helps to start application faster.

app-routing.module.ts

```
const routes: Routes = [
  { path: 'home', component: HomeComponent},
  { path: '', redirectTo: '/home', pathMatch: 'full'},

   //lazy loading 'employee' is the common prefix. And this line should be before wildcard
  { path: 'employees', loadChildren: './employee/employee.module#EmployeeModule'},
  { path: '**', component: PageNotFoundComponent}
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

employee-routing.module.ts

```
const routes: Routes = [
  // localhost:port/employee
  { path: '', component: ListEmployeeComponent},

  // localhost:port/employee/create
  { path: 'create', component: CreateEmployeeComponent},

  // localhost:port/employee/edit/3
  { path: 'edit/:id', component: CreateEmployeeComponent}
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class EmployeeRoutingModule { }
```

Pre-Loading

Feature Modules under Pre-Loading would be loaded automatically after the application starts.

App-routing.module.ts

```
...
@NgModule({
  imports: [RouterModule.forRoot(routes, {preloadingStrategy: PreloadAllModules})],
  exports: [RouterModule]
})
```

Template syntax

- <script> element is forbidden, eliminating the risk of script injection attacks.
- The <html>, <body>, and <base> elements have no useful role

Interpolation {{...}}

Embedding expressions into marked up text. All expressions in double curly braces, converts the expression results to strings, and links them with neighboring literal strings

Template expressions

Moving data in one direction. A template expression produces a value and appears within the double curly braces. The interpolation braces in {{1 + 1}} surround the template expression 1 + 1

```
<h3>Current customer: {{ currentCustomer }}</h3>
<h3>Name: {{ getFullName() }}</h3>
<p>{{title}}</p>
<div><img src="{{itemImageUrl}}"></div>
<p>The sum of 1 + 1 is {{1 + 1}}.</p>
<p>The sum of 1 + 1 is not {{1 + 1 + getVal()}}.</p>
<p>{{pageHeader ? pageHeader : ‘No header’}}</p>
<img src=’{{imagePath}}’>
```

Expression Context

The expression context is typically the component instance. “**recommended**” and “**itemImageUrl2**” are properties of the AppComponent

```
<h4>{{recommended}}</h4>
<img [src]="itemImageUrl2">
```

Template Input Variable (properties of the template's context)

```
<li *ngFor="let customer of customers">{{customer.name}}</li>
```

Template ReferenceVariable (properties of the template's context)

```
<input #customerInput>{{customerInput.value}}
```

Template statements

(event)="statement".

```
<button (click)="deleteHero()">Delete hero</button>
```

Statement context

The statement context is typically the component instance. The deleteHero in (click)="deleteHero()" is a method of the data-bound component. Template statements cannot refer to anything in the global namespace. They can't refer to window or document. They can't call console.log or Math.max.

```
<button (click)="deleteHero()">Delete hero</button>
```

Binding syntax

- From the source-to-view
- From view-to-source
- Two-way sequence: view-to-source-to-view

From the Component to the DOM

Interpolation: {{ value }}

component → value of a property

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

Property binding: [property]=”value”

component → value of a property

```
<input type="email" [value]="user.email">
<div [style.background-color]="selectedColor">
<div [class.selected]="isSelected">
<button [disabled]="isUnchanged">Save</button>
<img [src]="itemImageUrl"> is same as <img bind-src="itemImageUrl">
```

Property binding vs. interpolation

Interpolation is a convenient alternative to property binding in many cases.When rendering data values as strings, there is no technical reason to prefer one form to the other, though readability tends to favor interpolation. However, when setting an element property to a non-string data value, you must use property binding.

```
<p><img src="{{itemImageUrl}}"> is the <i>interpolated</i> image.</p>
Same as
<p><img [src]="itemImageUrl"> is the <i>property bound</i> image.</p>

<p><span>"{{interpolationTitle}}" is the <i>interpolated</i> title.</span></p>
Same as
<p>"<span [innerHTML]="propertyTitle"></span>" is the <i>property bound</i>
title.</p>
```

Attribute binding

Note: Usually, setting an element property with a **property binding is preferable** to setting the attribute with a string.

```
<button [attr.aria-label]="actionName">{{actionName}} with Aria</button>
```

Class binding

The NgClass directive can be used as an alternative to direct [class] bindings. However, using the below class binding syntax without NgClass is preferred because due to improvements in class binding in Angular, NgClass no longer provides significant value, and might eventually be removed in the future.

Single Class Binding

```
[class.foo]="hasFoo” (hasFoo can be true or false to add/remove class)
```

Multiple Class Binding

```
[class]="classExpr"
```

classExpr can be

- `“my-class-1 my-class-2 my-class-3”)`
- `{foo: true, bar: false}`
- `['foo', 'bar']`

Style binding

The NgStyle directive can be used as an alternative to direct [style] bindings. However, using the above style binding syntax without NgStyle is preferred because due to improvements in style binding in Angular, NgStyle no longer provides significant value, and might eventually be removed in the future.

Single Style Binding

```
[style.width]="width" (width can be “100px” of type string)
```

Single Style Binding with Units

```
[style.width.px]="width" (width can be 100 of type number)
```

Multiple Style Binding

```
[style]="styleExpr"
```

styleExpr can be

- `"width: 100px; height: 100px"`
- `{width: '100px', height: '100px'}`
- `['width', '100px']`

From DOM to the Component

Event binding

```
<button (click)="cookBacon()"></button>
<button (click)="onSave($event)">Save</button>
<button on-click="onSave($event)">on-click Save</button> (Canonical Form)

<h4>myClick is an event on the custom ClickDirective:</h4>
<button (myClick)="clickMessage=$event" clickable>click with myClick</button>
{{clickMessage}}
```

$event

$event. Can be an event object, string, or number. If the target event is a native DOM element event, then $event is a DOM event object, with properties such as target and target.value.

```
<input [value]="currentItem.name" (input)="currentItem.name=$event.target.value" >
-> without NgModel
```

Custom events with EventEmitter

src/app/item-detail/item-detail.component.html (template)

```
<img src="{{itemImageUrl}}" [style.display]="displayNone">
<span [style.text-decoration]="lineThrough">{{ item.name }}
</span>
<button (click)="delete()">Delete</button>
```

src/app/item-detail/item-detail.component.ts (deleteRequest)

```
// This component makes a request but it can't actually delete a hero.
@Output() deleteRequest = new EventEmitter<Item>();

delete() {
  this.deleteRequest.emit(this.item);
  this.displayNone = this.displayNone ? '' : 'none';
  this.lineThrough = this.lineThrough ? '' : 'line-through';
}
```

src/app/app.component.html (event-binding-to-component)

```
<app-item-detail (deleteRequest)="deleteItem($event)"
[item]="currentItem"></app-item-detail>
```

Two-way

Two-way binding does two things:

- Sets a specific element property.
- Listens for an element change event.

Two-way data binding: [(ngModel)]=”value” (banana in a box syntax)

Import the FormsModule and add it to the NgModule's imports list

```
<input [(ngModel)]="currentItem.name">
Same, without ngModel
<input [value]="currentItem.name" (input)="currentItem.name=$event.target.value"
id="without">
```

To streamline the syntax, the ngModel directive hides the details behind its own ngModel input and ngModelChange output properties:

```
<input [ngModel]="currentItem.name" (ngModelChange)="currentItem.name=$event"
id="example-change">
```

HTML attribute vs. DOM property

HTML Tag → <body id="page"> → id is HTML attribute
DOM object → body.id="page" → id is DOM property
Template binding works with properties and events, not attributes of the target object. The HTML attribute value specifies the initial value; the DOM value property is the current value.

Add numbers of two input box

(input) event fires immediately after keypress

```
<!--method 1→
<div>
    <input type="text" [(ngModel)]="input1" (input)="showSum()">
    <input type="text" [(ngModel)]="input2" (input)="showSum()">
    <input type="text" [(ngModel)]="input3">
</div>

showSum(){
    this.input3=+this.input1 + +this.input2
}

<!--method 2-->
<div>
    <input type="text" [(ngModel)]="a" (input)="c=+a + +b">
    <input type="text" [(ngModel)]="b" (input)="c=+a + +b">
    <input type="text" [(ngModel)]="c">
</div>

