Learn Angular

Angular Setup & Config

1. The first version of Angular is called **AngularJS**. After that, it is referred to simply as **Angular**.
2. Typescript provides errors and warnings during compile time that are easier to understand. Typescript is a **superset** of Javascript.
3. Install Angular with npm → `npm install -g @angular/cli@latest`
4. Creating app → `ng new my-app`
5. Create app in existing directory → `ng new existingDir --directory ./`
6. Run app → `ng serve`

Project Setup using Bootstrap for Styling

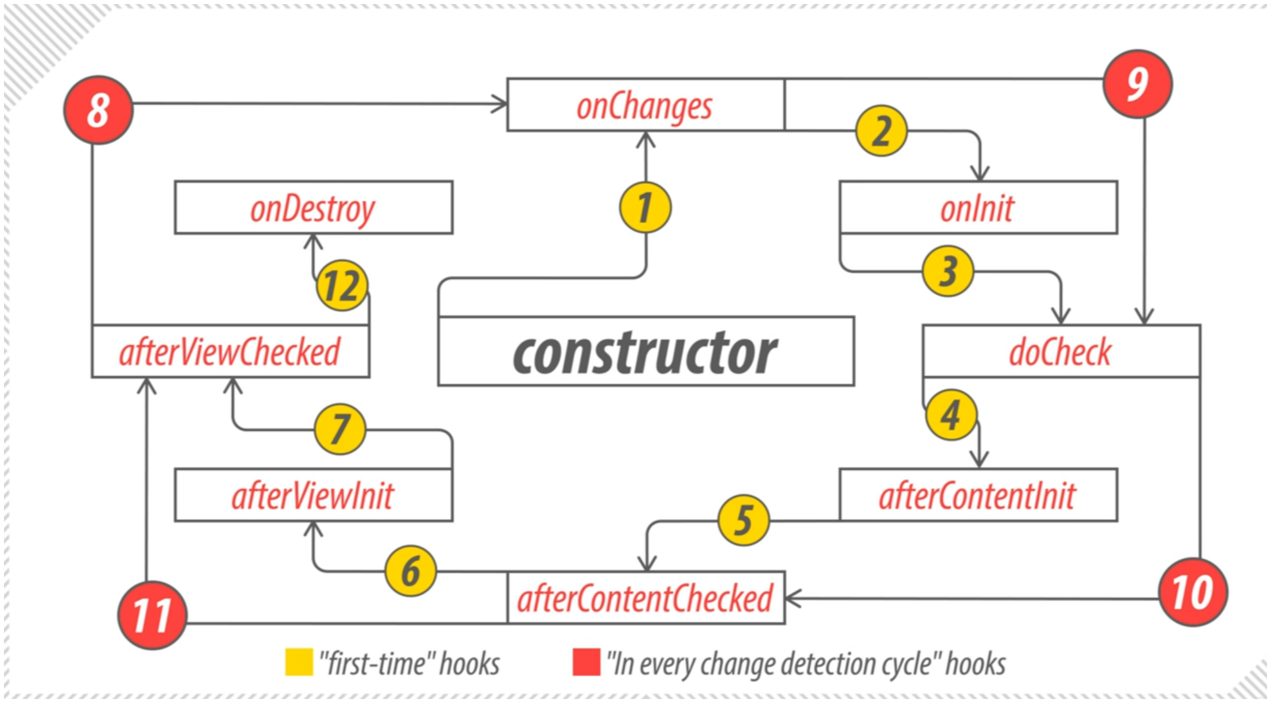
1. Run the command - `npm install -save bootstrap`
2. Add the **Bootstrap** reference link to the **styles** array under the **\`architect>build\`** section of the angular.json file.
   Link → "node_modules/bootstrap/dist/css/bootstrap.min.css"

Lifecycle Method

1. **OnChanges()**: Called after a bound input property changes. **'Bound'** means properties decorated with **@Input()**. It receives arguments and is also invoked when a component is initialized.
2. **OnInit()**: Called once the component is initialized and will run after the constructor. It used to perform a custom change detection & responding to the changes in a component.
3. **DoCheck()**: Called during every change detection run and will be executed frequently.
4. **AfterContentInit()**: Called after content (**ng-content**) has been projected into view.

S M HEMEL
Senior Software Engineer
Email: smhemel.eu@gmail.com

5. **AfterContentChecked()**: Called every time the projected content has been checked.
6. **AfterViewInit()**: Called after the component's view (with child views) has been initialized.
7. **AfterViewChecked()**: Called every time the view (with child view) has been checked.
8. **OnDestroy()**: Called once the component is destroyed.



Angular Basics

Component

1. It is best practice to create all components in the 'src' directory, and each component should have its own dedicated directory.
2. If we create a component manually, we need to create three different files. (e.g. xyz.component.ts, xyz.component.html, xyz.component.scss)
3. General Structure of a Component(**demo.component.ts**) →

```
----------------------- demo.component.ts -----------------------
import { Component } from '@angular/core';
@Component({
    selector: 'app-demo',
    // template: '<p>Hello</p>' // We can write HTML code here
    templateUrl: './demo.component.html',
    styleUrls: ['./demo.component.scss'],
    // styles: [`h3{color: white}`] // We can write CSS code here
})
export class DemoComponent {}
```

[Note: We cannot use `**template**` and `**templateUrl**` simultaneously; the same rule applies to `**styles**` and `**styleUrls**`.**]**

4. To use a component in a project, ensure that it is added to the **declarations** array in either the app.module.ts file or a dedicated module.ts file.
5. Create a component using CLI command →
   `ng generate component component_name`
   Shortly, → **`ng g c component_name`**
   Note: All components will be created inside the **app** directory.
6. When creating a component using the CLI command, add **`--skip-tests true`** to avoid generating the test file.
7. An Angular component has three types of selectors. Component use in HTML by the help of selector.
   CSS Selector → **`selector: 'app-xyz'`**

```
<app-xyz></app-xyz>
```

   Attribute Selector → **`selector: '[app-xyz]'`**

```
<div app-xyz></div>
```

   Select by Class → **`selector: '.app-xyz'`**

```
<div class="app-xyz"></div>
```

Data Binding

1. String interpolation → **`{{data}}`**

```
<p>{{ 'Server' }} with ID {{ severId }} is {{ getServerStatus() }}</p>
```

2. Property binding → **`[property]="data"`**

```
<button class="btn btn-primary" [disabled]="allowServer"></button>
<p [innerText]="allowServer"></p>
```

3. Event binding → `(event)="demoFunction()"`

```
<button [disabled]="allowServer" (click)="onCreateServer()"></button>
<p>{{ serverCreationStatus }}</p>
```

4. To enable two-way binding, the **ngModel** directive must be enabled by adding **FormsModule** to the **imports** array in the app.module.ts file.

```
<input type="text" class="form-control" [(ngModel)]="serverName" />
<p>{{ serverName }}</p>
```

5. Triggering an event from another component is called Custom Events binding →

```
------------------------ app.component.html ----------------------
<div class="container">
    <app-demo
        (serverCreated)="onServerAdded($event)"
        (blueprintCreated)="OnblueprintCreated($event)">
    </app-demo>
</div>
------------------------- app.component.ts -----------------------
export class AppComponent {
    onServerAdded(serverData: {name: string, content: string}) {}
    onblueprintAdded(serverData: {name: string, content: string}) {}
}
------------------------ demo.component.ts -----------------------
export class DemoComponent {
    @Output() serverCreated = new EventEmitter<data_type>();
    @Output() blueprintCreated = new EventEmitter<data_type>();
// Added parenthesis end of the event emitter to call its constructor to create a new event emitter object which is stored in the serverCreated and blueprintCreated.
    onAddedServer() {
        this.serverCreated.emit({ name: "", content: "" });
    }
}
```

[Note: Add parentheses at the end of EventEmitter to call its constructor and create a new EventEmitter object, which is now stored in the serverCreated and blueprintCreated variables.**]**

S M HEMEL
Senior Software Engineer
Email: smhemel.eu@gmail.com

6. Pass data to another component by binding to custom properties →

```
// Pass data from parent component
<app-child [srvElement]="data"> </app-child>
—- OR —-
<app-child [element]="data"> </app-child>

// Receive data in the child component as an element
@Input('srvElement') element: data_type
—- OR —-
@Input() element: data_type
```

Local Reference in Templates

1. In situations where two-way binding with **ngModel** is unnecessary, such as with input fields or other cases requiring interaction with an HTML tag or DOM element, the Local Reference technique can be used.

```
------------------------ app.component.ts -----------------------
<div class="container">
    <input type="text" class="form-control" #serverNameInput>
    <button (click)="onServerCreate(serverNameInput)"></button>
</div>
----------------------- demo.component.ts -----------------------
onServerCreate(nameInput: HTMLInputElement) { }
```

2. We can directly access a local reference variable in the **TypeScript** file using **@ViewChild()**.

```
export class DemoComponent {
   @ViewChild('serverNameInput') nameInput: ElementRef;

   onAddServer() {
       const name = this.nameInput.nativeElement.value;
   }
}
```

3. To access a variable declared with **@ViewChild()** inside the **ngOnInit()** lifecycle method, we must include **{static: true}** as an argument.

```
@ViewChild('serverNameInput', {static: true}) element: ElementRef;
```

4. It is strongly recommended not to manually change the value of a local reference element.

S M HEMEL
Senior Software Engineer
Email: smhemel.eu@gmail.com

5. If we attempt to access a template variable in the TypeScript file before the **ngAfterViewInit()**, we will get **undefined** or encounter an error.
6. When utilizing a template variable inside the **<ng-content>** in the HTML file, which is passed from the parent component, we use **@ContentChild()** similarly to **@ViewChild()**. In this case, **@ViewChild()** does not work as expected.

```
--------------------- control.component.ts ---------------------
export class ControlComponent {
  @ContentChild('input') private control?: ElementRef<HTMLInputElement |
HTMLTextAreaElement>;

  onCLick() {
    console.log(this.control);
  }
}
--------------------- control.component.html --------------------
<label>{{ label }}</label>
<ng-content select="input, textarea" />
--------------------- parent.component.html ---------------------
<form (ngSubmit)="onSubmit()" #form>
  <app-control label="Title">
    <input name="title" id="title" #input />
  </app-control>

  <app-control label="Request">
    <textarea name="request" id="request" rows="3" #textInput #input></textarea>
  </app-control>
</form>
```

Directive

1. Directives are instructions for the DOM. To create a directive, use the command → **`ng g d directive_name`**
2. Angular has two types of directives.
   a) **Structural directive**

```
<p *ngIf="server; else noServer">Server is available!</p>
<ng-template #noServer>
    <p>No Server!</p>
</ng-template>
```

b) **Attribute directive.**

In an HTML tag, we can use only one structural directive, while multiple attribute directives can be applied. It's important to note that attribute directives do not add or remove elements.

