Life Cycle Hooks

- Angular calls lifecycle hook methods on directives and components as it creates, changes, and destroys them.
- Angular manages lifecycle of a components and directives.
- A directive has the same set of lifecycle hooks, minus the hooks that are specific to component content and views.
- Angular offers lifecycle hooks that provide the ability to act when they occur.

Hook	Purpose and Timing
ngOnChanges()	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()	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()	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()	Respond after Angular projects external content into the component's view. Called once after the first ngDoCheck(). A component-only hook.
ngAfterContentChecked()	Respond after Angular checks the content projected into the component. Called after the ngAfterContentInit() and every subsequent ngDoCheck(). A component-only hook.

ngAfterViewInit()	Respond after Angular initializes the component's views and child views. Called once after the first ngAfterContentChecked(). A component-only hook.
ngAfterViewChecked()	Respond after Angular checks the component's views and child views. Called after the ngAfterViewInit and every subsequent ngAfterContentChecked(). A component-only hook.
ngOnDestroy	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.

Angular inspects directive and component classes and calls the hook methods if they are defined.

Angular finds and calls methods like **ngOnInit**(), with or without the interfaces like **onInit**.

lifecycle.component.ts

```
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 Destoryed!");
        }
}
```

Dependency Injection

- It is a coding pattern in which a class receives its dependencies from external sources rather than creating them itself.
- 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.

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.
- 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.

Providers Property

- A service is to be registered before it is used.
- A service provider 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);
        }
}
```

uselog.component.html

```
<html>
<head>
</head>
<body>
        <h1>Using Log Service</h1> <input
        type="text" #message> <button
(click)="logMessage(message.value)">Log</button>
</body>
</html>
```

Optional dependencies

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);
    }
}
```

Forms

- An Angular form coordinates a set of data-bound user controls, tracks changes, validates input, and presents errors.
- Angular support two types of forms – Template-driven Forms and Reactive (model-driven) forms.

Template-driven Forms

- We build forms 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
- Angular automatically creates and attaches an **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. You can leverage those class names to change the appearance of the control.

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
    [(ngModel)]="cust.name" name="name" #spy>
<br>{{spy.className}}
```

Submitting form

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

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

<button type="submit" class="btn btn-success"
[disabled]="!heroForm.form.valid">Submit</button>
```

Customer.ts

```
export class Customer
{
    constructor(
        public id: number, public name:
        string, public email: string,
        public mobile : string
    ) {       }
}
```

customer.component.ts

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

@Component({
    selector: 'customer',
    templateUrl: './app/customer.component.html' })
export class CustomerComponent { model = new
    Customer(1,"","",""); submitted = false;
```

```
    resetForm()
    {
        this.model = new Customer(1,"","","");  this.submitted =
        false;
    }

    onSubmit() { this.submitted = true;
          console.log( JSON.stringify(this.model));
    }
}
```

customer.component.html

```
<html>

<body>
    <h1>Customer</h1>
    <form (ngSubmit)="onSubmit()" #custForm="ngForm"> Name <br>
        <input type="text" id="name" [(ngModel)]="model.name" name="name"
required #name="ngModel">
        <span [hidden]="name.valid || name.pristine">Name is required!</span>
        <p></p>
        Email<br>
        <input type="text" id="email"
[(ngModel)]="model.email" name="email">
        <p></p>
        <button type="submit"
[disabled]="!custForm.form.valid">Submit</button>
        <button type="reset" (click)="resetForm()"
*ngIf="submitted">Reset</button>
    </form>
</body>
</html>
```

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.
- Create template
- Import ReactiveFormsModule

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}, {onlySelf, emitEvent}?: {onlySelf?: boolean,
emitEvent?: boolean}) : void
```

patchValue(value: {[key: string]: any}, {onlySelf, emitEvent}?: {onlySelf?: boolean, emitEvent?: boolean}) : void

reset(value?: any, {onlySelf, emitEvent}?: {onlySelf?: boolean, emitEvent?: boolean}) : void

getRawValue() : any

Note: But unlike setValue, patchValue cannot check for missing control values and does not throw helpful errors.

Each **FormControl** has the following properties.

