# Angular – Part 2

By

Srikanth Pragada

## Life Cycle Hooks

- Angular calls lifecycle hook methods on directives and components as it creates, changes, and destroys them.
- Angular manages lifecycle of components and directives.
- A directive has the same set of lifecycle hooks, minus the hooks that are specific to component content and views.
- Angular inspects directive and component classes and calls the hook methods if they are defined.

✓ `ngOnChanges()`

✓ `ngOnInit()`

✓ `ngDoCheck()`

✓ `ngAfterContentInit()`

✓ `ngAfterContentChecked()`

✓ `ngAfterViewInit()`

✓ `ngAfterViewChecked()`

✓ `ngOnDestroy()`

## Life Cycle Hooks

**ngOnChanges() of OnChanges**

- ❑ Respond when Angular (re)sets data-bound input properties.
- ❑ The method receives a SimpleChanges object of current and previous property values.
- ❑ Called before ngOnInit() and whenever one or more data-bound input properties change.

**ngOnInit() of OnInit**

- ❑ Initialize the directive/component after Angular first displays the data-bound properties and sets the directive/component's input properties.
- ❑ Called once, after the first ngOnChanges().

**ngDoCheck() of DoCheck**

- ❑ Detect and act upon changes that Angular can't or won't detect on its own.
- ❑ Called during every change detection run, immediately after ngOnChanges() and ngOnInit().

**ngAfterContentInit() of AfterContentInit**

- ❑ Respond after Angular projects external content into the component's view.
- ❑ Called once after the first ngDoCheck().
- ❑ A component-only hook.

## Life Cycle Hooks

**ngAfterContentChecked() of AfterContentChecked**

- ❑ Respond after Angular checks the content projected into the component.
- ❑ Called after the ngAfterContentInit() and every subsequent ngDoCheck().
- ❑ A component-only hook.

**ngAfterViewInit() of AfterViewInit**

- ❑ Respond after Angular initializes the component's views and child views.
- ❑ Called once after the first ngAfterContentChecked().
- ❑ A component-only hook.

**ngAfterViewChecked() of AfterViewChecked**

- ❑ Respond after Angular checks the component's views and child views.
- ❑ Called after the ngAfterViewInit and every subsequent ngAfterContentChecked().
- ❑ A component-only hook.

**ngOnDestroy of OnDestroy**

- ❑ Cleanup just before Angular destroys the directive/component.
- ❑ Unsubscribe Observables and detach event handlers to avoid memory leaks.
- ❑ Called just before Angular destroys the directive/component.

## Life Cycle Hooks – Example

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

@Component({
  selector: 'st-lifecycle',
  template: `<h1>Life Cycle Hooks</h1>`,
})
export class LifeCycleComponent implements OnInit, OnDestroy {
    ngOnInit() {
        console.log("Component Initialized!");
    }
    ngOnDestroy() {
        console.log("Component Destroyed!");
    }
}
```

# Dependency Injection

- It is a coding pattern in which a class receives its dependencies from external sources rather than creating them itself.
- Angular ships with its own dependency injection framework.
- Decorators @Component, @Directive and @Pipe are subtypes of @Injectable().
- The @Injectable() decorator identifies a class as a target for instantiation by an injector.
- Angular creates an application-wide injector for you during the bootstrap process.
- You do have to configure the injector by registering the providers that create the services the application requires.
- At runtime, injectors can read class metadata in the transpiled JavaScript code and use the constructor parameter type information to determine what things to inject.
- You can either register a provider within an NgModule or in application components using providers property, which is an array.
- Any provider registered within an NgModule will be accessible in the entire application.

## Services

- ❑ A service is a class with a specific purpose.
- ❑ It is injected into other components and services that need it.
- ❑ It must be declared as injectable using @Injectable decorator.
- ❑ In order to refer to a service and get it injected into our component, we have to declare a parameter of service in constructor.
- ❑ Use services as singletons within the same injector.
- ❑ Use them for sharing data and functionality.
- ❑ Services are ideal for sharing stateful in-memory data.
- ❑ A service can depend on another service.
- ❑ The Angular injector is hierarchical. When providing the service to a top level component, that instance is shared and available to all child components of that top level component.
- ❑ When two different components need different instances of a service, it would be better to provide the service at the component level that needs the new and separate instance.

## Steps to create and use service

- ❑ Create a class with **@Injectable()** decorator.
- ❑ Provide required methods in this class.
- ❑ Register service class with Module or a Component by listing it in providers array of **@NgModule** or **@Component** decorators.
- ❑ In component, inject service into a property by using  a **private parameter** in constructor which is of service type.

## Providers Property

- ❑ A service is to be registered before it is used.
- ❑ A service provider is to be registered with the injector, or it won't know how to create the service.
- ❑ The **providers** property of **@Component** or **@NgModule** decorators is used to list services to be registered with the component or module.
- ❑ When a service is registered using module it is available to all components of that module.

## log.service.ts

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

@Injectable()
export class LogService {
   log(msg : string ) : void {
       console.log(msg);
   }
}
```