[Note: After creating a custom attribute directive, it is necessary to add the directive to the declarations array in the app.module.ts or specific module.ts file.**]**

3. Dynamically styling and binding property into a directive →

```
<p [ngStyle]="{backgroundColor: getColor()}"
   [ngClass]="{online: onlineStatus === 'online'}">
  Hello
</p>
```

4. The general structure of a directive component →

```
import { Directive } from '@angular/core';
@Directive({
    selector: '[textTurnGreen]'
})
export class TextGreenDirective {}
```

5. **ngFor** structural directive →

```
<div
    *ngFor="let paint of paints; let i = index"
    [ngStyle]="{backgroundColor: i >= 5 ? 'blue' : 'red'}"
    [ngClass]="{'white-text': i < 5}"
>
    {{ paint }}
</div>
```

6. Create a basic attribute directive →

```
@Directive({
    selector: '[appBasicHighlights]'
})
export class BasicHighlightDirective implements OnInit {
    constructor(private elementRef: ElementRef) { }

    ngOnInit() {
        this.elementRef.nativeElement.style.backgroundColor = 'blue';
    }
}
```

S M HEMEL
Senior Software Engineer
Email: smhemel.eu@gmail.com

7. Using Renderer to build better attribute directives. Directly accessing an element is not considered a good practice, which is why Angular offers a better approach: **Renderer2**. Learn more about Renderer2.

```
@Directive({ selector: '[appBetterHighlights]' })
export class BasicHighlightsDirective implements OnInit {
  constructor(private elRef: ElementRef, private renderer: Renderer2) {}

  ngOnInit() {
    this.renderer.setStyle(this.elRef.nativeElement, 'background-color', 'blue');
  }
}
```

8. **@HostListener()** is a convenient way to listen for events on a DOM element.

```
@HostListener('mouseenter') mouseover(eventData: Event) {
  this.renderer.setStyle(
      this.elRef.nativeElement, 'background-color', 'blue', false, false
  );
}
```

9. **@HostBinding()** is more efficient than **renderer2** for manipulating DOM into directives.

```
export class BasicHighlightsDirective implements OnInit {
    @HostBinding('style.backgroundColor') bgColor: string = 'red';
    constructor(private elRef: ElementRef) { }

    @HostListener('mouseenter') mouseover(eventData: Event) {
        this.bgColor = 'blue';
    }
}
```

10. Custom property binding in a directive involves using the **@Input()** variable within the directive component, allowing us to pass a value through property binding.

```
----------------- better-highlights.directive.ts -----------------
export class BetterHighlightsDirective implements OnInit {
    @Input() defaultColor: string = '';
    @Input() highlightColor: string = '';
    @HostBinding('style.backgroundColor') bgColor: string;

    constructor(private elRef: ElementRef) { }
    ngOnInit() {
        this.bgColor = this.defaultColor;
    }
    @HostListener('mouseenter') mouseover(eventData: Event) {
        this.bgColor = this.highlightColor;
    }
}
---------------- HTML file where we use the directive ----------------
<p
   appBetterHightlights
   [defaultColor]="'mouseenter'" [highlightColor]="'mouseenter'"
>
</p>
```

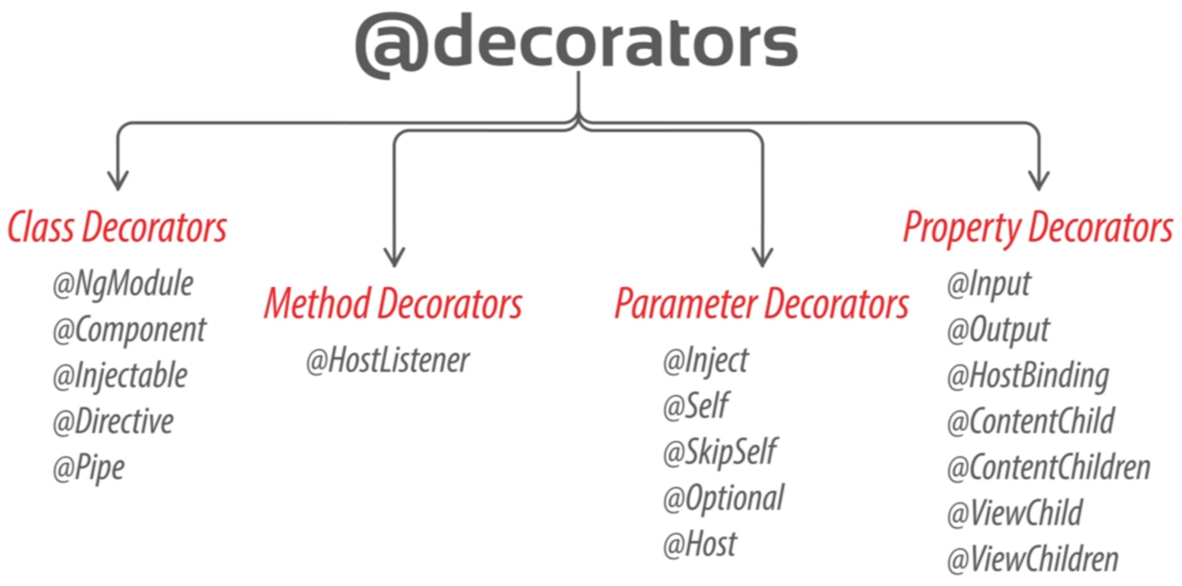
11. Custom Structural directive →

```
---------------------- unless.directive.ts ----------------------
@Directive({
    selector: '[appUnless]'
})
export class UnlessDirective {
    @Input() set appUnless(condition: boolean) {
        if (!condition) {
            this.vcRef.createEmbeddedView(this.templateRef)
        } else {
            this.vcRef.clear();
        }
    }
    constructor(
       private templateRef: TemplateRef<any>,
       private vcRef: ViewControllerRef
    ) {}
}
---------------- HTML file where we use the directive ----------------
<div *appUnless="onlyOdd"></div>
```

12. Understanding ngSwitch →

```
<div [ngSwitch]="value">
    <p *ngSwitchCase="5">Value is 5</p>
    <p *ngSwitchDefault>Value is default</p>
</div>
```

Decorator



1. **NgModule** → It marks the class as a module and provides a set of properties which are used for providing metadata.
2. **Component** → It marks the class as an Angular class and provides necessary metadata that determines how the component should be processed during runtime.

Styling

1. If we want to share a component's **CSS/SCSS** styles with all other components, we need to set the **View Encapsulation** value to '**None**' on the corresponding TypeScript file.

```
@Component({
    selector: 'app-xyz',
    templateUrl: './xyz.component.html',
    styleUrls: ['./xyz.component.scss'],
    Encapsulation: ViewEncapsulation.None -OR- ViewEncapsulation.ShadowDom
})
```

S M HEMEL
Senior Software Engineer
Email: smhemel.eu@gmail.com

Services & Dependency Injection

A service is fundamentally a function defined as a piece of code or logic used to perform a specific task and can be shared across different components in the application. We can create a service by running the command: → **`ng g s SERVICE_NAME`**.

1. Inject service into a component →

```
@Component({
    selector: 'app-xyz',
    templateUrl: './xyz.component.html',
    providers: [DemoService]
})
export class AccountComponent {
    constructor(private demoService: DemoService) { }
}
--------------------------------- OR ---------------------------------
export class AccountComponent {
    private demoService?: DemoService;
    constructor() { this.demoService = inject(DemoService); }
}
```

2. **Hierarchical Injector:** If we add a service to a component's **providers'** array, Angular creates an instance of the service for that component. This instance is then shared among its child components and their descendants.
   → Angular's dependency injector is hierarchical; therefore, if we add a service to a component's **providers'** array, it will not be accessible from its upper-level components.
   → To make a service accessible to all components, we should add it to the **providers'** array in the app.module.ts file.

3. Inject Services into Services →

```
@Injectable()
export class AccountService {
    constructor(private demoService: DemoService) { }
}
```

[Note: If we want to inject one or more services into another service, we need to add the **@Injectable()** decorator at the top of the service file. This is necessary to make the service injectable.**]**

4. Communicating two components using a service is called cross-component communication. Let's see an example:

```
------------------------- account.service.ts -------------------------
export class AccountService {
    statusUpdated = new EventEmitter<string>();
}
------------------------ account.component.ts ------------------------
// From where we send data to another component
export class AccountComponent {
    constructor(private accountService: AccountService) {}
    sendData() {
        this.accountService.statusUpdated.emit(_DATA_);
    }
}
------------------------- user.component.ts -------------------------
// Which component we will receive data
export class UserComponent {
    constructor(private accountService: AccountService) {
        this.service.statusUpdated
              .subscribe((status: string) => { // Our logical code });
    }
}
```

5. Angular offers a more streamlined approach to providing a service application-wide. Instead of adding the service to the **providers'** array in the app.module.ts file, we can simply include **{providedIn: 'root'}** in the **@Injectable()** decorator of the service. This enables the use of the service throughout the entire application.

```
@Injectable({providedIn: 'root'})
```

Angular Tokens

Tokens are useful when there are multiple dependencies in the application. There are three types of tokens.