<!--method 3-->
<div>
    <input type="text" [value]="x" (input)="x=$event.target.value; z=+x + +y">
    <input type="text" [value]="y" (input)="y=$event.target.value; z=+x + +y">
    <input type="text" [value]="z">
</div>
```

Tips

```
<div>
    <input type="text" [attr.type]="'button'"> <!--attribute not property-->
    <input type="text" [style.color]="'red'" [style.backgroundColor]="xyz">
    <input type="button" (click)="xyz=(xyz==='green'?'yellow':'green')">
    <input type="button" value="Show" (click)="showMe=(showMe===true?false:true)">
</div>
```

Accordion example

`youtube.com/watch?v=2SJ9Ch8jX3A&list=PL6n9fhu94yhXwcl3a6rIfAI7QmGYIkfK5&index=62`

Here `[hidden]="isHidden"` is used instead of `*ngIf="panelExpanded"` because *ngIf removes the div. But if say, you want to show certain section only to the administrators and not to normal users then *ngIf should be used.

Simple

```
<div class="panel-heading pointerCursor" (click)="isHidden=!*ngIf="panelExpanded"">
        ...
</div>

<div class="panel-body" [hidden]="isHidden">
...
</div>

<div class="panel-footer" [hidden]="isHidden">
...
</div>

.pointerCursor {
    cursor: pointer;
}
```

<ng-content>

accordion.component.ts

```
import ..

@Component ..
export class AccordionComponent implements OnInit {

  @Input() hasJustViewed: boolean
  @Input() title: string
  isHidden = false
  ..
}
```

accordion.component.html

```
<div class="panel panel-primary" [class.panel-success]="hasJustViewed">
    <div class="panel-heading pointerCursor" (click)="isHidden=!isHidden">
        <h3 class="panel-title">{{title | uppercase}}</h3>
    </div>
    <div class="panel-body" [hidden]="isHidden">
        <ng-content select=".myPanelBody"></ng-content> <!--place holder-->
    </div>
    <div class="panel-footer" [hidden]="isHidden">
        <ng-content select=".myPanelFooter"></ng-content> <!--place holder-->
    </div>
</div>
```

display-employee.component.html

```
<app-accordion [title]="employee.name"
[hasJustViewed]="selectedEmployeeId===employee.id">
    <!--myPanelBody is the selector at ng-content-->
    <div class="col-xs-10 myPanelBody">
       ...
    </div>
    <div class="myPanelFooter">
       ...
    </div>
</app-accordion>
```

Built-in structural directives

Structural directives shape or reshape the DOM's structure, typically by adding, removing, and manipulating the elements to which they are attached. Any directive with an **asterisk**, *, is a **structural directive**.

NgIf

If you are hiding large component trees, consider NgIf as a more efficient alternative to showing/hiding.

Advantages

- When NgIf is false, Angular removes the element and its descendants from the DOM (Not hiding). It destroys their components, freeing up resources, which results in a better user experience.
- You can use it to guard against null. Angular will throw an error if a nested expression tries to access a property of null.

```
<app-item-detail *ngIf="isActive" [item]="item"></app-item-detail>
```

*ngFor (repeater directive)

```
<div *ngFor="let product of products">
<app-item-detail *ngFor="let item of items" [item]="item"></app-item-detail>
```

*ngFor with index

```
<div *ngFor="let item of items; let i=index">{{i + 1}} - {{item.name}}</div>
```

*ngFor with trackBy

src/app/app.component.ts

```
trackByItems(index: number, item: Item): number { return item.id; }
```

src/app/app.component.html

```
<div *ngFor="let item of items; trackBy: trackByItems">
  ({{item.id}}) {{item.name}}
</div>
```

NgSwitch (attribute directive, controller directive)

You'll get an error if you try to set *ngSwitch because NgSwitch is an attribute directive, not a structural directive. Rather than touching the DOM directly, it changes the behavior of its companion directives. The NgSwitchCase and NgSwitchDefault directives are structural directives because they add or remove elements from the DOM

```
<div [ngSwitch]="currentItem.feature">
  <app-stout-item *ngSwitchCase="'stout'" [item]="currentItem"></app-stout-item>
  <app-device-item *ngSwitchCase="'slim'" [item]="currentItem"></app-device-item>
  <app-lost-item *ngSwitchCase="'vintag'" [item]="currentItem"></app-lost-item>
  <app-best-item *ngSwitchCase="'bright'" [item]="currentItem"></app-best-item>
  ...
  <app-unknown-item *ngSwitchDefault [item]="currentItem"></app-unknown-item>
</div>
```

Work as well with native elements and web components too

```
<div *ngSwitchCase="'bright'"> Are you as bright as {{currentItem.name}}?</div>
```

Template reference variables (#var)

Reference to a DOM element within a template, directive (which contains a component), an element, TemplateRef, or a web component. In most cases, Angular sets the reference variable's value to the element on which it is declared. The scope of a reference variable is the entire template

```
<input #phone placeholder="phone number" />
...
<button (click)="callPhone(phone.value)">Call</button>
```

ref- prefix alternative to

```
<input ref-fax placeholder="fax number"/>
```

ngForm

The reference value of itemForm, without the ngForm attribute value, would be the HTMLFormElement. With NgForm, itemForm is a reference to the NgForm directive with the ability to track the value and validity of every control in the form. The native <form> element doesn't have a form property, but the NgForm directive does, which allows disabling the submit button if the itemForm.form.valid is invalid and passing the entire form control tree to the parent component's onSubmit() method.

```
<form #itemForm="ngForm" (ngSubmit)="onSubmit(itemForm)">
  <label for="name"
    >Name <input class="form-control" name="name" ngModel required />
  </label>
  <button type="submit">Submit</button>
</form>

<div [hidden]="!itemForm.form.valid">
  <p>{{ submitMessage }}</p>
</div>
```

Input() and @Output() properties

@Input() and @Output() allow Angular to share data between the parent context and child directives or components. An @Input() property is writable while an @Output() property is observable.

@Input()

In the child

(src/app/item-detail/item-detail.component.ts)

```
import { Component, Input } from '@angular/core';
export class ItemDetailComponent {
  @Input() item: string;
}
```

In the parent

src/app/app.component.html

```
<app-item-detail [item]="currentItem"></app-item-detail>
```

src/app/app.component.ts

```
export class AppComponent {
  currentItem = 'Television';
}
```

@Output()

Just like with @Input(), you can use @Output() on a property of the child component but its type should be EventEmitter.

In the child

src/app/item-output/item-output.component.ts

```
import { Output, EventEmitter } from '@angular/core';

export class ItemOutputComponent {

  @Output() newItemEvent = new EventEmitter<string>();

  addNewItem(value: string) {
    this.newItemEvent.emit(value);
  }
}
```

src/app/item-output/item-output.component.html

```
<label>Add an item: <input #newItem></label>
<button (click)="addNewItem(newItem.value)">Add to parent's list</button>
```

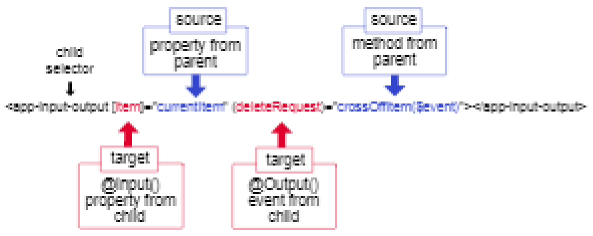
In the parent

src/app/app.component.ts

```
export class AppComponent {
  items = ['item1', 'item2', 'item3', 'item4'];
  addItem(newItem: string) {
    this.items.push(newItem);
  }
}
```

src/app/app.component.html

```
<app-item-output (newItemEvent)="addItem($event)"></app-item-output>
<ul>
  <li *ngFor="let item of items">{{item}}</li>
</ul>
```


source
property from parent
source
method from parent
child selector
<app-input-output [item]="currentItem" (deleteRequest)="crossOffItem($event)"></app-input-output>
target
@Input() property from child
target
@Output() event from child

Template expression operators

The pipe operator (|)

For the transformation of the result of an expression

```
<p>Title through uppercase pipe: {{title | uppercase}}</p>

<!-- convert title to uppercase, then to lowercase -->
<p>Title through a pipe chain: {{title | uppercase | lowercase}}</p>

<!-- pipe with configuration argument => "February 25, 1980" -->
<p>Manufacture date with date format pipe: {{item.manufactureDate |
date:'longDate'}}</p>

<p>Item json pipe: {{item | json}}</p>
```

The safe navigation operator (?) and null property paths

Guards against null and undefined

```
<p>The item name is: {{item?.name}}</p>
a?.b?.c?.d.
```

The non-null assertion operator (!)

Unlike the safe navigation operator, the non-null assertion operator does not guard against null or undefined. Rather, it tells the TypeScript type checker to suspend strict null checks for a specific property expression.