# uselog.component.ts

```
import { Component } from '@angular/core';
import { LogService} from './log.service';

@Component({
  selector: 'test-log',
  templateUrl : 'app/uselog.component.html',
  providers : [ LogService ]
})
export class UseLogComponent  {
    // Injects LogService into logService property
    constructor(private logService : LogService) {
    }
    logMessage(msg : string) : void
    {
       this.logService.log(msg);
    }
}
```

## Optional dependency

When a service is optional, inform Angular that the dependency is optional by annotating the constructor argument with @**Optional**().

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

constructor(@Optional() private logger: LogService) {
  if(this.logger) {
       this.logger.logMessage(some_message);
  }
}
```

# Component Interactions

- ❑ Passing data from parent to child using @Input() decorator.
- ❑ Sending event from child to parent using @Output() decorator.
- ❑ Parent and child communicating via a service.

## Input property

- ❑ In order to send data from parent component, a property in child component must be declared as input property.
- ❑ Input property receives data.
- ❑ Input property is marked with @Input() decorator.
- ❑ It is also possible to mark input properties using inputs array in @Component() decorator.

# @Input() Example

course.component.ts

```
import { Component, Input } from '@angular/core';
import { Course } from './Course';
@Component({
  selector: 'course',
  templateUrl: './course.component.html'
})
export class CourseComponent   {
  @Input() course : Course;

}
```

course.component.html

```
<h2>{{course.title}}</h2>
<h3>{{course.duration}} </h3>
<h3>{{course.fee}} </h3>
```

courses-list.component.html

```
<div *ngFor="let course of courses">
      <course [course]="course"></course>
</div>
```

# Using setter method for input property

course.component.ts

```
@Component ( {…} )
export class CourseComponent   {
  private _course : Course;

  @Input()
  set course(course : Course) {
       this._course = course;
       if(this._course.fee < 1000)
           this._course.fee = 1000;
  }

}
```

course.component.html

```
<h2>{{ _course.title }}</h2>
<h3>{{ _course.duration }} </h3>
<h3>{{ _course.fee }} </h3>
```

## Parent listens to child event

- ❑ Child component can emit an event to communicate with parent.
- ❑ Child declares EventEmitter as an output property using @Output() decorator.
- ❑ Using **EventEmitter** child emits event so that parent can respond to event.

## Child sending event to parent example

course.component.ts

```
import { Component, Input, Output, EventEmitter} from '@angular/core';
import { Course } from './Course';
@Component({
  selector: 'course',
  templateUrl : './course.component.html'
})
export class CourseComponent {

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

 delete() {
    this.deleteCourse.emit(this.course.title);
 }
}
```

## Child sending event to parent example

course.component.html

```
<h2>{{course.title}}</h2>
<h3>{{course.duration}} </h3>
<h3>{{course.fee}} </h3>

<button (click)="delete()">Delete</button>
```

## Child sending event to parent example

courses-list.component.ts

```
import { Component } from '@angular/core';
import { Course } from './Course';
@Component({
  selector: 'courses-list',
  templateUrl: './courses-list.component.html'
})
export class CoursesListComponent   {
   courses : Course[] =
    [ new Course ( "Android Programming", 40,  4000),
      new Course ( "Microsoft.Net",60, 6000)
    ];

    deleteCourse(title) {
      console.log("Deleting course :" + title);
      // code
    }
}
```

## Child sending event to parent example

courses-list.component.html