1. **Type token**
2. **String token**
3. **Injection token object**

Injecting Token →

It acts as a key to uniquely identify a provider, which is important because there can be multiple dependencies within an application. Tokens are therefore very useful for uniquely identifying these dependencies.

```
------------------------- log-message1.service.ts -------------------------
@Injectable()
export class LogMessage1Service {
  log() { console.log('This is Service 1'); }
}
------------------------- log-message2.service.ts -------------------------
@Injectable()
export class LogMessage2Service {
  log() { console.log('This is Service 2'); }
}
---------------------------- app.component.ts ----------------------------
export class AppComponent {
  constructor(private logger: LogMessage1Service) {
     this.logger.log();
  }
}
------------------------------ app.module.ts ------------------------------
@NgModule({
    declarations: [],
    imports: [BrowserModule, AppComponent],
    providers: [{
        provide: LogMessage1Service, useClass: LogMessage1Service,
        provide: LogMessage1Service, useClass: LogMessage2Service,
        // We use the same token for two different classes. In this case, the
output is displaying the second service file message.
    }],
    bootstrap: [AppComponent],
})
```

[Note: Here, we use a type token to inject the instance of the **logMessage1** service. The **provide** property holds the token, which acts as a key, allowing the dependency injection system to locate the provider associated with the token. The **useClass** property tells Angular's dependency injection system to instantiate the provided class when the dependency is injected.**]**

String Token →

String-based injection tokens are another way to register a provider.

```
---------------------------- app.component.ts ----------------------------
export class AppComponent {
    constructor(
      @Inject('LOG_MSG1')
      private logger: LogMessage1Service
    ) {
       this.logger.log();
    }
}
------------------------------ app.module.ts ------------------------------
providers: [{
    provide: 'LOG_MSG1', useClass: LogMessage1Service,
}],
```

Injection Token Object →

It is used as a provider token for non-class dependencies. When we need to provide a value or configuration that is not based on a class or type, then we can use an **InjectionToken** object.

```
--------------------------- injection-token.ts ---------------------------
import { InjectionToken } from '@angular/router';
export const LOG_MSG1 = new InjectionToken<LogMessage1Service>('');
---------------------------- app.component.ts ----------------------------
export class AppComponent {
    constructor(
        @Inject('LOG_MSG1')
        private logger: LogMessage1Service
    ) {
        this.logger.log();
    }
}
------------------------------ app.module.ts ------------------------------
providers: [{
    provide: 'LOG_MSG1', useClass: LogMessage1Service,
}],
```

Provider

Value Provider(`useValue`) → It provides a specific value to be used as a dependency. Instead of creating an instance of a class, it allows values to be directly assigned when the dependency is injected.

```
-------------------------- injection-token.ts --------------------------
import { InjectionToken } from '@angular/router';
export const STR_MSG = new InjectionToken<string>('Greeting');
--------------------------- app.component.ts ---------------------------
export class AppComponent {
    constructor(@Inject('STR_MSG') public msg: string) {
        console.log(this.msg); // Output - This is the string message
    }
}
----------------------------- app.module.ts -----------------------------
providers: [{
    provide: 'STR_MSG', useValue: 'This is the string message',
}],
```

[Note: This technique is mostly used when we want runtime configuration constants such as website base API addresses, etc.**]**

Alias Provider(useExisting) → It is used to map one token to another like an alias.

```
------------------------ alert-message1.service.ts ------------------------
export class AlertMessage1Service {
  showAlert() { alert('This is Service 1'); }
}
------------------------ alert-message2.service.ts ------------------------
export class AlertMessage2Service {
  showAlert() { alert('This is Service 2'); }
}
---------------------------- app.component.ts ----------------------------
export class AppComponent {
  constructor(private alert: AlertMessage1Service) { }
  displayAlert() { this.alert.showAlert(); // Output - This is Service 1 }
}
------------------------------ app.module.ts ------------------------------
providers: [ AlertMessage1Service,
    { provide: AlertMessage2Service, useExisting: AlertMessage1Service }
],
```

Note: The purpose of providing an alias token is to resume the functionality of **Alertmessage1** in different components/services.

Factory Provider(useFactory) → It allows us to define a dependency or service instance by calling a factory function. The factory provider(useFactory) expects a function to be defined each time it is declared, as it is used to provide a value that is not a class instance.

```
------------------------- message.service.ts -------------------------
export class MessageService {
    msg(): string { return 'This is a message'; }
}
-------------------------- app.component.ts --------------------------
export class AppComponent {
    constructor(private msgService: MessageService) {
        console.log(this.msgService.msg()); // Output - This is a message
    }
}
---------------------------- app.module.ts ---------------------------
providers: [{
    provide: MessageService,
    useFactory: () => {
      return new MessageService()
    }
}],
```

[Note: It is used to handle injectables and asynchronous calls, such as fetching values that are not available until runtime. For example, it can be used for loading data from another page or displaying data retrieved from a database.**]**

Example of useFactory with dependencies →

```
------------------------ app-config.service.ts ------------------------
export class AppConfigService {
    getAppConfig(): any {
        return {
            version: '1.0.0',
            environment: 'production',
            appName: 'my-app',
        };
    }
}
```

```
------------------------- app-update.service.ts -------------------------
export class AppUpdateService {
    constructor(@Inject('APP_UPDATE') private config: any) {}
    getAppUpdate(): string {
        const version = this.config.version;
        const environment = this.config.environment;
        const appName = this.config.appName;

        return 'Welcome Production Environment';
    }
}

--------------------------- app.component.ts ---------------------------
export class AppComponent {
    constructor(private appUpdateService: AppUpdateService) {
        this.message = this.appUpdateService.getAppUpdate();
    }
}

----------------------------- app.module.ts -----------------------------
providers: [AppConfigService, {
    provide: AppUpdateService,
    useFactory: (configService: AppConfigService) => {
        const config = configService.getAppConfig();
        return new AppUpdateService(config);
    },
    deps: [AppConfigService]
}],
```

[Note: The **deps** property is used to specify the dependencies used with the factory.**]**

Routing

We can create a routing module by running the command → **`ng g module app-routing --routing --flat`**.
In this command, **--routing** creates a dedicated file for routing, and **--flat** places the module file in the root directory of our project.

Setting & Config

1. To implement routing, start by creating a **Routes** array. Then, add this array to the **imports** array in the app.module.ts file using **RouterModule**. Additionally, include **<router-outlet></router-outlet>** in either the root HTML file or the default app.component.html file to render the route components.

```
------------------------- app.module.ts -------------------------
const appRoutes: Routes = [
    { path: '', component: HomeComponent },
    { path: 'users', component: UsersComponent }
];

@NgModule({ imports: [ RouterModule.forRoot(appRoutes) ] })
----------------------- app.component.html ----------------------
<div class="container"> // Commonly use components </div>
<div>
    <router-outlet></router-outlet>
</div>
```

2. When implementing routing in an **Angular** app, it's a good practice to create an app-routing.module.ts file to configure all routes. After setting the routing configurations, be sure to include **AppRoutingModule** in the **imports** array of the app.module.ts file.

```
--------------------- app-routing.module.ts ---------------------
const appRoutes: Routes = [
    { path: '', component: HomeComponent },
    { path: 'users/:id/:name', component: UserDetailsComponent },
    { path: 'servers', component: ServersComponent, children: [
       { path: ':id', component: ServerComponent },
       { path: ':id/edit', component: ServerEditComponent }
    ]},
    { path: 'not-found', component: NotFoundComponent },
    { path: '**', redirectTo: '/not-found', pathMatch: 'full' }
];

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

Navigation

1. In Angular, it is recommended to use **routerLink="/"** instead of **href="/"** to navigate between pages within an app. **routerLink** is an attribute directive that allows us to add behavior to an existing element by modifying its attributes. We can navigate using router **Links** as follows ⟶

```
<div class="container">
    <ul class="nav nav-tabs">
        <li class="active"><a routerLink="/">Home</a></li>
        <li><a routerLink="/servers">Servers</a></li>
        <li><a [routerLink]="['/users']">Users</a></li>
    </ul>
</div>
```

2. We already understand navigation paths, but there are two different ways to navigate using router links.

   a) **Relative path**: If we don't use a slash(/) or use relative paths(**./, ../, ../../, etc.**) before the route, the path will be appended to the current one. There are three distinct ways to utilize relative path navigation.

```
<li><a routerLink="servers">Servers</a></li>
----OR----
<li><a routerLink="./servers">Servers</a></li>
----OR----
<li><a routerLink="../servers">Servers</a></li>
```

   b) **Absolute path**: If we use a slash**(/)** before the path, it will automatically append to the root domain, starting from the base URL.

```
<li><a routerLink="/servers">Servers</a></li>
```

3. Programmatically Navigation ⟶

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

// Absolute Routing
this.router.navigate(['/servers']);

// Relative Routing
this.router.navigate(['servers'], {relativeTo: this.route});
```

Styling Active Router Links

We can set a class name based on the active router link →

```
<ul class="nav nav-tabs">
  <li
    routerLinkActive="active"
    [routerLinkActiveOptions]="{exact: true}">
       <a routerLink="/">Home</a>
  </li>
  <li routerLinkActive="active"><a routerLink="/servers">Servers</a></li>
  <li routerLinkActive="active"><a [routerLink]="['/users']">Users</a></li>
</ul>
```

In this code, **'active'** is a class name that can be customized to our preference. Additionally, we use **routerLinkActiveOptions** to specify options for matching the exact router link.

Passing Parameter & Accessing Data

1. Navigate a router link using route Parameters(id, name) →

```
------------------------- app.module.ts -------------------------
const appRoutes: Routes = [
    { path: 'users/:id/:name', component: UserDetailsComponent },
];
---------------------- users.component.ts ----------------------
<div class="container">
    <a [routerLink]="['/users', user.id, 'Anna']">Load User</a>
</div>
```

2. Navigate a router link with Query Parameter & Fragments →

```
------------------------- app.module.ts -------------------------
const appRoutes: Routes = [
    { path: 'servers/:id/edit', component: UserDetailsComponent },
];
--------------------- servers.component.html --------------------
<div class="container">
    <a [routerLink]="['/servers', server.id, 'edit']"
       [queryParams]="{allowEdit: '1'}"
       fragment="loading">
         {{ server.name }}
    </a>
</div>
```

```
// The navigated url is - localhost:4200/servers/1/edit?allowEdit=1
// If we use fragment - localhost:4200/servers/1/edit?allowEdit=1#loading
```

3. Navigate a router link with **Query Params** programmatically on a **ts** file →

```
this.router.navigate(
    ['/servers', id, 'edit'],
    {queryParams: {allowEdit: '1'}, fragment: 'loading' }
);
```

4. Passing parameters to routes & Fetch →

```
------------------------- app.module.ts -------------------------
const appRoutes: Routes = [
    { path: '', component: HomeComponent },
    { path: 'users', component: UsersComponent },
    { path: 'users/:id/:name', component: UserDetailsComponent },
    { path: 'servers', component: ServersComponent }
];
                      Fetch Data from Route
------------------- user-details.component.ts --------------------
user: {id: number, name: string} = {};
constructor(private route: ActivatedRoute) {}

getData() {
    this.user = {
        id: this.route.snapshot.params['id'],
        name: this.route.snapshot.params['name'];
    };
}
                      Fetch Data Reactively
-------------------- user-details.component.ts -------------------
paramsSubscriptions: Subscription;

ngOnInit() {
   this.paramsSubscriptions = this.route.params.subscribe(
      (params: Params) => {
          this.user.id = params['id'];
          this.user.name = params['name'];
      });
}

ngOnDestroy() { this.paramsSubscription.unsubscribe(); }
```