Property		Description
myControl.value		Value of a FormControl.
myControl.status		Validity of a FormControl. Possible values: VALID, INVALID, PENDING, or DISABLED.
myControl.pristine		True if the user has not changed the value in the UI. Its opposite is myControl.dirty.
myControl.untouched		True if the control user has not yet entered the HTML control and triggered its blur event. Its opposite is myControl.touched.

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.

```
export class FormComponent { myForm:
    FormGroup;

    constructor(private fb: FormBuilder) { this.myForm =
          this.fb.group({ name: '',
            email : '', … });
    }
}
```

It is possible to include Validators at the time of creating FormControl objects.

```
this.myForm = this.fb.group({ name: ['',
    Validators.required ],
});
```

register.component.ts

```
import { Component } from '@angular/core';
import { FormControl, FormGroup, FormBuilder, Validators} from '@angular/forms';
@Component({
    selector: 'st-register',
    templateUrl: '/app/reactive/register.component.html' })
export class RegisterComponent {
      /* -- We can create a FormGroup in this way also registerForm = new
    FormGroup ({
          email : new FormControl("Aspire@yahoo.com"), mobile : new
          FormControl("9059057000")
```

```
    });
    */
    registerForm : FormGroup;
    constructor (private fb: FormBuilder ) { this.registerForm =
        this.fb.group(
            {
                email : ["", Validators.required], mobile :
                ["",Validators.required]
            }
        )
    }
    onSave(): void{
        console.log( this.registerForm); this.registerForm.setValue( {
        "email" : ""});
    }
}
```

register.component.html

```
<h2>Registraion</h2>
<form [formGroup]="registerForm" novalidate> Email Address
        <br>
        <input type="text" formControlName="email" required> <p></p>
        Mobile Number <br>
        <input type="text" formControlName="mobile" required> <p></p>
        <button (click)="onSave()"
[disabled]="!registerForm.valid">Submit </button>
        <p></p>
        Value : {{ registerForm.controls.email.value }} <p></p>
        Pristine : {{ registerForm.controls.email.pristine }} <p></p>
        Touched : {{ registerForm.controls.email.touched }} <br> {{
        registerForm.get("email").value }}
</form>
```

Making AJAX calls

- 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/http.
- Before you can use the Http client, you need to register it as a service provider with the dependency injection system.

```
import { HttpModule } from '@angular/http';

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

Observable

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

Observable Operators

- Observable provides set of operators where each operator performs a single operation.
- It is possible to chain result of one operator with another as most of the operators return an Observable.
- For more information about operators refer to reactivex.io.
- In order to use operator, we need to import the operator as follows:

```
import 'rxjs/add/operator/map'; import
'rxjs/add/operator/takeLast';
```

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	Calculates the average of numbers emitted by an Observable and emits 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

Http class

- Http class is used to make AJAX calls using methods like get(), post(), delete() and put().
- In order to use Http, we need to import it from @angular/http
- Inject Http into our component using DI mechanism in Angular, i.e. as a parameter to constructor.
- Use get() method of Http class and provide URL of the endpoint.
- Return type of get() is Observable<Response>
- Client should consume the value coming from Observable by subscribing to Observable.
- 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 and we need to convert into JSON using json() method.

```
class Http {

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

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

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

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

The following example shows how to make an Ajax request using Http and consume data.

Books.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 { Http, Response } from '@angular/http'; import
'rxjs/add/operator/map';

@Component({
	selector: 'st-books',
	templateUrl: 'app/http/list-books.component.html'
})
export class ListBooksComponent implements OnInit { books: Book[];
	constructor(private http: Http) {

	}
	ngOnInit() { this.http.get("/app/http/books.json")
	    .map(resp => resp.json()) // convert to JSON
	                                .subscribe( resp => this.books = <Book[]> 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.

The following steps are need to use routing.

<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="/">
```

Router imports

- 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';
```

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.

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

- We need to pass the above array 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.
- The **:id** in the second 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 the third 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.
- 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.

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 after a RouterOutlet that you've placed in the host view's HTML.

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

Router links

- 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>]
```

Router state

- After the end of each successful navigation lifecycle, the router builds a tree of **ActivatedRoute** objects that make up the current state of the router. You can access the current RouterState from anywhere in the application using the Router service and the routerState property.
- Each ActivatedRoute in the RouterState provides methods to traverse up and down the route tree to get information from parent, child and sibling routes.

Route 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 **routeLink** directive as shown below:

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

Reading Route 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.