```
<h1>List of Courses </h1>
<div *ngFor="let course of courses">
   <course [course]="course"   (deleteCourse)="deleteCourse($event)">
</course></div>
```

# Forms

- An Angular form coordinates a set of data-bound user controls, tracks changes, validates input, and presents errors.
- Angular supports two types of forms – Template Driven and Reactive.

Template Driven Forms

Reactive Forms
(Model Driven)

# Template Driven

- ❑ These forms are built by creating templates using Angular template syntax with form-specific directives.
- ❑ Create properties that are to be bound to controls in the form.
- ❑ Use two-way data binding with ngModel directive.
- ❑ In the application root module import FormsModule from @angular/forms.
- ❑ List FormsModule in imports array of the module.
- ❑ Specify form name using NgForm directive to the <form> tag.
- ❑ The NgForm directive supplements the form element with additional features. It holds the controls you created for the elements with an ngModel directive and name attribute, and monitors their properties, including their validity. It also has its own valid property which is true only if every contained control is valid.
- ❑ Internally, Angular creates FormControl instances and registers them with an NgForm directive that Angular attached to the <form> tag.
- ❑ Each FormControl is registered under the name you assigned to the name attribute.
- ❑ The NgModel directive doesn't just track state; it updates the control with special Angular CSS classes that reflect the state.
- ❑ We can leverage those class names to change the appearance of the control.

## Example

```
 <form (ngSubmit)="onSubmit(loginForm)" #loginForm="ngForm">
   Username <br>
   <input type="text" #uname="ngModel" [(ngModel)]="username"
                 name="username"  minlength="4"   required>
   <span *ngIf="uname.dirty && uname.errors">
     <span *ngIf="uname.errors.required">Username is required! </span>
     <span *ngIf="uname.errors.minlength">Enter at least 4 chars</span>
   </span>
   <p></p>
   ...
   <button  [disabled]= "!loginForm.valid">Login</button>
   <button  [hidden]="!loginForm.dirty" type="reset">Reset</button>
</form>
```

# NgForm  Properties

- ❑ NgForm directive creates a top-level FormGroup and binds it to form to track form values and validation status.
- ❑ Associate a local template variable with ngForm directive.  Ex :   #loginForm =“ngForm”.
- ❑ This template variable provides access to properties related to form and child controls.
- ❑ To register child controls with the form, we have to use NgModel with name attribute.

```
controls
dirty
errors
invalid
valid
value
touched
submitted
```

## Classes associated with FormControl

| State | Class if true | Class if false |
|---|---|---|
| The control has been visited. | ng-touched | ng-untouched |
| The control's value has changed. | ng-dirty | ng-pristine |
| The control's value is valid. | ng-valid | ng-invalid |

You can access classes associated with a form control using its template reference variable.

```
<input type="text" id="name"
           required
           minlength="4"
           [(ngModel)]="name"
           name="name"
           #custname="ngModel" />
```

## Submitting Form

- Use ngSubmit directive to take action when form is submitted with a submit button.

```
<form (ngSubmit)="onSubmit()" #loginForm="ngForm">

   <button [disabled]="!loginForm.valid">Submit</button>
   <button type="reset">Reset</button>

</form>
```

## Reactive Forms (Model Driven Forms)

- ❑ In this we create a tree of Angular form control objects in the component class and bind them to native form. control elements in the component template.
- ❑ You create and manipulate form control objects directly in the component class.
- ❑ Here are steps to create Reactive Form:

  - ✓ Create data model
  - ✓ Create reactive form component like FormGroup and FormControl
  - ✓ Create template - html
  - ✓ Import ReactiveFormsModule and include in modules array of Module

# FormControl

- Each object of FormControl represents a control in form.
- An object of FormControl is created for each control and associated with a property in Component.

| Property | Description |
|---|---|
| value | Value of the control. |
| status | Validity of a FormControl. Possible values: VALID, INVALID, PENDING, or DISABLED. |
| pristine, dirty | True if the user has not changed the value in the UI. Its opposite is dirty. |
| untouched, touched | True if the control user has not yet entered the HTML control and triggered its blur event. |
| parent | The parent control – FormGroup or FormArray. |
| valid, invalid | True or false indicating whether value in control is valid. |
| errors | Collection of errors as ValidationErrors or null. |
| disabled, enabled | True or false. |
| valueChanges | Observable<Any>  - Emits an event every time the value of the control changes, in UI or programmatically. |

## FormGroup

- ❑ A FormGroup aggregates the values of each child FormControl into one object, with each control name as the key.
- ❑ It calculates its status by reducing the statuses of its children. For example, if one of the controls in a group is invalid, the entire group becomes invalid.
- ❑ FormGroup provides methods using which we can get or set values to FormControls.