[Note: Unsubscribing from observables is essential to prevent memory leaks. When a component is destroyed and removed from memory, its observables can persist. To avoid this, it's crucial to unsubscribe from all subscriptions in the component's **ngOnDestroy** lifecycle method.**]**

5. Retrieves data from Routes →

```
                    Retrieves Data from Route
-------------------- server-edit.component.ts -------------------
constructor(private activeRoute: ActivatedRoute) {}

getData() {
    const isEdit = this.activeRoute.snapshot.queryParams['allowEdit'];
    const fragment = this.activeRoute.snapshot.fragment;
}
                     Fetch Data Reactively
ngOnInit() {
    this.activeRoute.queryParams.subscribe(
        (params: Params) => { // Our own code });
            this.activeRoute.fragment.subscribe();
        )};
}
```

[Note: We don't need to unsubscribe this because angular automatically handles it.**]**

```
                     Another better Way
ngOnInit() {
    const subscription = this.activeRoute.paramMap.subscribe({
      next: (paramMap: any) => this.userId = paramMap.get('userId')
    });
}
                     For Query Parameter
ngOnInit() {
    const subscription = this.activeRoute.queryParams.subscribe({
        next: (param: any) => this.userId = param.get('userId')
    });
}
```

[Note: This is a reactive mechanism. When we change a component's data, such as updating the URL (e.g., a user change), the component is re-executed.**]**

6. If we want to retain the current query parameters when navigating to the next route, we need to use **queryParamsHandling** while initiating the navigation to a new route.

```
this.router.navigate(
    ['edit'],
    { relativeTo: this.route, queryParamsHandling: 'preserve' }
);
```

[Note: When **'merge'** is used instead of **'preserve'** in queryParamsHandling, it will combine the existing query parameters with the new ones during navigation.**]**

7. Passing Static Data to a Route →

```
--------------------- app-routing.module.ts ---------------------
const appRoutes: Routes = [
    { path: '', component: HomeComponent },
    { path: 'not-found', component: NotFoundComponent,
      data: { message: 'Page not Found!' }
    },
    { path: '**', redirectTo: '/not-found', pathMatch: 'full' }
];
                        Retrieves data from route
--------------------- not-found.component.ts --------------------
import { Data } from '@angular/router';

constructor(private route: ActivatedRoute) { }
ngOnInit() {
    this.errorMessage = this.route.snapshot.data['message'];
    // -- OR -- //
    this.route.data.subscribe(
        (data: Data) => { this.errorMessage = data['message']; });
}
```

8. Accessing Route Data In Components →

```
constructor(private activatedRoute: ActivatedRoute) { }
ngOnInit() {
    this.activatedRoute.data.subscribe({
        next: (data: any) => { console.log(data); }
    });
}
```

9. To pass data dynamically to a route, we can use **Resolver Guard**. Please refer to the **Guard** section for more information.

Nested Route

1. The general form of nested route →

```
const appRoutes: Routes = [
    { path: '', component: HomeComponent },
    { path: 'users/:id/:name', component: UserDetailsComponent },
    { path: 'servers', component: ServersComponent,
        children: [
            { path: ':id', component: ServerComponent },
            { path: ':id/edit', component: ServerEditComponent }
        ]
    }
];
```

2. Splitting Route Across Multiple files →

```
------------------------ user.routes.ts ------------------------
export const routes: Routes = [
    { path: '', redirectTo: 'tasks', pathMatch: 'full' },
    { path: 'tasks', component: TasksComponent },
    { path: 'tasks/new' component: NewTaskComponent }
];


------------------------- app.routes.ts ------------------------
import { routes as userRoutes } from './users/users.routes';

export const routes: Routes = [
    { path: 'tasks', component: TasksComponent },
    {
        path: 'user/:userId', component: UserComponent,
        children: userRoutes
    },
    { path: '**', component: NotFoundComponent }
];
```

3. In the **servers.component.html** file, we can use **<router-outlet>**. This serves as a placeholder where child route components will be loaded, allowing for the dynamic rendering of components associated with the child routes of the current route in the **server's** component.

```
<div class="container">
    <ul class="nav nav-tabs">
        <li class="active"><a routerLink="/">Servers</a></li>
        <li><a routerLink="/:id">Server</a></li>
        <li><a [routerLink]="/:id/edit">Edit</a></li>
    </ul>
    <div> <router-outlet></router-outlet> </div>
</div>
```

Guard

We can create a route guard by running the command → **`ng g guard NAME`**.

1. To implement the auth guard feature, we will create a new file called auth-guard.service.ts in the root(**src**) directory. This service will contain all the guard-related code, and It is important that Angular executes this code before a route is loaded. Typically, Angular guards are implemented using a service that adheres to the **CanActivate** or other guard **interfaces**.

```
--------------------- auth-guard.service.ts ---------------------
@Injectable()
export class AuthGuard implements CanActivate {
  constructor(private authService: AuthService, private router: Router) {}

  canActivate( route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> |
boolean {
      return this.authService.isAuthenticated()
        .then((authenticated: boolean) => {
            if (authenticated) { return true; }
            else { this.router.navigate(['/']); }
        });
    }
}
------------------------ auth.service.ts ------------------------
export class AuthService {
    loggedIn: boolean = false;
    isAuthenticated() {
        return new Promise((resolve, reject) => {
            resolve(this.loggedIn);
        });
    }
    login() { this.loggedIn = true; }
    logout() { this.loggedIn = false; }
}
--------------------- app-routing.module.ts ---------------------
const appRoutes: Routes = [
    { path: '', component: HomeComponent },
    { path: 'users/:id/:name', component: UserDetailsComponent },
    { path: 'servers',
      canActivate: [AuthGuard], component: ServersComponent,
      children: [
         { path: ':id', component: ServerComponent },
         { path: ':id/edit', component: ServerEditComponent }
      ]
    },
];
```

[Note: Must add both services into the providers' array of the app.module.ts file.]

2. **canActivate:** It can return either an **observable**, a **promise**, or a **boolean** value. This means **canActivate** can operate both asynchronously(by returning an observable or a promise) and synchronously(by returning a boolean value).
3. **Protecting Child Routes:** To protect child routes, we need to implement **canActivateChild** in the auth-guard.service.ts file.

```
--------------------- auth-guard.service.ts ---------------------
@Injectable()
export class AuthGuard implements CanActivate, CanActivateChild {
  constructor(private authService: AuthService, private router: Router) { }

  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean
  {
      // Logical Code that we added above }
     canActivateChild( route: ActivatedRouteSnapshot,
       state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean
     {
        return this.canActivate(route, state);
     }
  }
}
```

```
-------------------- app-routing.module.ts --------------------
const appRoutes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'users/:id/:name', component: UserDetailsComponent },
  {
    path: 'servers',
    canActivate: [AuthGuard],
    canActivateChild: [AuthGuard],
    component: ServersComponent,
    children: [
      { path: ':id', component: ServerComponent },
      { path: ':id/edit', component: ServerEditComponent }
    ]
   },
];
```

[Note: Here, the parent and child routes are protected. If we remove canActivate: [AuthGuard], only the child routes will be protected.**]**

4. **Resolve Guard:** This service, similar to **CanActivate** and **CanDeactivate**, allows us to execute code before a route is rendered. The resolver performs preloading tasks, such as fetching data that the component will need later, before rendering the component.

```
------------------- server-resolver.service.ts -------------------
import { Resolve } from '@angular/router';
interface Server {
    id: number;
    status: string;
}

@Injectable()
export class ServerResolver implements Resolve<Server> {
    constructor(private serverService: ServerService) { }

    resolve( route: ActivateRouteSnapshot, state: RouterStateSnapshot):
        Observable<Server> | Promise<Server> | Server {
      return this.serverService.getServer(+route.params['id']);
    }
}
```

```
--------------------- app-routing.module.ts ---------------------
const appRoutes: Routes = [
    { path: '', component: HomeComponent },
    {  path: 'servers', canActivate: [AuthGuard],
       canActivateChild: [AuthGuard], component: ServersComponent,
       children: [
          { path: ':id', component: ServerComponent,
            resolve: { server: ServerResolver }
          }, // We can use any name as key instead of 'server'.
          { path: ':id/edit', component: ServerEditComponent }
       ]
    }
];
                        Retrieves data from route
---------------------- server.component.ts ----------------------
import { Data } from '@angular/router';
export class ServerComponent implements OnInit {
    constructor(private route: ActivatedRoute) { }
    ngOnInit() {
        this.route.data.subscribe(
           (data: Data) => { this.server = data['server']; });
    }
}
```

Must Know

1. Execute Route Resolver when QueryParams change →

```
----------------------- tasks.component.ts ----------------------
export const resolveUserTasks: ResolveFn<Task[]> = (activatedRoute:
ActivatedRouteSnapshot, routerState: RouterStateSnapshot) => {
  const order = activatedRoute.queryParams['order'];
  const tasksService = inject(TasksService);
  const tasks = tasksService.allTasks().filter(
     (task: any) => task.userId === activatedRoute.paramMap.get('userId')
  );

  if (order && order === 'asc') tasks.sort((a, b) => (a.id > b.id ? 1 : -1));
  else tasks.sort((a: any, b: any) => (a.id > b.id ? -1 : 1));

  return tasks.length ? tasks : [];
};
```

S M HEMEL
Senior Software Engineer
Email: smhemel.eu@gmail.com

```
------------------------ users.routes.ts ------------------------
const appRoutes: Routes = [
    { path: '', component: HomeComponent },
    {
        path: 'tasks', component: TasksComponent,
        runGuardsAndResolvers: 'always', // values will be ->
paramsOrQueryParamsChange | paramsChange | pathParamsChange |
pathParamsOrQueryParamsChange
        resolve: { userTasks: resolveUserTasks }
    }
];
```

2. Setting Browser Tab title depends on Route ⟶

```
----------------------- tasks.component.ts ----------------------
export class TasksComponent {
    userName = input.required<string>();
}

export const resolveUserName: ResolveFn<string> = (activatedRoute:
ActivatedRouteSnapshot, routerState: RouterStateSnapshot) => {
  const usersService = inject(UsersService);
  const userName = usersService.users.find(
    (u: any) => u.id === activatedRoute.paramMap.get('userId'))?.name || '';
  return userName;
};

export const resolveTitle: ResolveFn<string> = (activatedRoute, routerState)
=> {
  return resolveUserName(activatedRoute, routerState) + '\'s Tasks';
}
------------------------- app.routes.ts -------------------------
const appRoutes: Routes = [
    { path: '', component: HomeComponent },
    { path: 'tasks', component: TasksComponent,
      runGuardsAndResolvers: 'always',
      resolve: { userName: resolveUserName }, title: resolveTitle
    }
];
```