```
<!--No color, no error -->
<p *ngIf="item">The item's color is: {{item!.color}}</p>
```

Built-in template functions

The $any() ‘type cast’ function

To silence a type error during AOT compilation, when it is not possible or difficult to fully specify the type the $any() cast function is used to cast the expression to the any type as in the following example:

```
<p>The item's undeclared best by date is: {{$any(item).bestByDate}}</p>
```

Also works with this to allow access to undeclared members of the component

```
<p>The item's undeclared best by date is: {{$any(this).bestByDate}}</p>
```

SVG in templates

When you use an SVG as the template and not as an image, you are able to use directives and bindings just like with HTML templates. This means that you will be able to dynamically generate interactive graphics.

src/app/svg.component.ts

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

@Component({
  selector: 'app-svg',
  templateUrl: './svg.component.svg',
  styleUrls: ['./svg.component.css']
})
export class SvgComponent {
  fillColor = 'rgb(255, 0, 0)';

  changeColor() {
    const r = Math.floor(Math.random() * 256);
    const g = Math.floor(Math.random() * 256);
    const b = Math.floor(Math.random() * 256);
    this.fillColor = `rgb(${r}, ${g}, ${b})`;
  }
}
```

src/app/svg.component.svg

```
<svg>
  <g>
    <rect x="0" y="0" width="100" height="100" [attr.fill]="fillColor"
(click)="changeColor()" />
    <text x="120" y="50">click the rectangle to change the fill color</text>
  </g>
</svg>
```

Components

- Components define **areas of responsibility** in the user interface, or UI
- It lets you **reuse sets of UI functionality**.
- An Angular application comprises a **tree of components**, in which each Angular component has a specific purpose and responsibility.
- Ideally, a component's job is to enable the user experience and nothing more
- A component should present properties and methods for data binding, in order to mediate between the view (rendered by the template) and the application logic (which often includes some notion of a model).
- You must declare every component in exactly one NgModule class
- If you use a component without declaring it, Angular returns an error message.

A component consists of three things:
- A **component class** that **handles data and functionality**.
- An **HTML template** that **determines the UI**.
- **Component-specific styles** that define the **look and feel**.

Create a new component

- Right click on the app folder and use the Angular Generator to generate a new component named “product-alerts”
- The generator creates starter files for all three parts of the component:
    - product-alerts.component.ts
    - product-alerts.component.html
    - Product-alerts.component.css
- **@Component() decorator** indicates that the following class is a component. It **provides metadata about the component**, including its **selector, templates, and styles.**
-
-

Sample Component.ts

src/app/product-alerts/product-alerts.component.ts

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

@Component({
  selector: 'app-product-alerts',
  templateUrl: './product-alerts.component.html',
  styleUrls: ['./product-alerts.component.css']
})
export class ProductAlertsComponent implements OnInit {
  constructor() { }
  ngOnInit() {   }
}
```

Passing property value to child component

Parent Component

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

@Component({
  selector: 'app-product-list',
  templateUrl: './product-list.component.html',
  styleUrls: ['./product-list.component.css']
})
export class ProductListComponent {
  products = products;
}
```

View

```
<div *ngFor="let product of products">
  <h3>{{ product.name }}</h3>
  <app-product-alerts [product]="product"></app-product-alerts>
</div>
```

Child Component

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

@Component({
  selector: 'app-product-alerts',
  templateUrl: './product-alerts.component.html',
  styleUrls: ['./product-alerts.component.css']
})
export class ProductAlertsComponent implements OnInit {

  @Input() product

  constructor() {}
  ngOnInit() {}
}
```

View

```
<p *ngIf="product.price>700">price is over 700</p>
```

Services

- A service is an **instance of a class** that can be made available to any part of your application using Angular's **dependency injection** system.

- Services are the place where you **share data** between parts of your application
- For the online store, the **cart service** is where you **store your cart data** and **methods**.
- A component can delegate certain tasks to services, such as **fetching data from the server, validating user input**, or **logging directly to the console**
- By defining such processing tasks in an **injectable service class**, you make those tasks available to any component
- Angular doesn't enforce these principles. Angular does help you follow these principles by making it easy to factor your application logic into services and make those services available to components through dependency injection
- Services can depend on other services

Why write a service

- to post-process the data
- add error handling
- some retry logic to cope with intermittent connectivity
- The component quickly becomes cluttered with data access minutia
- The component becomes harder to understand
- The component becomes harder to test
- The data access logic can't be re-used or standardized

It's a best practice to separate presentation of data from data access by encapsulating data access in a separate service

Services can depend on other services

```
export class HeroService {
  private heroes: Hero[] = [];

  constructor(
    private backend: BackendService,
    private logger: Logger) { }

  getHeroes() {
    this.backend.getAll(Hero).then( (heroes: Hero[]) => {
      this.logger.log(`Fetched ${heroes.length} heroes.`);
      this.heroes.push(...heroes); // fill cache
    });
    return this.heroes;
  }
}
```

Singleton services

There are two ways to make a service a singleton in Angular:

- Declare root for the value of the @Injectable() providedIn property

    - Beginning with Angular 6.0, the preferred way to create a singleton service is to set providedIn to root on the service's @Injectable() decorator. This tells Angular to provide the service in the application root.
- Include the service in the AppModule or in a module that is only imported by the AppModule

Sample

```
File: src/app/cart.service.ts

import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class CartService {

  constructor() {}

}
```

Dependency

A dependency doesn't have to be a service—it could be a function, for example, or a value.

Observable vs Promise

Promise

```
const promise = new Promise((callback, reject) => {
      console.log('1. Inside Promise');
      callback();
});

console.log('2. Before calling then on Promise');

promise.then(() => { console.log('3. Callback'); });
```

Observable

```
const greetingLady$ = new Observable(observer => {
  console.log('2. Inside Observable (proof of being lazy)');
  observer.next('3. Hello! I am glad to get to know you.');
  observer.complete();
});

console.log('1. Before calling subscribe on Observable');

greetingLady$.subscribe({
  next: console.log,
  complete: () => console.log('4. End of conversation with preety lady')
});
```

Observables

- Provide support for passing messages between publishers and subscribers
- Observables are declarative
  - you define a function for publishing values, but it is not executed until a consumer subscribes to it
  - The subscribed consumer then receives notifications until the function completes, or until they unsubscribe.
- An observable can deliver multiple values of any type—literals, messages, or events etc
- The API for receiving values is the same whether the values are delivered synchronously or asynchronously
- application code only needs to worry about subscribing to consume values, and when done, unsubscribing

Basic usage and terms

- As a publisher, you create an Observable instance that defines a subscriber function
- This is the function that is executed when a consumer calls the subscribe() method
- The subscriber function defines how to obtain or generate values or messages to be published.
-

HttpClient

- The **HttpClient** in **@angular/common/http**
  - Offers a simplified client **HTTP API** for Angular applications
  - Rests on the **XMLHttpRequest interface** exposed by browsers
  - **Testability** features
  - **Typed request** and **response** objects
  - Request and response interception
  - **Observable apis**
  - Streamlined **error handling**

Typed response

Use an **interface** rather than a class; a response **cannot** be automatically **converted** to an **instance of a class**