```
✓ controls : {[key: string]: AbstractControl}
✓ addControl(name: string, control: AbstractControl) : void
✓ removeControl(name: string) : void
✓ setControl(name: string, control: AbstractControl) : void
✓ contains(controlName: string) : Boolean
✓ setValue(value: { [key: string]: any; },
               options: { onlySelf?: boolean; emitEvent?: boolean; } = {}): void
✓ patchValue(value: { [key: string]: any; },
               options: { onlySelf?: boolean; emitEvent?: boolean; } = {}): void
✓ reset(value: any = {}, options: { onlySelf?: boolean; emitEvent?: boolean; } = {}): void
✓ getRawValue(): any
```

## FormGroup Example

```
this.loginForm = new FormGroup(
     {
       username : new FormControl('',
                                 [Validators.required, Validators.minLength(6)]),
       password : new FormControl('', Validators.required)
     }
)
```

## FormBuilder

- ❑ This provides a factory method to create FormGroup.
- ❑ It is easier to create a set of controls in single object rather than creating individual FormControl objects.

```
import { FormControl, FormGroup, FormBuilder, Validators} from '@angular/forms';
. . .
export class ReactiveLoginComponent {
  loginForm : FormGroup;
  constructor (private fb: FormBuilder ) {
    this.loginForm =  this.fb.group(
      {
        username : ["",Validators.compose
                           ([Validators.required, Validators.minLength(4)])],
        password : ["",Validators.required]
      }
    )
  }
}
```

## Reactive Form Template – Example Page 1

```
<h2>Login</h2>
<form [formGroup]="loginForm">
    <label for="username">Username</label> <br>
    <input type="text" id="username" formControlName="username">
    <span *ngIf="loginForm.controls.username.errors">
        <span *ngIf="loginForm.controls.username.errors?.required">
          Username must be entered!
        </span>
        <span *ngIf="loginForm.controls.username.errors?.minlength">
          Username must be >=  4 chars
        </span>
    </span>
```

## Reactive Form Template – Example Page 2

```
       <p></p>
        <label for="password">Password </label> <br>
        <input type="password" id="password" formControlName="password">
         <span *ngIf="loginForm.controls.password.errors">
            <span *ngIf="loginForm.controls.password.errors.required">
              Password must be entered!
            </span>
        </span>
        <p></p>
        <button>Login</button>
    </form>
```

# Custom Validation

- ❑ Custom validation is done by creating a function, which:
  - ✓ Takes a parameter of type FormControl or FormGroup (for validating multiple controls).
  - ✓ Returns null on success and an object on error.
- ❑ Provide this function name in place of validator.

```
validatePassword(control : FormControl) {
      if (control.value.indexOf("*") >= 0)
        return null;  // success
      else
        return { mustHaveStar : true };  // error
}
```

```
 this.loginForm = this.fb.group(
      {
        username: ["", [Validators.required, Validators.minLength(4)]],
        password: ["", [Validators.required, this.validatePassword]]
      }
 )
```

# Custom Validation -  Form

```
   <form (ngSubmit)="login()" [formGroup]="loginForm">
         . . .
         <label for="password">Password </label> <br>
         <input type="password" id="password" formControlName="password">
          <span *ngIf="loginForm.controls.password.errors && isSubmitted">
             <span *ngIf="loginForm.controls.password.errors.required">
                          Password must be entered!
             </span>
             <span *ngIf="loginForm.controls.password.errors.mustHaveStar">
                          Password must have a star (*)
             </span>
          </span>
          . . .
 </form>
```

# Making Ajax calls with HttpModule

- ❑ The Angular Http client communicates with the server using a familiar HTTP request/response protocol.
- ❑ The Http client is one of a family of services in the Angular HTTP library - @angular/common/http.
- ❑ Before you can use the Http client, you need to register it as a service provider with the dependency injection system.

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

@NgModule({
  imports: [
    BrowserModule,
    HttpClientModule
  ],
})
export class AppModule {
}
```