3. Wildcard Route: Defined with a **double asterisk**(**) in Angular, this route catches all routes that do not exist in the application. It is conventionally placed at the end of the routes array.

```
const appRoutes: Routes = [
    { path: '', component: HomeComponent },
    { path: 'not-found', component: NotFoundComponent },
    { path: '**', redirectTo: '/not-found', pathMatch: 'full' }
];
```

4. Control accidentally clicks the back button or navigates back which is called <u>Deactivate Route</u> →

```
-------------------- edit-server.component.ts --------------------
import { CanComponentDeactivate } from './can-deactivate-guard.service';
export class EditServerComponent implements CanComponentDeactivate {
  changesSaved: boolean = false;

  onUpdateServer() {
    this.serverService.updateServer(this.server.id);
    this.changesSaved = true;
  }

  canDeactivate(): Observable<boolean> | Promise<boolean> | boolean {
    if (this.allowEdit) { return true; }
    if (this.serverName !== this.server.name) {
      return confirm('Do you want to discard the changes?');
    } else { return true; }
  }
}

---------------- can-deactivate-guard.service.ts ----------------
import { CanDeactivate } from '@angular/router';
export interface CanComponentDeactivate {
    canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean;
}

export class CanDeactivateGuard implements CanDeactivate<CanComponentDeactivate>
{
  canDeactivate(
    component: CanComponentDeactivate, currentRoute: ActivatedRouteSnapshot,
    currentState: RouterStateSnapshot, nextState?: RouterStateSnapshot
  ): Observable<boolean> | Promise<boolean> | boolean {
      return component.canDeactivate();
  }
}
```

```
--------------------- app-routing.module.ts ---------------------
const appRoutes: Routes = [
    { path: '', component: HomeComponent },
    { path: 'users/:id/:name', component: UserDetailsComponent },
    { path: 'servers',
      canActivateChild: [AuthGuard],
      component: ServersComponent, children: [
            { path: ':id', component: ServerComponent },
            { path: ':id/edit',
              component: ServerEditComponent,
              canDeactivate: [CanDeactivateGuard]
            }
       ]
    },
];
```

[Note: Now, Angular will execute the CanDeactivateGuard when attempting to leave the edit component. Make sure to add the CanDeactivateGuard service to the providers' array in the **app.module.ts** file.**]**

5. When deploying our app on a server, a specific configuration is needed to handle routes correctly. After deployment, navigating to **`domain/our_path`** might result in a **404 error** because the server handles the URL. To address this, add **`{useHash: true}`** to the **imports** array in the app.module.ts file. This enables hash mode routing, where a **'#'** symbol is added between the domain and path. In this mode, the web server ignores paths after the hash(#), ensuring proper routing.

```
------------------------- app.module.ts -------------------------
const appRoutes: Routes = [
    { path: '', component: HomeComponent },
    { path: 'users', component: UsersComponent },
];

@NgModule({
    imports: [
        RouterModule.forRoot(appRoutes, {useHash: true})
    ],
    exports: [RouterModule]
})
```

Observable

1. **What is an Observable?** → An Observable is a data source in an Angular project, implemented using the observable pattern from the **RxJS** library. It allows us to handle multiple emitted events asynchronously. Paired with an **Observer**, the Observable can emit data in response to certain triggers, enabling efficient event handling.
2. Install **RxJS** & **RxJS** Compat →
   ```
   npm install --save rxjs
   npm install --save rxjs-compat
   ```
3. **Angular Observable:** In routing, we often subscribe to route **parameters**, which are essentially observables. We use the **"subscribe"** function because, when the observable value changes, the updated value is provided through the subscribe callback. It's important to note that Angular has many built-in observables, which we will explore later.
   Note: Angular's built-in observables don't require manual unsubscription, as Angular manages this automatically.
4. Create an observable by using the **Interval** method of **RxJS** →
   ```
   import { interval, Subscription } from 'rxjs';
   export class DemoComponent implements OnInit, OnDestroy {
       private demoObservable: Subscription;
       ngOnInit() {
           this.demoObservable = interval(1000).subscribe(next => {
               // Our code which will trigger after 1 second interval
           });
       }

       ngOnDestroy() {
           this.demoObservable.unsubscribe();
       }
   }
   ```
5. Build custom observable manually with **Error & Completion example** →
   ```
   import { Observable } from 'rxjs';

   export class DemoComponent implements OnInit, OnDestroy {
       private demoObservable: Subscription;
      ngOnInit() {
   ```

```
        const customObservable = Observable.create(observer => {
            let count = 0;
            setInterval(() => {
                observer.next(count);
                // observer.error(new Error('Throw Demo Error'));
                // observer.complete(new Error('Throw Demo Error'));
                count++;
            }, 1000);
        });

        this.demoObservable = customObservable.subscribe(data => {
            // logical code
        }, err => { console.log(err); },
        () => { console.log('Completed!'); // logical Code });
    }

    ngOnDestroy() { this.demoObservable.unsubscribe(); }
}
```

[Note: If we wish to handle errors in an observable, we include an error block. To manage the completion of the observable, we add a third block to our code, which handles the completion event.**]**

6. If an Observable throws an error, it terminates and will not emit any more values, so there's no need to unsubscribe. Similarly, when an Observable completes, it stops emitting values, and no unsubscription is necessary.
7. **Operators** → Operators are a powerful feature of the **RxJS** library. When we have an **observable** and an **observer**, we typically receive data by subscribing to the observable. However, there are times when we want to transform, filter, or manipulate the data before handling it. This can be done either within the subscription or by passing a function to it.
   Rather than manually setting up transformations inside the subscription, we can use built-in RxJS operators. These operators process the data before we subscribe, allowing the data to flow through the operators for manipulation. RxJS provides a wide range of operators. Below are examples of two common operators: **`map`** and **`filter`**, demonstrating how to use them.

```
import { Observable } from 'rxjs';
import { map, filter } from 'rxjs/operators';

export class DemoComponent implements OnInit, OnDestroy {
    private demoObservable: Subscription;

    ngOnInit() {
        const customObservable = Observable.create(observer => {
            let count = 0;
            setInterval(() => {
                observer.next(count);
                count++;
            }, 1000);
        });

        this.demoObservable = customObservable.pipe(filter(data => {
            return data > 0;
        }), map((data: number) => {
            return 'Round: ' + (data + 1);
        })).subscribe(data => { // Our logical code });
    }

    ngOnDestroy() {
        this.demoObservable.unsubscribe();
    }
}
```

[Note: Every observable has a pipe method.**]**

8. **Subject** → It is similar to Angular's **`EventEmitter`**, but using a **Subject** is the recommended and more superior way to emit events in Angular. It provides greater flexibility and control over event streams.

```
------------------------ user.service.ts ------------------------
import { EventEmitter, Injectable } from '@angular/core';
import { Subject } from 'rxjs';

@Injectable({ providedIn: 'root' })
export class UserService {
  // activatedEmitter = new EventEmitter<boolean>(); // Old way
  activatedEmitter = new Subject<boolean>();
  constructor(private authService: AuthService, private router: Router) { }
}
----------------------- user.component.ts -----------------------
```

```
export class UserComponent {
    constructor(private userService: UserService) { }

    onActivate() {
        // this.userService.activatedEmitter.emit(true); // Old way
        this.userService.activatedEmitter.next(true);
    }
}

------------------------ app.component.ts -----------------------
export class AppComponent implements OnInit, OnDestroy {
    userActivated = false;
    private activateSub: Subscription;

    constructor(private userService: UserService) { }

    ngOnInit() {
        this.activateSub = this.userService.activatedEmitter
          .subscribe(didActivate => {
            this.userActivated = didActivate;
          });
    }

    ngOnDestroy(): void {
        this.activateSub.unsubscribe();
    }
}
```

[Note: We should ensure to unsubscribe from our subjects when they are no longer needed to prevent memory leaks.**]**

Important Note: The **`Subject`** is specifically used as a cross-component event emitter, where we manually invoke **`next`** to emit events. It is not intended to replace **`EventEmitter`** when using **`@Output()`** in Angular.

9. RxJS documentation → https://rxjs-dev.firebaseapp.com/

Angular Forms

Angular offers us two approaches to handling forms.

a) **Template Driven** → Angular infers the Form Object from the DOM.

b) **Reactive** → Form is created programmatically and synchronized with the DOM.

To enable the use of form features, first ensure that the **FormsModule** is imported into the **imports** array of the app.module.ts file. This will include a variety of forms-related functionality in our application. This provides various forms-related functionalities within the application.

Template Driven Approach

1. When we import the **FormsModule** into the app.module.ts file, Angular automatically generates a JavaScript representation of the form when it detects a form element in the HTML code.

   Create Form →

```
--------------------- demo.component.html ---------------------
<form class="form-container">
    <div class="form-group">
        <label for="name">Name</label>
        <input type="text" id="name" ngModel name="name" required>
    </div>
    <div class="form-group">
        <label for="email">Email</label>
        <input type="email" id="email" ngModel name="email" required>
    </div>
    <div class="form-group">
        <label for="secret">Secret Questions</label>
        <select type="secret" id="secret" ngModel name="secret">
            <option value="pet">Your First pet?</option>
            <option value="teacher">Your first teacher?</option>
        </select>
    </div>

    <button class="btn form-btn" type="submit">Submit</button>
</form>
```

2. Submit Form →

```
---------------------- demo.component.html ----------------------
<form class="form-container" (ngSubmit)="onSubmit(f)" #f="ngForm"> </form>
----------------------- demo.component.ts -----------------------
onSubmit(form: NgForm) {
    console.log(form); // Now, we can access all the properties of the form
that we will use to control the form
    form.form.reset();
}
```

3. Accessing form with @ViewChild() →

```
@ViewChild('f') signUpForm: NgForm;
onSubmit() { console.log(signUpForm); }  // We can access all the properties
of the form that we will use to control the form
```

4. **Check out all validators** → https://angular.io/api/forms/Validators
5. Use Form State →

```
---------------------- demo.component.html ----------------------
<form class="form-container" (ngSubmit)="onSubmit(f)" #f="ngForm">
  <button class="btn" type="submit" [disabled]="!f.valid">
      Submit
  </button>
</form>
----------------------- demo.component.css ----------------------
input.ng-invalid.ng-touched {
   border: 1px solid red;
}
```