```
interface MyInterface{
  name: string
  price: number
}

myf(){
       this.http.get<MyInterface[]>("/assets/products.json")
              .subscribe((data: MyInterface[]) =>
              {
                     data.forEach((item,index) => //item is of type MyInterface
                     {
                            console.log(item.name + " " + item.price)
                     })
              }, (error)=>
              {
                     console.log(error.message);
              })
}
```

Retrying

```
getConfig() {
  return this.http.get<Config>(this.configUrl)
    .pipe(
      retry(3), // retry a failed request up to 3 times
      catchError(this.handleError) // then handle the error
    );
}
```

Headers

```
import { HttpHeaders } from '@angular/common/http';

const httpOptions = {
  headers: new HttpHeaders({
    'Content-Type':  'application/json',
    'Authorization': 'my-auth-token'
  })
}
```

Post

```
addHero (hero: Hero): Observable<Hero> {

  //set header if needed
  httpOptions.headers = httpOptions.headers
              .set('Authorization', 'my-new-auth-token');

  //post
  return this.http.post<Hero>(this.heroesUrl, hero, httpOptions)
    .pipe(
      catchError(this.handleError('addHero', hero))
    );
}
```

Delete

```
deleteHero (id: number): Observable<{}> {
  const url = `${this.heroesUrl}/${id}`; // DELETE api/heroes/42
  return this.http.delete(url, httpOptions)
    .pipe(
      catchError(this.handleError('deleteHero'))
    );
}

this.heroesService
  .deleteHero(hero.id)
  .subscribe();
```

Put

```
updateHero (hero: Hero): Observable<Hero> {
  return this.http.put<Hero>(this.heroesUrl, hero, httpOptions)
    .pipe(
      catchError(this.handleError('updateHero', hero))
    );
}
```

Always subscribe

An HttpClient method does not begin its HTTP request until you call subscribe() on the observable returned by that method. This is true for all HttpClient methods. In fact, each subscribe() initiates a separate, independent execution of the observable. Subscribing twice results in two HTTP requests

```
const req = http.get<Heroes>('/api/heroes');
// 0 requests made - .subscribe() not called.
req.subscribe();
// 1 request made.
req.subscribe();
// 2 requests made.
```

Sample

app.module.ts

```
…
import { HttpClientModule } from '@angular/common/http';
@NgModule({
  declarations: [
    ...
  ],
  imports: [
    BrowserModule,
    HttpClientModule, //After BrowserModule
    ...
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

cart.service.ts

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http'

@Injectable({
  providedIn: 'root'
})
export class CartService {
  items = []
  constructor(private http: HttpClient) { }
  getItems(){
    return this.http.get('/assets/products.json')
  }
}
```

rajeesh.component.ts

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { CartService } from '../cart.service';

@Component({
  selector: 'app-rajeesh',
  templateUrl: './rajeesh.component.html',
  styleUrls: ['./rajeesh.component.css']
})
export class RajeeshComponent implements OnInit {
  items
  constructor(private route: ActivatedRoute, private cartService: CartService) { }

  ngOnInit() {
    this.items =this.cartService.getItems();
  }
}
```

rajeesh.component.html

```
<div *ngFor='let item of items | async'>
    <h3> {{ item.name + ', Price ' + item.price}} </h3>
</div>
```

/assets/products.json

```
[
    {"name": "Chair", "price": 2000},
    {"name": "Table", "price": 10000},
    {"name": "cot", "price": 20000}
]
```

Pipes

```
birthday is {{ birthday | date }}
birthday is {{ birthday | date:"MM/dd/yy" }} //Parameterizing a pipe
birthday is {{ birthday | date | uppercase}} //Chaining pipes
```

Custom pipes

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({name: 'exponentialStrength'})
export class ExponentialStrengthPipe implements PipeTransform {
  transform(value: number, exponent?: number): number {
    return Math.pow(value, isNaN(exponent) ? 1 : exponent);
  }
}
```

Pure and impure pipes

```
@Pipe({
  name: 'flyingHeroesImpure',
  pure: false
})
```

Pure pipes

Angular executes a pure pipe only when it detects a pure change to the input value. A pure change is either a change to a primitive input value (String, Number, Boolean, Symbol) or a changed object reference (Date, Array, Function, Object).

Angular ignores changes within (composite) objects. It won't call a pure pipe if you change an input month, add to an input array, or update an input object property. This may seem restrictive but it's also fast.

Impure pipes

Angular executes an impure pipe during every component change detection cycle. An impure pipe is called often, as often as every keystroke or mouse-move.

With that concern in mind, implement an impure pipe with great care. An expensive, long-running pipe could destroy the user experience.

RxJS library

- **Reactive programming** is an asynchronous programming paradigm concerned with data streams and the propagation of change
- RxJS (Reactive Extensions for JavaScript) is a library for reactive programming using observables that makes it easier to compose asynchronous or callback-based code
- Features
    - Converting existing code for async operations into observables
    - Iterating through the values in a stream
    - Mapping values to different types
    - Filtering streams
    - Composing multiple streams

Angular Reactive Form

There are two parts to an Angular Reactive form

- the **objects that live in the component to store** and manage the form
- and the **visualization of the form that lives in the template.**

app.modules.ts

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

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

my-form.component.ts

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

@Component({
...
})
```

```
export class MyFormComponent implements OnInit {

  myFormGroup: FormGroup

  constructor(private formBuilder: FormBuilder) {
    this.myFormGroup = this.formBuilder.group({
      name: '',
      address: ''
    })
  }

  mySubmitFn(myData){
    console.warn('data is submitted', myData)
  }
  ngOnInit() {
  }

}
```

my-form.component.html

```
<p>my-form works!</p>
<form [formGroup]="myFormGroup" (ngSubmit)="mySubmitFn(myFormGroup.value)">
    <input type="text" formControlName="name">
    <input type="text" formControlName="address">
    <button type="submit">Purchase</button>
</form>
```

User input

To bind to a DOM event, surround the DOM event name in parentheses and assign a quoted template statement to it

```
<button (click)="onClickMe()">Click me!</button>
```

Form Validation

```
<input
  required minlength="4"
  appForbiddenName="bob"
  [(ngModel)]="hero"
  #myNgModelVariable="ngModel"
>

<div
       *ngIf="myNgModelVariable.invalid
       && (myNgModelVariable.dirty || myNgModelVariable.touched)"
>

  <div *ngIf="myNgModelVariable.errors.required">
    Name is required.
  </div>
  <div *ngIf="myNgModelVariable.errors.minlength">
    Name must be at least 4 characters long.
  </div>
  <div *ngIf="myNgModelVariable.errors.forbiddenName">
    Name cannot be Bob.
  </div>

</div>
```

Routing

- Routing usually mean matching components (the resource people want) to a URL (the way of telling the system what they want)
- The Angular Router enables navigation from one view to the next as users perform application tasks

<base href>

Most routing applications should add a <base> element to the index.html as the first child in the <head> tag to tell the router how to compose navigation URLs

```
<!doctype html>
<html lang="en">
<head>
  ...
  <base href="/">
  ...
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

Router imports

src/app/app.module.ts (import)

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

Configuration

example creates five route definitions

```
const appRoutes: Routes = [
  { path: 'crisis-center', component: CrisisListComponent },
  { path: 'hero/:id',      component: HeroDetailComponent },
  {
    path: 'heroes',
    component: HeroListComponent,
    data: { title: 'Heroes List' }
  },
  { path: '',
    redirectTo: '/heroes',
    pathMatch: 'full'
  },
  { path: '**', component: PageNotFoundComponent }
];