## Using Promise

```
import { Component } from '@angular/core';
import { Book } from './Book';
import { Http, Response } from '@angular/http';
import { Observable } from 'RxJS';
import 'rxjs/add/operator/toPromise';

@Component({
    selector: 'st-books',  templateUrl: 'books.component.html'
})
export class BooksPromiseComponent {
    books: Book[];
    constructor(private http: Http) {
    }
    ngOnInit() {
      this.http.get("assets/books.json")
        .toPromise()
        .then( response => this.books = response.json());
    }
}
```

## Observable

- ❑ Angular uses Observables to connect to backend servers.
- ❑ Observable is a concept introduced in a library called Reactive extensions – reactivex.io.
- ❑ An Observable represents an asynchronous data stream where data arrives asynchronously.
- ❑ An observer (subscriber or watcher) subscribes to an Observable.
- ❑ Observer reacts to whatever item or sequence of items the Observable emits.

## Important Terms

- ❑ A publisher creates an Observable instance that defined a subscriber function.
- ❑ Subscriber function is executed when a consumer calls **subscribe**() method.
- ❑ To begin receiving notifications, we need to call subscribe() method.
- ❑ We can call unsubscribe() method to stop receiving notification.
- ❑ Data published by an observable is known as a **stream.**

# TimeComponent - Displays running digital clock

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';
@Component({
selector: 'st-time',
template: `<h1>{{time}}</h1> <button (click)="stopTimer()">Stop Timer</button>
                             <button (click)="startTimer()">Start Timer</button>`})
export class TimeComponent {
  time : string;
  subscriber : any;

  timeObservable = Observable.create(
     function(observer) {
       const interval = setInterval(() => observer.next(new Date().toString()),1000);
       // returns process to clean up resources
       return () => clearInterval(interval); }
  );
```

# TimeComponent - Displays running digital clock

```
  stopTimer() {
        this.subscriber.unsubscribe();
  }
  startTimer() {
        this.subscriber = this.timeObservable.subscribe( val => this.time = val);
  }
}
```

## Observable Operators

- Observable provides set of operators where each operator performs a single operation.
- Most operators operate on an Observable and return an Observable.
- It is possible to chain result of one operator with another as most of the operators return an Observable.
- In order to use operator, we need to import the operator as follows:

```
import { map, finalize } from 'rxjs/operators'
```

# Observable Operators

| Operator | What it does? |
|---|---|
| filter | Emit only those items from an Observable that pass a predicate test |
| distinct | Suppress duplicate items emitted by an Observable |
| first | Emit only the first item, or the first item that meets a condition, from an Observable |
| last | Emit only the last item emitted by an Observable |
| skip | Suppress the first n items emitted by an Observable |
| skipLast | Suppress the last n items emitted by an Observable |
| take | Emit only the first n items emitted by an Observable |
| takeLast | Emit only the last n items emitted by an Observable |
| map | Transform the items emitted by an Observable by applying a function to each item |
| catch | Recover from an onError notification by continuing the sequence without error |
| do | Register an action to take upon a variety of Observable lifecycle events |
| defaultIfEmpty | Emit items from the source Observable, or a default item if the source Observable emits nothing |
| average | Calculate the average of numbers emitted by an Observable and emit this average |
| concat | Emit the emissions from two or more Observables without interleaving them |
| count | Count the number of items emitted by the source Observable and emit only this value |
| max | Determine, and emit, the maximum-valued item emitted by an Observable |
| min | Determine, and emit, the minimum-valued item emitted by an Observable |
| reduce | Apply a function to each item emitted by an Observable, sequentially, and emit the final value |
| sum | Calculate the sum of numbers emitted by an Observable and emit this sum |

# HttpClient class

- ❑ Http class is used to make AJAX calls using methods like get(), post(), delete() and put().
- ❑ In order to use HttpClient, we need to import it from @angular/common/http.
- ❑ Inject HttpClient into our component using DI mechanism in Angular, i.e. as a parameter to constructor.
- ❑ Use get() method of HttpClient class and provide URL of the endpoint.
- ❑ Return type of get() is Observable<any>.
- ❑ Client should consume the value coming from Observable by subscribing to Observable using subscribe() method.
- ❑ It is possible to perform operations on Observable using operators provided by RxJS library.
- ❑ Each code file in RxJS should add the operators it needs by importing from an RxJS library.
- ❑ Response object contains data coming from server that is already converted to JSON.

# HttpClient Class Methods