6. Output validation error message →

```
<form class="form-container" (ngSubmit)="onSubmit(f)" #f="ngForm">
   <div class="form-group">
      <label for="email">Email</label>
      <input #email="ngModel" type="email" id="email" ngModel name="email" required>
      <span *ngIf="!email.valid & email.touched" class="help-message">
            Please enter a valid email.
      </span>
   </div>

   <button class="btn form-btn" type="submit">Submit</button>
</form>
```

7. Set default values →

```
---------------------- demo.component.html ----------------------
<form class="form-container" (ngSubmit)="onSubmit(f)" #f="ngForm">
  <div class="form-group">
    <label for="secret">Secret Questions</label>
    <select type="secret" id="secret" ngModel name="secret" [ngModel]="defaultQuestion">
      <option value="pet">Your First pet?</option>
      <option value="teacher">Your first teacher?</option>
    </select>
  </div>
  <button class="btn form-btn" type="submit">Submit</button>
</form>
----------------------- demo.component.ts -----------------------
defaultQuestion: any = "pet";
```

8. Two-way-binding using ngModel →

```
---------------------- demo.component.html ----------------------
<form class="form-container" (ngSubmit)="onSubmit(f)" #f="ngForm">
  <div class="form-group">
    <label for="email">Email</label>
    <input type="email" id="email" [(ngModel)]="emailAddress" name="email" required>
  </div>
  <button class="btn form-btn" type="submit">Submit</button>
</form>
----------------------- demo.component.ts -----------------------
emailAddress: string = "";
```

9. Grouping form controls →

```
---------------------- demo.component.html ----------------------
<form class="form-container" (ngSubmit)="onSubmit(f)" #f="ngForm">
  <div id="user-data" ngModelGroup="userData" #userData="ngModelGroup">
    <div class="form-group">
      <label for="name">Name</label>
      <input type="text" id="name" ngModel name="name" required>
    </div>
    <div class="form-group">
      <label for="email">Email</label>
      <input type="email" id="email" ngModel name="email" required>
    </div>
  </div>
  <span *ngIf="!userData.valid & userData.touched">User data is invalid!</span>
</form>
```

10. Handling radio buttons →

```
---------------------- demo.component.html ----------------------
<form class="form-container" (ngSubmit)="onSubmit(f)" #f="ngForm">
   <div class="radio" *ngFor="let gender of genders">
      <label>
         <input type="radio" name="gender" ngModel [value]="gender">
         {{ gender }}
      </label>
   </div>
   <button class="btn form-btn" type="submit">Submit</button>
</form>
```

11. Setting up form values →

```
----------------------- demo.component.ts -----------------------
@ViewChild('f') signUpForm: NgForm;
initFormValue() {
   this.signUpForm.setValue({
      userData: { username: 'Mark', email: 'mark@gmail.com' },
      secret: 'pet',
      questionAnswer: '',
      gender: 'male'
   });
}
```

12. Patching or updating any form field values →

```
----------------------- demo.component.ts -----------------------
@ViewChild('f') signUpForm: NgForm;
updateFormValue() {
   this.signUpForm.form.patchValue({
      userData: { username: 'Mark 2' }
   });
}
```

[Note: It will update only the **username** field value, all other field values are unchanged. **]**

13. Reset Form →

```
@ViewChild('f') signUpForm: NgForm;
onSubmit() {
   console.log(this.signUpForm.value.userData.username);
   this.signUpForm.reset();
}
```

Reactive Approach

1. Angular provides numerous tools for efficiently creating reactive forms. To utilize these reactive form features, it's essential to include **ReactiveFormsModule** in the **imports** array of the app.module.ts file or a specific module file. Now, let's explore how to set up a reactive form by creating one →

```
----------------------- demo.component.ts -----------------------
export class DemoComponent implements OnInit {
    signupForm: FormGroup;
    ngOnInit() {
        this.signupForm = new FormGroup({
            'username': new FormControl(null),
            'email': new FormControl(null),
            'gender': new FormControl('male')
        });
    }
}
```

[Note: We should initialize the form before initializing the template.**]**

2. Syncing HTML with form and submit functionality →

```
---------------------- demo.component.html ----------------------
<form class="form-container" [formGroup]="signupForm" (ngSubmit)="onSubmit()">
    <div class="form-group">
        <label for="name">Name</label>
        <input type="text" id="name" formControlName="username">
    </div>
    <div class="form-group">
        <label for="email">Email</label>
        <input type="email" id="email" formControlName="email">
    </div>
    <div class="radio" *ngFor="let gender of genders">
        <label>
            <input type="radio" formControlName="gender" [value]="gender">
            {{ gender }}
        </label>
    </div>
    <button class="btn form-btn" type="submit">Submit</button>
</form>
----------------------- demo.component.ts -----------------------
onSubmit() { console.log(this.signupForm); }
```

3. Add Validation →

```
ngOnInit() {
  this.signupForm = new FormGroup({
    'username': new FormControl(null, Validators.required),
    'email': new FormControl(null, [Validators.required, Validators.email]),
      'gender': new FormControl('male')
  });
}
```

4. Access form controls to display help messages or anything else →

```
---------------------- demo.component.html ----------------------
<form class="form-container" [formGroup]="signupForm" (ngSubmit)="onSubmit()">
  <div class="form-group">
    <label for="email">Email</label>
    <input type="email" id="email" formControlName="email">
    <span class="help-message"
          *ngIf="signupForm.get('email').valid && signupForm.get('email').touched"
    >
        Please enter a valid email.
    </span>
  </div>
</form>
<span *ngIf="signupForm.valid && signupForm.touched">
    Please enter valid data.
</span>
---------------------- demo.component.css -----------------------
input.ng-invalid.ng-touched {
    border: 1px solid red;
}
```

5. Grouping or nested form controls →

```
----------------------- demo.component.ts -----------------------
ngOnInit() {
  this.signupForm = new FormGroup({
     'userData': new FormGroup({
        'username': new FormControl(null, Validators.required),
        'email': new FormControl(null, [Validators.required]),
     }),
     'gender': new FormControl('male')
  });
}
```

```
---------------------- demo.component.html ----------------------
<form class="form-container" [formGroup]="signupForm" (ngSubmit)="onSubmit()">
  <div formGroupName="userData">
    <div class="form-group">
      <label for="name">Name</label>
      <input type="text" id="name" formControlName="username">
    </div>
    <div class="form-group">
     <label for="email">Email</label>
     <input type="email" id="email" formControlName="email">
     <span *ngIf="signupForm.get('userData.email').valid &&
signupForm.get('userData.email').touched">
         Please enter a valid email.
     </span>
    </div>
  </div>
  <div class="radio" *ngFor="let gender of genders">
    <label>
      <input type="radio" formControlName="gender" [value]="gender"> {{ gender }}
    </label>
  </div>
  <button class="btn form-btn" type="submit">Submit</button>
</form>
```

6. **FormArray** is used to create an array of form controls dynamically, allowing us to manage a dynamic number of input fields in our reactive forms. This is particularly useful when the number of inputs can change based on user interactions or other conditions.

```
---------------------- demo.component.html ----------------------
<form class="form-container" [formGroup]="signupForm" (ngSubmit)="onSubmit()">
  <div formGroupName="userData"> //Other fields </div>

  <div formArrayName="hobbies">
    <h4>Your Hobby</h4>
    <button type="button" (click)="onAddHobby()">Add Hobby</button>
    <div class="form-group" *ngFor="let hobby of getControls(); let i = index">
       <input type="text" [formControlName]="i">
    </div>
  </div>
  <button class="btn form-btn" type="submit">Submit</button>
</form>
```

```
----------------------- demo.component.ts -----------------------
ngOnInit() {
  this.signupForm = new FormGroup({
     'userData': new FormGroup({
        'username': new FormControl(null, Validators.required),
        'email': new FormControl(null, [Validators.required]),
     }),
     'hobbies': new FormArray([])
  });
}
onAddHobby() {
  const control = new FormControl(null, Validators.required);
  (<FormArray>this.signupForm.get('hobbies')).push(control);
    // We use <FormArray> to explicitly cast the form
}
getControls() { return (<FormArray>this.signupForm.get('hobbies')).controls; }
```

7. **Create custom validators →**

```
----------------------- demo.component.ts -----------------------
forbiddenUsernames: any = ['Chris', 'Anna'];
ngOnInit() {
  this.signupForm = new FormGroup({
    'userData': new FormGroup({
       'username': new FormControl(null, [Validators.required,
this.forbiddenNames.bind(this)]),
       'email': new FormControl(null, [Validators.required,
this.mustContainQuestionMark.bind(this)]),
    }),
    'hobbies': new FormArray([])
  });
}

forbiddenNames(control: FormControl): { [s: string]: boolean } {
    if (this.forbiddenUsernames.indexOf(control.value) !== -1)
        return { 'nameIsForbidden': true };
    return null;
}
mustContainQuestionMark(control: AbstractControl) {
    if (control.value.contains('?')) return null;
    return { doesNotContainQuestionMark: true };
}
```

[Note: We bind **`this`** when setting the **forbiddenNames** function as a validator. This is essential because, when Angular checks the validity and calls the **forbiddenNames** function, it needs to know the context of the caller. Since the function is not called from the class, **`this`** would not refer to our class. Therefore, binding **`this`** ensures the correct context during the validation process.**]**

8. **Handling Error Codes from Validators:** When working with custom validators or built-in validators in Angular, it's important to handle the error codes they generate. Each validator can return an object containing error codes that indicate what went wrong. →

```
---------------------- demo.component.html ----------------------
<form class="form-container" [formGroup]="signupForm" (ngSubmit)="onSubmit()">
  <div formGroupName="userData">
    <div class="form-group">
      <label for="name">Name</label>
      <input type="text" id="name" formControlName="username">
      <span *ngIf="signupForm.get('userData.username').errors['nameIsForbidden']">
         Please enter a valid Name.</span>
    </div>
  </div>
  <button class="btn form-btn" type="submit">Submit</button>
</form>
```