@NgModule({
  imports: [
    RouterModule.forRoot(
      appRoutes,
      { enableTracing: true } // <-- debugging purposes only
    )
    ...
  ],
  ...
})
export class AppModule { }
```

Router outlet

The RouterOutlet is a directive from the router library that is used like a component. It acts as a placeholder that marks the spot in the template where the router should display the components for that outlet

```
<router-outlet></router-outlet>
<!-- Routed components go here -->
```

Router links

src/app/app.component.html

```
<style>
  .active { color: green; background-color: red;}
</style>

<h1>Angular Router</h1>
<nav>
  <a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
  <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
</nav>
<router-outlet></router-outlet>
```

App.module.ts (Registering Route)

```
@NgModule({
  imports: [
    ...
    RouterModule.forRoot([
      { path: '', component: ProductListComponent },
      { path: 'products/:productId', component: ProductDetailsComponent},
    ])
  ],
```

product-list.component.html

```
<div *ngFor="let product of products; index as productId">
    <a [routerLink]="['/products',productId]">{{ product.name }}</a>
</div>
```

product-details.component.ts

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { products } from '../products';

@Component({
  selector: 'app-product-details',
  templateUrl: './product-details.component.html',
  styleUrls: ['./product-details.component.css']
})
export class ProductDetailsComponent implements OnInit {
  product
  constructor(private route: ActivatedRoute) { }

  ngOnInit() {
    this.route.paramMap.subscribe(params=>{
      this.product = products[+params.get('productId')];
    });
  }
}
```

Allow access outside localhost

```
ng serve --host 0.0.0.0
```

Summary

- **Router** Displays the application component for the active URL. Manages navigation from one component to the next.
- **RouterModule** A separate NgModule that provides the necessary service providers and directives for navigating through application views.
- **Routes** Defines an array of Routes, each mapping a URL path to a component.
- **Route** Defines how the router should navigate to a component based on a URL pattern. Most routes consist of a path and a component type.
- **RouterOutlet** The directive (<router-outlet>) that marks where the router displays a view.

- **RouterLink** The directive for binding a clickable HTML element to a route. Clicking an element with a routerLink directive that is bound to a string or a link parameters array triggers a navigation.
- **RouterLinkActive** The directive for adding/removing classes from an HTML element when an associated routerLink contained on or inside the element becomes active/inactive.
- **ActivatedRoute** A service that is provided to each route component that contains route specific information such as route parameters, static data, resolve data, global query params, and the global fragment.
- **RouterState** The current state of the router including a tree of the currently activated routes together with convenience methods for traversing the route tree.
- **Link parameters array** An array that the router interprets as a routing instruction. You can bind that array to a RouterLink or pass the array as an argument to the Router.navigate method.
- **Routing component** An Angular component with a RouterOutlet that displays views based on router navigations.

Angular CRUD

Create new Angular project

To eliminate test file generated for components include **--skip-tests true**

```
ng new AngularCrud --skip-tests true
```

Angular Version

Check **package.json**, under dependencies

Install Bootstrap (say version 3) with jquery

Will be installed on D:\AngularCrud\node_modules\bootstrap. Bootstrap may need to use jquery, hence included. Command should be executed from within application directory. Installed information can be found in package.json file,

```
D:\AngularCrud>npm install bootstrap@3 jquery --save
```

Add path "../node_modules/bootstrap/dist/css/bootstrap-theme.min.css" to angular.json file

angular.json

```
Following bold lines are manually added to the file
..
Line No:48
"styles": [
       "src/styles.css",
        "node_modules/bootstrap/dist/css/bootstrap.min.css"
],
"scripts": [
        "node_modules/jquery/dist/jquery.min.js",
        "node_modules/bootstrap/dist/js/bootstrap.min.js"
]
```

Routing module (app-routing.module.ts)

If this file is not created then run the following command to create it

```
ng g m app-routing --flat=true --module=app
```

Create Component

```
ng g c employees/listEmployees --spec false --flat true
```

(g for generate, c for component, flat and employees for employees folder)

Production

```
ng build -prod --base-href /emp/
```

Result: http://localhost:4200/emp/list. Also will generate **dist** folder

Angular Forms

Two ways to create forms in Angular

1. Template Driven Forms
    a. To create simple forms
2. Model Driven Forms (Reactive Forms)
    a. To create complex forms
    b. Create controls dynamically

Error: If ngModel is used within a form tag, either the name attribute must be set or the form control must be defined as 'standalone' in ngModelOptions

Validation

- By default Angular4 and above disables browser built in validation
- use ngNativeValidation to enable browser default validation
- It is recommended to disable browser validation.

HTML 5 Validation Attributes

required
maxlength
pattern

min
max

Angular Form Validation Properties

touched/untouched
pristine/dirty
valid/invalid

Code Sample without Model Binding

create-employee.component.ts

```
import { Component, OnInit } from '@angular/core';
import { NgForm } from '@angular/forms';
import { Department } from '../model/department.model';
import { Employee } from '../model/employee.model';

@Component({
  selector: 'app-create-employee',
  templateUrl: './create-employee.component.html',
  styleUrls: ['./create-employee.component.css']
})
export class CreateEmployeeComponent implements OnInit {
  toggler=true
  gender="male"
  isActive = true
  department="3"
  departments: Department[] = [
    {id:1, name:"Dep1"},
    {id:2, name:"Dep2"},
    {id:3, name:"Dep3"}
  ];
  constructor() { }

  ngOnInit() {
  }

  doToggle(){
    this.toggler = !this.toggler
  }

  saveEmployee(empForm: NgForm) : void{ //should be ngFormType
    console.log(empForm.value)
  }
}
```