```
get(url: string, options?: RequestOptionsArgs) : Observable<any>

post(url: string, body: any, options?: RequestOptionsArgs) : Observable<any>

put(url: string, body: any, options?: RequestOptionsArgs) : Observable<any>

delete(url: string, options?: RequestOptionsArgs) : Observable<any>
```

# Subscribe Method

- ❑ **Subscribe** method is used to connect an Observer to an Observable.

```
subscribe(onNext, onError, onComplete)
```

**onNext**

- ❑ An Observable calls this method whenever the Observable emits an item.
- ❑ This method takes as a parameter the item emitted by the Observable.
- ❑ This may be called zero or more times.

**onError**

- ❑ An Observable calls this method to indicate that it has failed to generate the expected data or has encountered some other error.
- ❑ It will not make further calls to **onNext** or **onCompleted**.
- ❑ The onError method takes as its parameter an indication of what caused the error.

**onCompleted**

- ❑ An Observable calls this method after it has called **onNext** for the final time, if it has not encountered any errors.
- ❑ This doesn’t take any parameter.

## assets/Books.json

```json
[
    {
        "title": "Beginning Angular with TypeScript",
        "author": "Greg Lim",
        "price": 399
    },
    {
        "title": "Angular 2 Cookbook",
        "author": "Matt Frisbie",
        "price": 1199
    }
]
```

# list-books.component.ts

```
import { Component } from '@angular/core';
import { OnInit } from '@angular/core';
import { Book } from './Book';
import { HttpClient} from '@angular/common/http';

@Component({
    selector: 'st-books',
    templateUrl: 'app/http/list-books.component.html'
})
export class ListBooksComponent implements OnInit {
    books: Book[];
    constructor(private http: HttpClient) {
    }
    ngOnInit() {
       this.http.get<Book[]>("assets/books.json")
          .subscribe( resp => this.books = resp);
    }
}
```

# list-books.component.html

```
<html>
<head>
    <title>Books</title>
</head>
<body>
    <h1>Books</h1>
    <table border="1" style="width:100%">
        <tr>
            <th>Title</th>
            <th>Author</th>
            <th>Price</th>
        </tr>
        <tr *ngFor="let b of books">
            <td> {{ b.title }} </td>
            <td> {{ b.author }} </td>
            <td> {{ b.price }} </td>
        </tr>
    </table>
</body>
</html>
```

# Routing

- Routing allows users to move from one view to another view using different URLs in the client.
- The Angular router is an external, optional Angular NgModule called RouterModule.
- The router is a combination of multiple provided services (RouterModule), multiple directives (RouterOutlet, RouterLink, RouterLinkActive), and a configuration (Routes).
- Router can interpret a browser URL as an instruction to navigate to a client-generated view.
- It can pass optional parameters along to view component.

# Routing Components

| Router Part | Meaning |
|---|---|
| Router | Displays the application component for the active URL. Manages navigation from one component to the next. |
| RouterModule | A separate Angular module 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. |

# Routing Components

| | |
|---|---|
| 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. |

## Step 1 - <base href>

- ❑ In index.html place <base href> as the first child of head section.
- ❑ It tells the router how to compose navigation URLs.

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

## Step 2 - Import Router

- ❑ The Angular Router is an optional service that presents a particular component view for a given URL.
- ❑ It is not part of the Angular core.
- ❑ It is in its own library package, @angular/router.
- ❑ Import what you need from it as you would from any other Angular package.

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

# Step 3 - Route Configuration

- ❑ A routed Angular application has one singleton instance of the Router service.
- ❑ When the browser's URL changes, that router looks for a corresponding Route from which it can determine the component to display.
- ❑ A router has no routes until you configure it.
- ❑ We need to pass the array of Routes objects to RouterModule.forRoot() to configure the router.
- ❑ Each Route maps a URL path to a component. There are no leading slashes in the path. The router parses and builds the final URL for you, allowing you to use both relative and absolute paths when navigating between application views.