9. Create custom **Async Validator** →

```
----------------------- demo.component.ts -----------------------
ngOnInit() {
  this.signupForm = new FormGroup({
    'userData': new FormGroup({
      'username': new FormControl(null, Validators.required),
      'email': new FormControl(null, [Validators.required, Validators.email],
this.forbiddenEmails)
      })
  });
}
forbiddenEmails(control: FormControl): Promise<any> | Observable < any > {
  return new Promise<any>((resolve, reject) => {
    setTimeout(() => {
      if (control.value === 'test@gmail.com')
         resolve({ 'nameIsForbidden': true });
      else resolve(null);
    }, 1500);
  }
}
```

```
             ------ A N O T H E R   E X A M P L E -------
[Note: For some websites, we need to check immediately whether an
email or username is valid or not. This is a best example to use
async validation.]

function emailIsUnique(control: AbstractControl) {
  // We can trigger API call here to check email
  if (control.value !== 'test@example.com') return of(null);
  return of({notUnique: true});
}
export class LoginComponent {
  signupForm = new FormGroup({
    username: new FormControl('', Validators.required),
    email: new FormControl('', {
      validators: [Validators.required, Validators.email,
Validators.minLength(15)],
      asyncValidators: [emailIsUnique],
    }),
  });
}
[Note: The RxJS method of() produces an observable that immediately
emits a value.]
```

10. Status and Value changes →

```
----------------------- demo.component.ts -----------------------
ngOnInit() {
  this.signupForm = new FormGroup({
    'userData': new FormGroup({
      'username': new FormControl(null, Validators.required),
      'email': new FormControl(null, [Validators.required,
Validators.email], this.forbiddenEmails)
    })
  });

  this.signupForm.valueChanges.subscribe((value) => {
    console.log(value);
  });
  this.signupForm.statusChanges.subscribe((status) => {
    console.log(status);
  });
}
```

S M HEMEL
Senior Software Engineer
Email: smhemel.eu@gmail.com

11. Setting up form values →

```
initFormValue() {
    this.signupForm.setValue({
        'userData': { 'username': 'Mark', 'email': 'mark@gmail.com' },
        'gender': 'male', 'hobbies': []
    });
}
```

12. Patching or updating any form field values →

```
initFormValue() {
    this.signupForm.patchValue({ 'userData': { 'username': 'Mark 2' } });
}
```

13. Creating Multi-Input Validators / Form Group Validators →

```
function equalValues(control: AbstractControl) {
    const password = control.get('password')?.value;
    const confirmPassword = control.get('confirmPassword')?.value;
    if (password === confirmPassword) return null;
    return { passwordNotEqual: true };
}
                            ----- OR -----
function equalValues(controlName: string, controlName2: string) {
    return (control: AbstractControl) => {
        const val1 = control.get(controlName)?.value;
        const val2 = control.get(controlName2)?.value;
        if (val1 === val2) return null;
        return { passwordNotEqual: true };
    }
}
export class LoginComponent {
  signupForm = new FormGroup({
    email: new FormControl(null, [Validators.required]),
    passwords: new FormGroup({
      password: new FormControl(null, Validators.required),
      confirmPassword: new FormControl(null, [Validators.required]),
    }, { validators: [equalValues]
            // validators: [equalValues('password', 'confirmPassword')]
    }),
    gender: new FormControl('male')
  });
}
```

S M HEMEL
Senior Software Engineer
Email: smhemel.eu@gmail.com

14. Reset form →

```
this.signupForm.reset();
```

Pipe To Transform Output

1. **`What is Pipe?`** → Pipe is a built-in feature in Angular that allows us to transform the output directly in our template.
   Angular Pipes: **async**, **lowercase**, **uppercase**, **json**, **filter**, **orderBy**, **map**, **percent**, **date**, **currency**, **slice**, **join**.
   To create a custom pipe, run the command: **`ng g p pipe_name`**.

```
<div class="server-info">
    <strong>{{ server.name }}</strong>
    <span>{{ server.type | uppercase }}-{{ server.start | date }}</span>
</div>
```

2. **Slice** pipe → It is a built-in pipe that allows us to extract or slice a portion of an array or string. It creates a new array or string containing the sliced portion, often referred to as a **subset** in programming terms.

```
<span>{{'Angular slice pipe' | slice:0:7 }}</span> // Output: Angular
<span>{{'Angular slice pipe' | slice:0:-11 }}</span> // Output: Angular
<span>{{'Angular slice pipe' | slice:-11 }}</span> // Output: slice pipe
<span>{{ sliceArr | slice:0:2 }}</span> // Output: Item1,Item2
<span>{{ sliceArr | slice:3 }}</span> // Output: Item3
```

3. **Number** pipe →

```
<span>{{ 110 | number:'3.1-2' }}</span> // Output: 110.0
<span>{{ 110 | number:'3.2-3' }}</span> // Output: 110.00
<span>{{ 110 | number:'3.3-4' }}</span> // Output: 110.000
<span>{{ 110.66 | number:'3.1-1' }}</span> // Output: 110.7
<span>{{ pi | number:'1.2-2' }}</span> // Output: 3.14
```

4. Use parameter with pipe →

```
<span>{{ server.start | date:'fullDate' }}</span>
```

[Note: We can use multiple parameters with pipe separating by **colon(:)**.**]**

5. Use multiple pipes →

```
<span>{{ server.start | date:'fullDate' | uppercase }}</span>
```

[Note: Make sure we will follow proper chaining while using multiple pipes. **]**

6. **Creating a Custom Pipe:** After creating a custom pipe, it's crucial to add the pipe to the **declarations** array in the app.module.ts file, just like we do with components and directives. This ensures that the custom pipe is recognized and can be used throughout the application.

```
------------------------ shorten.pipe.ts ------------------------
import { PipeTransform, Pipe } from '@angular/core';
@Pipe({
    name: 'shorten'
})
export class ShortenPipe implements PipeTransform {
    transform(value: any) {
        if (value.length > 10)
            return value.substr(0, 10) + '...';
        return value;
    }
}
----------------------- demo.component.ts -----------------------
<div class="server-info">
    <strong>{{ server.name | shorten }}</strong>
    <span>{{ server.type | uppercase }}-{{ server.start | date }}</span>
</div>
```

[Note: The **transform** method is pipe built-in functional method. **]**

7. **Parametrizing a Custom Pipe:** In Angular, a custom pipe can accept parameters to further modify or customize its behavior. To pass parameters to a custom pipe, simply add them after the pipe's name in the template. Inside the pipe's transform method, you can handle these parameters and apply the desired logic.

```
------------------------ shorten.pipe.ts ------------------------
export class ShortenPipe implements PipeTransform {
    transform(value: any, limit: any, args: any) {
        if (value.length > limit)
            return value.substr(0, limit) + args;
        return value;
    }
}
----------------------- demo.component.ts -----------------------
<div class="server-info">
    <strong>{{ server.name | shorten:12:'Extra' }}</strong>
</div>
```

8. Create a **Custom Filter Pipe** to understand more →

```
----------------------- filter.pipe.ts ------------------------
@Pipe({
    name: 'filter'
})
export class FilterPipe implements PipeTransform {
  transform(value: any, filterString: string, propName?: string): any {
    if (value.length === 0 || filterString === '') return value;
    const resultArray = [];
    for (const item of value)
       if (item[propName] === filterString)
          resultArray.push(item);
    return resultArray;
  }
}
----------------------- demo.component.ts -----------------------
<input class="search" type="text" [(ngModel)]="filteredText" />
<div class="server-info">
    <ul class="list-group">
        <li class="list-group-item" *ngFor="let server of servers |
filter:filteredText:'status'">
            <strong>{{ server.name | shorten:12:'Extra' }}</strong>
        </li>
    </ul>
</div>
```

[Note: Ensure that we have added the FilterPipe into the module.ts file. **]**

9. **Re-running Pipes in Angular:** By default, Angular pipes are **`pure`**, meaning they do not automatically re-run the data changes. This is a performance optimization to prevent unnecessary recalculations. However, if we need a pipe to re-execute when data changes, we can set the **`pure`** option to **`false`**.

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

10. **Async Pipe:** It is designed to work with both observables and promises. It automatically subscribes to the observable or promise and handles the work for us. Once the observable emits a value or the promise is resolved, the async pipe updates the view with the new value.

```
------------------------ filter.pipe.ts -------------------------
appStatus = new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('stable');
    }, 2000);
});
------------------------ demo.component.ts -----------------------
<h2>App Status: {{ appStatus | async }}</h2>
<li *ngFor="let course of (allCources | async)"></li>
```

[Note: Here allCources receive observable type values.**]**

11. We can learn more pipes from docs → https://angular.io/api?query=pipe

Http API Call

1. To enable API call functionality, the first step is to add **HttpClientModule** to the **imports** array in the app.module.ts file.
2. **POST** request & **GET** request →

```
------------------------ post.component.ts -----------------------
export class PostComponent {
    constructor(private http: HttpClient) { }

    onCreatePost(postData: { title: string, content: string }) {
        this.http.post(API_URL, postData).subscribe(responseData => {
            console.log(responseData);
        });
    }
    onFetchPost() {
        this.http.get(API_URL)
            .subscribe(responseData => { console.log(responseData); });
    }
}
```

3. RxJS Operators to **Transform Response Data** →

```
import { map } from 'rxjs/operators';
onFetchPost() {
    this.http.get(API_URL).pipe(map(responseData => {
            // Our logical code
            // Note: Make sure we return value from here.
        })).subscribe(responseData => { console.log(responseData); });
}
```

4. Using Type with the HttpClient →

```
onCreatePost(postData: { title: string, content: string }) {
    this.http.post<{ name: string }>(API_URL, postData)
        .subscribe(responseData => {
            console.log(responseData);
        });
}
```

5. **DELETE** request →

```
onClearPosts() {
    this.http.delete(API_URL)
        .subscribe(responseData => { this.allPosts = []; });
}
```

6. Handle **errors** & use **Subject** to do this best way →

```
------------------------- post.service.ts -------------------------
import { Subject } from 'rxjs';
@Injectable({ providedIn: 'root' })
export class PostService implements OnInit {
    error = new Subject<string>();
    onFetchPost() {
        this.http.get(API_URL)
            .subscribe(responseData => {
                console.log(responseData);
            }, error => {
                console.log(error.message);
            });
    }
}
------------------------ post.component.ts ------------------------
private errorSub: Subscription;

ngOnInit() {
    this.errorSub = this.PostService.error
        .subscribe(errorMessage => {
            this.error = errorMessage;
        });
}
ngOnDestroy() {
    this.errorSub.unsubscribe();
}
```

S M HEMEL
Senior Software Engineer
Email: smhemel.eu@gmail.com

7. Use the **catchError** operator of RxJS to handle error →

```
import { map, catchError, throwError } from 'rxjs/operators';
onFetchPost() {
  this.http.get(API_URL).pipe(map(responseData => {
          // Our logical code and make sure we return value from here.
        }),
        catchError(errorRes => { return throwError(errorRes); })
     ).subscribe(responseData => { console.log(responseData); });
}
```