create-employee.component.html

```
<!--an automatic form model is created-->
```

```
<form #employeeForm="ngForm" (ngSubmit)=saveEmployee(employeeForm)>
    <!--template reference variable, import import { FormsModule } from
'@angular/forms' in apps module for this-->
    <!--#employeeForm="ngForm" means we're exporting ngForm in to employeeForm
variable -->
    <div class="panel panel-primary">
        <div class="panel-heading">
            <h3 class="panel-title">Create Employee</h3>
        </div>
        <div class="panel-body">

            <!--#fullName1="ngModel" is exporting ngModel directive into a local
template reference variable-->
                <!--#fullName1: is called local variable or template variable or
local template reference variable-->
                <!--with [(ngModel)], name attribute must be set-->
                <!--fullName1 coz name="fullName" already exists -->
                <!--[class.has-error]="fullName1.invalid, class binding,
conditional to apply only if is invalid-->
            <div class="form-group" [class.has-error]="fullName1.invalid &&
fullName1.touched"
            [class.has-success]="fullName1.valid">
                <label for="fullName">Full Name</label>
                <input required [(ngModel)]="fullName" name="fullName" type="text"
id="fullName" class="form-control"
                #fullName1="ngModel">
                <span class="help-block" *ngIf="fullName1.invalid &&
fullName1.touched">
                    Full Name is required
                </span>
            </div>

            <div class="form-group">
                <label for="email">Email</label> <!-- for="email" links label with
input with id:email -->
                <input [(ngModel)]="email" name="email" type="text" id="email"
class="form-control">
            </div>

            <div class="form-group">
                <label for="phone">Phone</label>
                <input [(ngModel)]="phone" name="phone" type="text" id="phone"
class="form-control">
            </div>

            <div class="form-group">
                <label>Gender</label>
                <div class="form-control">
                    <label class="radio-inline">
                        <input type="radio" value="male" name="gender"
[(ngModel)]="gender" disabled>
                        <!--to make male checked by default, set gender property in
component as "male"-->
                        Male
                    </label>
                    <label class="radio-inline">
                        <input type="radio" value="female" name="gender"
[(ngModel)]="gender" disabled>
                        <!--disabled attribute will disable both radios-->
                        <!--setting both male and female with name="gender" makes
both controls related to each other-->
                        Female
                    </label>
                </div>
            </div>

            <div class="form-group">
                <div class="form-control">
                    <label class="checkbox-inline">
                        <input type="checkbox" name="isActive"
[(ngModel)]="isActive">
                        <!--to make checked by default, set isActive property in
component as true-->
                        Is Active
                    </label>
                </div>
            </div>

            <div class="form-group">
                <label for="department">Department</label>
                <select id="department" name="department" [(ngModel)]="department"
class="form-control">
                    <option *ngFor="let dept of departments" [value]="dept.id">
                        <!--use square bracket when binding properties
eg.[value]-->
                        {{dept.name}}
                    </option>
                    <!--<option value="1">Dep1</option>
                    <option value="2">Dep2</option>
                    <option value="3">Dep3</option>-->
                </select>
            </div>

            <!--with bsDatepicker-->
            <div class="form-group">
                <div class="form-control">
                    <input id="dob" type="text" name="dob" bsDatepicker
[(ngModel)]="dob">
                </div>
            </div>

            <!--without bsDatepicker-->
            <div class="form-group">
                <div class="form-control">
                    <input id="dob1" type="date" name="dob1" [(ngModel)]="dob1">
                </div>
            </div>
```

```
        </div>
        <div class="panel-footer">

            <!--toggle button text-->
            <!--type button should be explicitly specified, else it will behave
like submit-->
            <button type="button" class="btn btn-primary" (click)="doToggle()">
                {{ toggler ? "Show " : "Hide " }} Me
            </button>

            <button type="submit" class="btn btn-primary"
[disabled]="employeeForm.invalid">Save</button> <!--type submit-->
        </div>
    </div>
</form>
Angular generated form model: {{employeeForm.value | json}}

<div><h3>FullName</h3>></div>
<div>touched: {{fullName1.touched}}</div>
<div>untouched: {{fullName1.untouched}}</div>
<div>pristine: {{fullName1.pristine}}</div>
<div>dirty: {{fullName1.dirty}}</div>
<div>valid: {{fullName1.valid}}</div>
<div>invalid: {{fullName1.invalid}}</div>
<div><h3>Form</h3>></div>
<div>touched: {{employeeForm.touched}}</div>
<div>untouched: {{employeeForm.untouched}}</div>
<div>pristine: {{employeeForm.pristine}}</div>
<div>dirty: {{employeeForm.dirty}}</div>
<div>valid: {{employeeForm.valid}}</div>
<div>invalid: {{employeeForm.invalid}}</div>
<div>invalid: {{employeeForm.value.department}}</div>
```

Code Sample with Model Binding

create-employee.component.ts

```
import { Component, OnInit } from '@angular/core';
import { NgForm } from '@angular/forms';
import { Department } from '../model/department.model';
import { Employee } from '../model/employee.model';

@Component({
  selector: 'app-create-employee',
  templateUrl: './create-employee.component.html',
  styleUrls: ['./create-employee.component.css']
})
export class CreateEmployeeComponent implements OnInit {
  toggler=true
  gender="male"
  isActive = true
  department="3"
  employee: Employee = {
    id: null,
    name: "Rajeesh Vengalapurath",
    contactPreference: null,
    dateOfBirth: null,
    department: null,
    gender: null,
    isActive: null,
    email: null,
    phoneNumber: null,
    photoPath: null
  }
  departments: Department[] = [
    {id:1, name:"Dep1"},
    {id:2, name:"Dep2"},
    {id:3, name:"Dep3"}
  ];
  constructor() { }

  ngOnInit() {
  }

  doToggle(){
    this.toggler = !this.toggler
  }

  saveEmployee() : void{
    console.log(this.employee)
  }
}
```

create-employee.component.html

```
<!--an automatic form model is created-->
<form #employeeForm="ngForm" (ngSubmit)=saveEmployee()>
    <!--template reference variable, import import { FormsModule } from
'@angular/forms' in apps module for this-->
    <!--#employeeForm="ngForm" means we're exporting ngForm in to employeeForm
variable -->
    <div class="panel panel-primary">
        <div class="panel-heading">
            <h3 class="panel-title">Create Employee</h3>
        </div>
        <div class="panel-body">

            <!--#name="ngModel" is exporting ngModel directive into a local
template reference variable-->
                <!--#name: is called local variable or template variable or local
template reference variable-->
                <!--with [(ngModel)], name attribute must be set-->
                <!--name coz name="fullName" already exists -->
                <!--[class.has-error]="name.invalid, class binding, conditional to
apply only if is invalid-->
            <div class="form-group" [class.has-error]="name.invalid &&
name.touched"
```

```
            [class.has-success]="name.valid">
                <label for="name">Name</label>
                <input required [(ngModel)]="employee.name" name="name" type="text"
id="name" class="form-control"
                #name="ngModel">
                <span class="help-block" *ngIf="name.invalid && name.touched">
                    Name is required
                </span>
            </div>

            <div class="form-group">
                <label for="email">Email</label> <!-- for="email" links label with
input with id:email -->
                <input [(ngModel)]="employee.email" name="email" type="text"
id="email" class="form-control">
            </div>

            <div class="form-group">
                <label for="phone">Phone</label>
                <input [(ngModel)]="employee.phone" name="phone" type="text"
id="phone" class="form-control">
            </div>

            <div class="form-group">
                <label>Gender</label>
                <div class="form-control">
                    <label class="radio-inline">
                        <input type="radio" value="male" name="gender"
[(ngModel)]="employee.gender" disabled>
                        <!--to make male checked by default, set gender property in
component as "male"-->
                        Male
                    </label>
                    <label class="radio-inline">
                        <input type="radio" value="female" name="gender"
[(ngModel)]="employee.gender" disabled>
                        <!--disabled attribute will disable both radios-->
                        <!--setting both male and female with name="gender" makes
both controls related to each other-->
                        Female
                    </label>
                </div>
            </div>

            <div class="form-group">
                <div class="form-control">
                    <label class="checkbox-inline">
                        <input type="checkbox" name="isActive"
[(ngModel)]="employee.isActive">
                        <!--to make checked by default, set isActive property in
component as true-->
                        Is Active
                    </label>
                </div>
            </div>

            <div class="form-group">
                <label for="department">Department</label>
                <select id="department" name="department"
[(ngModel)]="employee.department" class="form-control">
                    <option *ngFor="let dept of departments" [value]="dept.id">
                        <!--use square bracket when binding properties
eg.[value]-->
                        {{dept.name}}
                    </option>
                    <!--<option value="1">Dep1</option>
                    <option value="2">Dep2</option>
                    <option value="3">Dep3</option>-->
                </select>
            </div>

            <!--with bsDatepicker-->
            <div class="form-group">
                <div class="form-control">
                    <input id="dob" type="text" name="dob" bsDatepicker
[(ngModel)]="dob">
                </div>
            </div>

            <!--without bsDatepicker-->
            <div class="form-group">
                <div class="form-control">
                    <input id="dob1" type="date" name="dob1"
[(ngModel)]="employee.dateOfBirth">
                </div>
            </div>

        </div>
        <div class="panel-footer">

            <!--toggle button text-->
            <!--type button should be explicitly specified, else it will behave
like submit-->
            <button type="button" class="btn btn-primary" (click)="doToggle()">
                {{ toggler ? "Show " : "Hide " }} Me
            </button>

            <button type="submit" class="btn btn-primary"
[disabled]="employeeForm.invalid">Save</button> <!--type submit-->
        </div>
    </div>
</form>
Angular generated form model: {{employeeForm.value | json}}

<div><h3>FullName</h3>></div>
<div>touched: {{name.touched}}</div>
<div>untouched: {{name.untouched}}</div>
<div>pristine: {{name.pristine}}</div>
```

```
<div>dirty: {{name.dirty}}</div>
<div>valid: {{name.valid}}</div>
<div>invalid: {{name.invalid}}</div>
<div><h3>Form</h3>></div>
<div>touched: {{employeeForm.touched}}</div>
<div>untouched: {{employeeForm.untouched}}</div>
<div>pristine: {{employeeForm.pristine}}</div>
<div>dirty: {{employeeForm.dirty}}</div>
<div>valid: {{employeeForm.valid}}</div>
<div>invalid: {{employeeForm.invalid}}</div>
<div>invalid: {{employeeForm.value.department}}</div>

<br>
Employee: {{ employee | json }}
```