```
@NgModule(
{ declarations: [AuthorsListComponent, AuthorDetailsComponent, MainComponent],
  imports: [RouterModule.forRoot(appRoutes)],
  providers: [],
  bootstrap: [MainComponent]
})
export class AppModule {
}
```

# Step 3 - Route Configuration

```
const appRoutes: Routes = [
 { path: 'list', component: AuthorsListComponent },
 { path: 'details/:id', component:  AuthorDetailsComponent },
 { path: '', component : HomeComponent, pathMatch : 'full'},
 { path: '**', component: HomeComponent}
];
```

- The :id in the third route is a token for a route parameter. In a URL such as /details/1, "1" is the value of the id parameter.
- The data property in route is a place to store arbitrary data associated with this specific route. The data property is accessible within each activated route. Use it to store items such as page titles, breadcrumb text, and other read-only, static data.
- The empty path represents the default path for the application, the place to go when the path in the URL is empty.

# Step 3 - Route Configuration

- ❑ The ** path in the last route is a wildcard. The router will select this route if the requested URL doesn't match any paths for routes defined earlier in the configuration.
- ❑ The order of the routes in the configuration matters and this is by design. The router uses a first-match wins strategy when matching routes, so more specific routes should be placed above less specific routes.
- ❑ A redirect route requires a pathMatch property to tell the router how to match a URL to the path of a route. Value full means the entire URL must match the given string.

## Step 4 - Place <router-outlet>

Given this configuration, when the browser URL for this application becomes /list, the router matches that URL to the route path /list and displays AuthorsListComponent in RouterOutlet that you've placed in the host view's HTML.

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

# Step 5 - Use RouterLink directive

- Most of the time you navigate as a result of some user action such as the click of an anchor tag.
- The RouterLink directives on the anchor tags give the navigation.
- The RouterLinkActive directive on each anchor tag helps visually distinguish the anchor for the currently selected "active" route.
- The router adds the active CSS class to the element when the associated RouterLink becomes active.

```
[<a routerLink="/home" routerLinkActive="active">Home</a>]
[<a routerLink="/list" routerLinkActive="active">List Authors</a>]
```

## Step 5 - Use Router Parameters

- ❑ Route parameters are used to pass data to a page.
- ❑ Parameters are prefixed with : (colon) in URL.
- ❑ For example, in URL /details/:id, id is route parameter.
- ❑ We can have multiple parameters each separated with /.
- ❑ For example, if we have a URL /author/:id/:subject then actual URL is /author/10/angular.
- ❑ You need to pass values to parameters using **routerLink** directive as shown below:

```
<a [routerLink]="['/details', author.id]"> {{author.name}}</a>
```

# Step 5 - Read Router Parameters

- In order to receive values of route parameters we need to inject ActivatedRoute object into our component and then read route parameter using params collection of snapshot of route.
- It is also possible to subscribe to params observable.

```
export class AuthorDetailsComponent {

  constructor(private route: ActivatedRoute, private router : Router) {
  }

  ngOnInit(): void {
    this.route.params.subscribe( params => {
      this.id =  params["id"];
    });

    // alternatively we can read param from snapshot
    this.id = this.route.snapshot.params["id"];
  }
  // rest of code
}
```

# Programmatic Navigation

- ❑ When we need to go to a page directly programmatically we use Router object.
- ❑ Inject Router object into component and then call navigate method with required url.