```
id = this.route.snapshot.params["id"];

// read parameter using Observable
this.route.params.subscribe(
        params => this.id = params["id"];
);
```

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.
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.

Imperative or 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.

```
this.router.navigate( ['list'])
```

app/routing/Author.ts

```
export class Author { constructor(public id:
      number,
                        public name: string, public email: string,
                        public imageUrl: string) { }
```

```
        public static getAuthors(): Author[] { return [
            new Author(1, "Herbert Schildt", "herbert@gmail.com",
                "herbert.jpg"),
            new Author(2, "Stephen Walther",
                "stephen@msn.com", "walther.jpg"),
            new Author(3, "Aspire Pragada", "Aspirepragada@gmail.com",
                "Aspire.jpg")];
        }
}
```

app/app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser'; import {
RouterModule, Routes } from '@angular/router'; import { AuthorsListComponent }
from './routing/authors-list.component';
import { AuthorDetailsComponent } from './routing/author-details.component';
import { MainComponent } from './routing/main.component'; import {
HomeComponent } from './routing/home.component'; const appRoutes: Routes = [
    { path: 'home', component: HomeComponent },
    { path: 'list', component: AuthorsListComponent }, { path: 'details/:id',
    component:
AuthorDetailsComponent },
    { path: '', component : HomeComponent}, { path: '**',
    component: HomeComponent }
];

@NgModule({
    imports: [RouterModule.forRoot(appRoutes),
BrowserModule, FormsModule,HttpModule],
    declarations: [HomeComponent, AuthorDetailsComponent,
AuthorsListComponent, MainComponent],
    bootstrap: [MainComponent] })
```

```
export class AppModule { }
```

app/routing/main.component.ts

```
import { Component } from '@angular/core'; @Component({
    selector: 'authors',
    templateUrl: 'app/routing/main.component.html' })
export class MainComponent                  {

}
```

```
<html>
<body>
    <h2>Authors</h2>
    [<a routerLink="/home"
routerLinkActive="active">Home</a>]
    [<a routerLink="/list" routerLinkActive="active">List Authors</a>]
    <p></p> <router-outlet></router-outlet>
  </body>

</html>
```

app/routing/home.component.ts

```
import { Component } from '@angular/core'; @Component({
    selector: 'authors',
    template :`<h2>About Authors </h2>
                        This application show details of Authors.`
})
export class HomeComponent                  {
}
```

app/routing/authors-list.component.ts

```
import { Component } from '@angular/core'; import { Author
} from "./Author"; @Component({
    selector: 'authors-list',
    templateUrl : 'app/routing/authors-list.component.html' })
export class AuthorsListComponent { authors : Author
          [] ; ngOnInit() {
              this.authors = Author.getAuthors();
          }
}
```

App/routing/authors-list.component.html

```
<html>
<body>
    <h2>Authors</h2>
    <ul>
        <li *ngFor="let author of authors">
              <a [routerLink]="['/details', author.id]"> {{author.name}} </a>
        </li>
    </ul>
  </body>
</html>
```

App/routing/author-details.component.ts

```
import { Component, Input } from '@angular/core'; import { Author }
from './Author';
import { ActivatedRoute, Params, Router } from '@angular/router';
@Component({
    selector: 'author-details',
    templateUrl: 'app/routing/author-details.component.html' })
```

```
export class AuthorDetailsComponent { author:
    Author;
    id : number;

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

  ngOnInit(): void {
            this.id = this.route.snapshot.params["id"]; this.author =
            this.getAuthor(this.id);
    }


    getAuthor(id : number) : Author
    {
          for (var a of Author.getAuthors()) { if (a.id == id) {
                return a;
            }
        }
    }
      back() {
        this.router.navigate( ['list']);
    }
}
```

App/routing/author-details.component.html

```
<div *ngIf="author">
    <h2>{{author.name}}</h2>
    <h3>{{author.email}}</h3>
    <img [src]="'/app/routing/' + author.imageUrl"> </div>

<button (click)="back()">Back</button>
```