ngx-bootstrap UI components

1. `npm install ngx-bootstrap --save`
2. `npm install bootstrap@3 --save`
3. `npm install @angular/platform-browser --save` (was not installed)
4. Open **angular.json** file and specify the path to the Bootstrap stylesheet (bootstrap.min.css)
   ```
   ...
   "styles": [
   "src/styles.css",
   "node_modules/bootstrap/dist/css/bootstrap.css",
   "node_modules/ngx-bootstrap/datepicker/bs-datepicker.css"
   ],
   "scripts": [
   "node_modules/bootstrap/dist/js/bootstrap.js"
   ]

   ...
   ```
5. **App.module.ts (to add datetime picker)**
   a.
      ```
      import { BrowserAnimationsModule } from
      '@angular/platform-browser/animations'
      import { BsDatepickerModule } from 'ngx-bootstrap/datepicker'
      ...
      imports: [
      ...
      BrowserAnimationsModule,
      BsDatepickerModule.forRoot(), //forRoot??
      ...
        ],
      ```
6. .Html
   ```
   <div class="form-group">
          <div class="form-control">
                 <input id="dob" type="text" name="dob" bsDatepicker
                        [(ngModel)]="dob">
          </div>
   </div>
   ```

Angular Router Guards

Steps

1. Build the route guard
2. Register the guard with angular dependency injection system
3. Tie the guard to a route

CanDeactivate

Guard navigation away from the current route

CanActivate

Guard navigation to a route

CanActivateChild

Guard navigation to a child route

CanLoad

Guard navigation to a feature module loaded

Resolve

Perform route data retrieval before route activation

Required route parameter vs optional route parameter

1. Required route parameters are part of route configuration (app module) where as optional route parameters are not

2. Required route parameters are used in pattern matching i.e when mapping an incoming URL to a named route in our application. Optional route parameters are not used in pattern matching.

3. Optional route parameters must be passed after the required route parameters if any.

4. In general, prefer a required route parameter when the value is simple and mandatory. For example, to view a specific employee details, the ID parameter is mandatory and it is a simple integer. On the other hand, prefer an optional route parameter when the value is optional and complex. For example you want to send many search parameters like NAME, AGE, DEPARTMENT, DOB etc...from the SEARCH PAGE to LIST PAGE.

3 types of parameters in Angular

1. Required parameters
2. Optional parameters
3. Query parameters

Query parameters

Query parameters are usually used when you want the parameters on the route to be optional and when you want to retain those parameters across multiple routes.

Fake REST API JSON Server

Install

npm install -g json-server

Start Server

json-server --watch db.json

Next Heading2 Here

Code

create-employee-can-deactivate-guard.service.ts

```
import { CreateEmployeeComponent } from './create-employee.component';
import { CanDeactivate } from '@angular/router';
import { Injectable } from '@angular/core';

@Injectable()
export class CreateEmployeeCanDeactivateGuardService implements
CanDeactivate<CreateEmployeeComponent>{
    canDeactivate(component: CreateEmployeeComponent) : boolean  { //no need of
other parameters
        if(component.createEmployeeForm.dirty){
            return confirm('Are you sure you want to discard your changes?')
        }
        return true;
    }
}
```

create-employee.component.ts

```
import { ..., ViewChild } from '@angular/core';
...
```

```
@Component({
  ...
})
export class CreateEmployeeComponent implements OnInit {

  @ViewChild('employeeForm', {static: false}) //#employeeForm is the template
reference variable of form on view
  public createEmployeeForm: NgForm //createEmployeeForm or any name
  ...
}
```

app.module.ts

```
...
import { CreateEmployeeCanDeactivateGuardService } from
'./employees/create-employee-can-deactivate-guard.service';

const appRouts: Routes = [
  ...
  {
    path: 'create',
    component: CreateEmployeeComponent,
    canDeactivate: [CreateEmployeeCanDeactivateGuardService]
  },
  ...
];
```

Subscribe to route parameter changes

```
...
import { ActivatedRoute, Router } from '@angular/router';
import { EmployeeService } from './employee.service';
import { Employee } from '../model/employee.model';

@Component({
...
})
export class EmployeeDetailsComponent implements OnInit {

  employee: Employee
  private _id: number

  constructor(private _activatedRoute: ActivatedRoute
    ,private _employeeService: EmployeeService
    ,private _router: Router) { }

  ngOnInit() {

    /*snapshot method
    viewNextEmployee() method won't work coz ngOnInit is called only once. So
snapshot only works once
    this._id = +this._activatedRoute.snapshot.params['id']
    this.employee = this._employeeService.getEmployee(this._id)*/

    //observable method. subscribe gets parameters everytime the route is changed
    //paramMap is angular 4+
    this._activatedRoute.paramMap.subscribe(params=>{
      this._id = +params.get('id')
      this.employee = this._employeeService.getEmployee(this._id)
    })
  }

  //called from view on net employee button click
  viewNextEmployee() {
    if(this._id<2)
      this._id++;
    else
      this._id=1;

    this._router.navigate(['/employees', this._id])

  }
}
```

Template driven forms vs Reactive forms

As the name implies, Template Driven Forms are heavy on the template meaning we create the form completely in HTML. Template driven forms are easy to build and understand. They are great for creating simple forms. However, creating complex forms using template driven approach is not recommended as the HTML can get very complicated and messy. It is not easy to unit test template forms as the logic is in the HTML.

Reactive forms on the other hand allow us to build the form completely in code. This is more flexible and has many benefits over template forms. For example, it is easy to add form input elements dynamically and adjust validation at runtime based on the decisions made in code. It is also easy to unit test as most of the logic and validation is in the component class. The only downside of reactive forms is that they require more code than template forms.

Angular Reactive Forms

setValue vs patchValue

setValue resets the entire form with new value. patchValue can change individual fields

```
this.employeeForm.setValue({
      fullName: 'Rajeesh Vengalapurath',
      email: 'rajeeshvp@gmail.com',
      skills:{
        skillName: 'Angular',
        experienceInYears: 5,
        proficiency: 'advanced'
      }
    })

    this.employeeForm.patchValue({
      fullName: 'Rajeesh VP',
      skills:{
        experienceInYears: 6,
      }
    })
  }
```

Code

Create-employee.component.ts

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

@Component({
  ...
})

export class CreateEmployeeComponent implements OnInit {
  employeeForm: FormGroup
  constructor() { }

  ngOnInit() {
    this.employeeForm = new FormGroup({
      fullName: new FormControl(), //as key value pair
      email: new FormControl()
    })
  }

  onSubmit(){
    console.log(this.employeeForm.touched);
    this.employeeForm.controls.fullName.setValue("test")
  }
}
```

create-employee.component.html

```
<!--[formGroup]="employeeForm" same as in ts file-->
<form [formGroup]="employeeForm" class="form-horizontal">

    <div class="panel panel-primary">

        <div class="panel-heading">
            <h3 class="panel-title">Create Employee</h3>
        </div>

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

```
                <label class="col-sm-2 control-label" for="fullName">Full
Name</label>
                <div class="col-sm-8">
                    <!--formControlName="fullName" same as in ts file-->
                    <input formControlName="fullName" class="form-control"
id="fullName" type="text">
                </div>
            </div>

            <div class="form-group">
                <label class="col-sm-2 control-label" for="email">Email</label>
                <div class="col-sm-8">
                    <!--formControlName="email" same as in ts file-->
                    <input formControlName="email" class="form-control" id="email"
type="text">
                </div>
            </div>

        </div>

        <div class="panel-footer">
            <button (click)="onSubmit()" class="btn-btn-primary"
type="submit">Save</button>
        </div>

    </div>
</form>
```