```
constructor(private router : Router) {
}

list(){
   this.router.navigate( ['list']);
}
```

## Route Guard

- ❑ A Route Guard is used to check whether navigation to route or from route is allowed.
- ❑ A guard can navigate elsewhere, cancelling the current navigation.
- ❑ A guard returns true or false. True will allow navigation process to continue and false stops navigation.
- ❑ The following are available guards :
    - ✓ CanActivate to mediate navigation to a route.
    - ✓ CanActivateChild to mediate navigation to a child route.
    - ✓ CanDeactivate to mediate navigation away from the current route.
    - ✓ Resolve to perform route data retrieval before route activation.
    - ✓ CanLoad to mediate navigation to a feature module loaded asynchronously.

## Step 1 – Create Route Guard

- ❑ Create a class that implements interface related to Route Guard.
- ❑ Use **@Injectable()** decorator with class.
- ❑ Define method declared in interface and return either true or false from that method.

```
import { Injectable }     from '@angular/core';
import { CanActivate }    from '@angular/router';

@Injectable()
export class AuthGuard implements CanActivate {
  canActivate() {
   If (ok_to_allow_user)
        return true;
   else {
        // take action like redirecting user to login page
        return false;
   }
  }
}
```

## Step 2 – Associate Guard with Route

```
const appRoutes: Routes = [
   { path: 'admin',
     component: AdminComponent,
     canActivate: [AuthGuard]
   },
   …
];
```

## Step 3 – Declare Route Guard in Providers Array

```
@NgModule({
declarations:
   [ . . .],
imports:
   [BrowserModule, FormsModule, HttpModule,
    RouterModule.forRoot(appRoutes)],
providers:
   [AuthGuard],
bootstrap:
   [MainComponent]
})
export class AppModule { }
```

## Deployment

- ❑ Making Angular application available to a server running remotely.
- ❑ When run in production, we need to optimize application size and performance.
- ❑ Not all files in development are needed in production environment.

# Tools

- ❑ Tools and techniques are used to reduce number of requests and size of responses.
- ❑ Goal is to reduce payload.

| | |
|---|---|
| **Ahead-of-Time (AOT) Compilation** | Pre-compiles Angular component templates. |
| **Bundling** | Concatenates modules into a single file (bundle). |
| **Inlining** | Pulls template html and css into the components. |
| **Minification** | Removes excess whitespace, comments, and optional tokens. |
| **Uglification** | Rewrites code to use short, cryptic variables and function names. |
| **Dead code elimination (DCE)** | Removes unreferenced modules and unused code. |
| **Pruned libraries** | Drops unused libraries and trims others down to the features you need. |

## Ahead-of-Time (AOT) compilation

- The Angular *Ahead-of-Time* compiler pre-compiles application components and their templates during the build process.
- Apps compiled with AOT launch faster for following reasons:
  - ✓ Application components execute immediately, without client-side compilation.
  - ✓ Templates are embedded as code within their components, so there is no client-side request for template files.
  - ✓ You don't download the Angular compiler, which is pretty big on its own.
  - ✓ The compiler discards unused Angular directives that a tree-shaking tool can then exclude.

# Build Project

- ❑ Use ng build to build project for deployment.
- ❑ Artifacts built by ng build are stored in dist folder of the project.
- ❑ Must be run from the folder in which angular cli placed project artifacts.
- ❑ Build can be either for production (--prod) or development  (--dev) environment.
- ❑ All builds make use of bundling and limited tree-shaking, while --prod builds also run limited dead code elimination via UglifyJS.

```
ng build --prod
```

| Flag | --dev | --prod |
|---|---|---|
| --aot | false | true |
| --environment | dev | prod |
| --output-hashing | media | all |
| --sourcemaps | true | false |
| --extract-css | false | true |

## Tree-shaking

- ❑ Drops unused pieces of code from module.
- ❑ Introduced in ES6 modules.
- ❑ It is possible to analyze module definition in a static way and decide which parts of code are used and which are not, known as live-code import.
- ❑ Made popular by Rich Harri’s Rollup project, which aims at creating smaller bundles from modules.
- ❑ In the following example,  tree-shaking will add only fun1 from module and doesn’t include other functions that are not used.

```
import { fun1 } from 'some_module';

fun1();
```

# Dead code elimination (DCE)

- ❑ Removes code that is not used in the program.
- ❑ It removes code that is never executed and variables that are never used.
- ❑ The goal is to shrink the size of production code and reduce execution time.

```
const x = 1;

if (false) {
  console.log(x);
}
```