ToDo

CodAffection
https://goo.gl/9shbS8 : Simple CRUD in Angular 7 + Web API
https://goo.gl/NNf5sp : Angular Material Popup
https://goo.gl/VXhCoC : Complete Angular Material Playlist

Read first: https://angular.io/guide/reactive-forms
https://angular.io/guide/router read completly
https://angular.io/guide/form-validation
Check route sample app for UI development
Angular CRUD
Check angular CLI tutorial from kudvenkat
Check angular2 course from kudvenkat after crud
Bootstrap
Only if have time: structural directives in depth
https://angular.io/guide/structural-directives

https://www.youtube.com/watch?v=jZJY70PY10w&list=PL6n9fhu94yhXwcl3a6rIfAI7QmGYIkfK5&index=42

Angular Flex Layout

Responsive Layouts with @angular/Flex-Layout – Ekaterina Orlova & Thomas Burleson

https://www.youtube.com/watch?v=geqjUtKJX5s

About the video

Regardless of the power of the Angular platform, developers always struggle UI component Layouts. Using Flexbox CSS, the HTML layouts becomes responsive to viewport size changes. HTML containers will auto-adjust their sizes and child elements will auto-adjust positioning and sizes accordingly.

But using Flexbox CSS is hard... and frustrating... and seemingly complex. You have to become a Flexbox CSS expert! Or do you? With @angular/flex-layout, developers have an Angular-native, HTML markup API that makes it super easy to layout their UI components. And the Flex-Layout will generate and apply, under-the-hood, the best FlexBox CSS for your needs.

And the Flex-Layout will generate and apply, under-the-hood, the best FlexBox CSS for your needs. Even better, Flex Layout has a Responsive API that makes it super easy to adapt the UI to different devices.

Hide or show components, change layout directions, change stylings... all super easy with @angular/flex-layout. Come to this presentation to learn more about @angular/flex-layout, how it compares to Angular Material, how it compares to Bootstrap CSS grids. Come to learn why this should be a critical new tool for your Angular developer toolbox.

Static API

API for DOM Containers	
`fxLayout`	`<div fxLayout="row" fxLayout.xs="column"></div>`
`fxLayoutWrap`	`<div fxLayoutWrap></div>`
`fxLayoutGap`	`<div fxLayoutGap="10px"></div>`
`fxLayoutAlign`	`<div fxLayoutAlign="start stretch"></div>`

fxLayoutOptions	
Value	Equivalent CSS
`default`	`{flex-direction: row}`
`row`	`{flex-direction: row}`
`row-reverse`	`{flex-direction: row-reverse}`
`column`	`{flex-direction: column}`
`column-reverse`	`{flex-direction: column-reverse}`

API for container children	
`fxFlex`	`<div fxFlex=”1 2 calc(15em + 20px)”></div>`
`fxFlexOrder`	`<div fxFlexOrder=”2”></div>`
`fxFlexAlign`	`<div fxFlexAlign=”center”></div>`
`fxFill`	`</div fxFill></div>`

Responsive API

Media Queries + Aliases	
Break Point Alias	mediaQuery
`xs`	`‘screen and (max-width: 599px)’`
`sm`	`‘screen and (min-width: 600px) and (max-width: 959px)’`
`md`	`‘screen and (min-width: 960px) and (max-width: 1279px)’`
`lg`	`‘screen and (min-width: 1280px) and (max-width: 1919px)’`
`xl`	`‘screen and (min-width: 1920px) and (max-width: 5000px)’`
`lt-sm (less than)`	`‘screen and (max-width: 599px)’`
`lt-md`	`‘screen and (max-width: 959px)’`
`lt-lg`	`‘screen and (max-width: 1279px)’`
`lt-xl`	`‘screen and (max-width: 1919px)’`

gt-xs (greater than)	‘screen and (min-width: 600px)’
gt-sm	‘screen and (min-width: 960px)’
gt-md	‘screen and (min-width: 1280px)’
gt-lg	‘screen and (min-width: 1920px)’

Examples
fxLayout.sm = ”..”
fxLayoutAlign.md = ”..”
fxHide.gt-sm = “..”	

Special responsive features	
fxShow	`<div fxShow [fxShow.xs]=”isVisibleOnMobile()”></div>`
fxHide	`<div fxShow [fxShow.gt-sm]=”isVisibleOnDesktop()”></div>`
ngClass	`<div [ngClass.sm]=”{‘fxClass-sm’: hasStyle}”></div>`
ngStyle	`<div [ngStyle.xs]=”{color: ‘blue’}”></div>`

Custom Directives

```
ng g directive highlight
```

Highlight.directive.ts

```
import { Directive, ElementRef, HostListener } from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {

  constructor(private el: ElementRef) {  //element injection
    el.nativeElement.style.backgroundColor = 'red'
  }

  @HostListener('mouseenter') onMouseEnter() {
    this.el.nativeElement.style.backgroundColor = 'yellow'
  }
}
```

app.component.html

```
<div appHighlight>Test</div>
```

Other options

1. **`@Input() highlightColor: string;`**
   - `<p appHighlight highlightColor="yellow">Highlighted in yellow</p>`
     `<p appHighlight [highlightColor]="'orange'">Highlighted in orange</p>`
2. **`@Input() appHighlight: string; //same name as directive`**
   `@Input('appHighlight') highlightColor: string; //or this`
   - `<p [appHighlight]="color">Highlight me!</p>`

Custom Structural Directive

```
ng g d myNgIf
```

my-ng-if.directive.ts

```
import { Directive, TemplateRef, OnInit, ViewContainerRef } from '@angular/core';

@Directive({
  selector: '[myNgIf]'
})
export class MyNgIfDirective implements OnInit{

  constructor(
    private viewContainer: ViewContainerRef,
    private templateReference: TemplateRef<object>
  ) { }

  ngOnInit(){
    const condition = false //true
    if(condition){
      this.viewContainer.createEmbeddedView(this.templateReference)

      /* ngFor would be like this
      for (let index = 0; index < 10; index++) {
        this.viewContainer.createEmbeddedView(this.templateReference)
      }
      */
    }
    else{
      this.viewContainer.clear()
    }
  }
}
```

app.component.html

```
<div *myNgIf>Test</div>
```

Microsyntax

```
*ngFor="let item of [1,2,3]"
<ng-template ngFor let-item [ngForOf]="[1,2,3]">

*ngFor="let item of [1,2,3] as items; trackBy: myTrack; index as i"
<ng-template ngFor let-item [ngForOf]="[1,2,3]" let-items="ngForOf"
[ngForTrackBy]="myTrack" let-i="index">

*ngIf="exp"
<ng-template [ngIf]="exp">

*ngIf="exp as value"
<ng-template [ngIf]="exp" let-value="ngIf">
```

<ng-template>

Ng-template is a virtual element. It is not created on DOM unless it is called. Can be displayed **only** using structural directives **ngIf ngFor nfSwitch**

```
<ng-template #otherContent>
       <p>Job List</p>
       <jobs></jobs> Another component
</ng-template>

This div will be replaced by template if false
<div *ngIf="showMyContent; else otherContent>
       <p>No Job List</p>
<div>

<button type="button" (click)="toggleContent()>Toggle</button>
-------
```

ViewChild

Sample 1

```
<button #btn>Click</button>

@ViewChild('btn') btn: ElementRef<HTMLButtonElement>;

ngAfterViewInit() {
   //not recommended. Use renderer instead
   this.btn.nativeElement.style.backgroundColor='red'
   this.btn.nativeElement.innerText = "test"
}
```

Sample 2

```
<app-my-component #child></app-my-component>

@ViewChild('child') child: MyComponentComponent

ngAfterViewInit() {
   this.child.name ="rajeesh" //name is an input property of child component
}
```

Mat-table

- Builds on the foundation of the CDK data-table and uses a similar interface for its data input and template
- Element and attribute selectors will be prefixed with **mat-** instead of **cdk-**
- Provide data to the table by passing a data array to the table's **dataSource** input.