8. **Set Header** →

```
onFetchPost() {
  this.http.get(API_URL, {
       headers: new HttpHeaders({ 'Custom-Header': 'hello' })
    }).subscribe(
       responseData => { console.log(responseData); },
       error => { this.error.next(error.message);
    });
}
```

9. Add **Query Parameter** →

```
onFetchPost() {
    this.http.get(API_URL, {
        headers: new HttpHeaders({ 'Custom-Header': 'hello' }),
        params: new HttpParams().set('filter': 'new')
    }).subscribe(
         responseData => { console.log(responseData); },
         error => { this.error.next(error.message);
    });
}
------------------------------- OR -------------------------------
let searchParams = new HttpParams();
searchParams.append('filter': 'new');
searchParams.append('custom': 'key');

this.http.get(API_URL, {
    headers: new HttpHeaders({ 'Custom-Header': 'hello' }), params: searchParams
  }).subscribe(
    responseData => { console.log(responseData); },
    error => { this.error.next(error.message);
});
```

10. Observe different types of Response Data →

```
// Get full API response
this.http.post<{ name: string }>(API_URL, postData, { observe: 'response' })
    .subscribe(responseData => { console.log(responseData); });

// Handle event
this.http.delete(API_URL, { observe: 'events' })
  .pipe(tap(event => {
     if (event.type === HttpEventType.Sent) { }
     if (event.type === HttpEventType.Response) { console.log(event.body); }
  }))
  .subscribe(responseData => { this.allPosts = []; });
```

11. Change **Response Body Type** when we get data from API →

```
this.http.get(API_URL, {
    headers: new HttpHeaders({ 'Custom-Header': 'hello' }),
    params: searchParams,
    responseType: 'text' // json | blob | image
 }).subscribe(responseData => {
      console.log(responseData);
 });
```

Interceptors

1. **What are interceptors?** → Interceptors are a powerful feature in Angular that allow us to modify or handle HTTP requests and responses globally before they are sent or after they are received. For example, if we want to append a custom header (such as an authentication token) to every outgoing HTTP request, we can use an interceptor to do this automatically.
   Here's a general overview of interceptors:
   - **Global Modification:** Interceptors provide a way to globally modify HTTP requests (e.g., adding headers) or responses (e.g., logging or error handling) without changing the logic for each individual request.
   - **Middleware-Like Functionality:** They act like middleware, processing HTTP requests and responses before the request reaches the server and before the response reaches the client.
   - **Multiple Interceptors:** You can chain multiple interceptors to handle different tasks like logging, authentication, caching, etc.

2. Interceptor is basically a service. Now, we see **how it works and configure** →

```
------------------- auth-interceptor.service.ts ------------------
export class AuthInterceptorService implements HttpInterceptor {
    intercept(req: HttpRequest<any>, next: HttpHandler) {
        // Here's code will run before API request leave
        return next.handle(req);
    }
}
```

[Note: If we don't return next.handle(req), the request will not continue.**]**

```
--------------------- app-routing.module.ts ---------------------
@NgModule({
    imports: [RouterModule.forRoot(appRoutes)],
    providers: [{
        provide: HTTP_INTERCEPTORS,
        useClass: AuthInterceptorService,
        multi: true,
    }]
})
```

[Note: Set **`multi`** as **`true`**, if we have multiple interceptors under the **HTTP_INTERCEPTORS** identifier.**]**

3. **`How Interceptors Work?`** → Once the interceptor is configured, Angular automatically captures all HTTP interceptors and executes their **intercept** method whenever an HTTP request leaves the application. This gives us the ability to modify or inspect the request object before it is sent to the server.
However, it's important to note that the request object is **immutable**. This means the request cannot be altered directly. If we want to modify something, such as the request URL or headers, we must **clone** the request object and make changes. The cloned request is then passed along for further handling.
4. Manipulate Request Object →

```
export class AuthInterceptorService implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler) {
    const modifiedHeader = req.clone({
      headers: req.headers.append('Custom-Header': 'hello')
    });

    return next.handle(modifiedHeader);
  }
}
```

5. Response Interceptors →

```
intercept(req: HttpRequest<any>, next: HttpHandler) {
  return next.handle(req)
    .pipe(
       tap(event => {
          if (event.type === HttpEventType.Response) {
             console.log('Response arrived, body data: ' + event.body);
          }
       })
    );
}
```

6. Multiple Interceptors →

```
----------------- logging-interceptor.service.ts -----------------
export class LoggingInterceptorService implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler) {
     return next.handle(req)
         .pipe(tap(event => {
              if (event.type === HttpEventType.Response) {
                 console.log('Response arrived ' + event.body);
              }
         })
     );
  }
}
-------------------- app-routing.module.ts --------------------
@NgModule({
    imports: [RouterModule.forRoot(appRoutes)],
    providers: [
        {
            provide: HTTP_INTERCEPTORS,
            useClass: AuthInterceptorService,
            multi: true,
        },
        {
            provide: HTTP_INTERCEPTORS,
            useClass: LoggingInterceptorService,
            multi: true,
        }
    ]
})
```

Dynamic Components

1. **`What is the dynamic component?`** → Dynamic component is a component created dynamically at runtime. For example, we might want to display an alert, modal, or overlay that only loads in response to certain actions. Dynamic component creation and behavior are not built-in Angular features but are implemented based on the application's requirements.
2. Create an **Alert Modal** component manually → github-gist
3. **`Prepare Alert Modal programmatically`** → While it is possible to manually instantiate a component in TypeScript (e.g. `const alertCmp = new AlertComponent()`), this approach isn't valid in Angular. Angular provides us a more structured way to create components dynamically at runtime using a **ComponentFactory**. This tool allows us to generate and insert components programmatically. To illustrate this concept, we have provided an example code in the link below - github-gist
4. **`entryComponents`** → It was traditionally an array in the module file where components, directives, or pipes were listed when they were created programmatically, such as in a modal. Angular would automatically check the **`declarations`** array for components used as HTML selectors or routing components. However, for components instantiated dynamically (e.g., modals), the component had to be included in the **`entryComponents`** array. Starting with Angular 9, this is no longer required, as Angular now automatically detects components that need to be created programmatically without the **`entryComponents`** array.

Modules & Optimizations

Module

1. **`What are Modules?`** → In Angular, a module is a way to organize and bundle together building blocks like components, directives, services, and pipes. Angular doesn't automatically detect all files in a project, so we need to explicitly declare and group these elements into modules. This helps Angular recognize and manage these features, ensuring they work together effectively within the application. Modules are essential for organizing an Angular project and enabling the reuse of its parts across different areas of the application.

2. In Angular, a **module** is composed of several key arrays that help organize various elements(e.g. components, services, directives, pipes etc.) needed to build an app:
   - **`Declarations`**: Here, we declare components, directives, and custom pipes used within the module.
   - **`Imports`**: This array is for importing other modules required for the current module to function.
   - **`Providers`**: It lists all services that the module depends on. To make a service globally available, we can use **`@Injectable({providedIn: 'root'})`** instead of adding it to providers.
   - **`Exports:`** This array contains components, directives, pipes, or modules that can be used outside of this module. Services do not need to be exported.
   - **`Bootstrap array:`** Defines the component that should be bootstrapped when the app starts, usually tied to the root component in the **`index.html`** file.

   `Important:` Each module functions independently, without direct communication with others. They rely on explicit imports and exports to share functionality across different parts of the app.

3. To optimize code and improve performance, it's recommended to split modules based on specific features of the project. For each feature module:
   - Declare the **components**, **services**, **directives**, and **pipes** related to that feature within the corresponding module file.
   - Import any **necessary modules** required to work with these components, services, directives, and pipes in the same module file.

   This modular approach keeps the code organized, enhances reusability, and helps with lazy loading for performance optimization, ensuring that only the necessary code is loaded when needed.

4. **`Shared Module`** → In situations when a module, component, directive or pipe is required across multiple features, we create a shared.module.ts file. In this file, we import and declare all the common elements. Then, we import this shared module into the feature module files where it is needed.

   `Super Import:` A component can only be declared in one module. However, a shared module can be imported into multiple modules without declaring its components, ensuring reusability across the applicaton.

5. Add Routes to a Feature Module →

```
-------------------- app-routing.module.ts --------------------
// Before splitting up
const appRoutes: Routes = [
    { path: '', redirectTo: '/recipes', pathMatch: 'full' },
    {
      path: 'recipes', component: RecipesComponent,
      canActivate: [AuthGuard], children: [
        { path: '', component: RecipeStartComponent },
        {
          path: ':id', component: RecipeDetailsComponent,
          resolve: [RecipesResolverService]
        },
        {
          path: ':id/edit', component: RecipeEditComponent,
          resolve: [RecipesResolverService]
        }
      ]
    },
    { path: 'not-found', component: NotFoundComponent },
    { path: '**', redirectTo: '/not-found', pathMatch: 'full' }
];
-------------------- recipes-routing.module.ts -------------------
//  After splitting up, we move the recipe route part from app-routing.module.ts to recipes-routing.module.ts file.
const routes: Routes = [
    {
        path: 'recipes', component: RecipesComponent,
        canActivate: [AuthGuard], children: [
            { path: '', component: RecipeStartComponent },
            {
              path: ':id', component: RecipeDetailsComponent,
              resolve: [RecipesResolverService]
            },
            {
              path: ':id/edit', component: RecipeEditComponent,
              resolve: [RecipesResolverService]
            }
        ]
    }
];
```

```
@NgModule({
    imports: [RouterModule.forChild(routes)], exports: [RouterModule]
})
----------------------- recipes.module.ts -----------------------
@NgModule({ imports: [ RecipesRoutingModule ] })
```

6. **`Core Module`** → A Core Module is created to house services that are intended to be provided throughout the application, rather than declaring them in the App Module. This approach helps in organizing global services, ensuring that they are available application-wide but are only instantiated once.

   After creating the Core Module (commonly named **core.module.ts**), it is imported into the **app.module.ts** file. This structure makes it easier to manage and separate core functionality, improving the overall maintainability of the app.

Lazy Loading

`What is lazy-loading?` → Lazy-loading refers to loading specific modules or components of an application only when they are required, rather than loading everything upfront. For example, in an Angular app with routes like **`/`**, **`/products`**, and **`/admin`**, each route may have its own module containing components, services, and more. When lazy-loading is implemented, only the module related to the visited route is loaded. If the user accesses the **`/products`** route, only the components and features for that route are loaded, improving performance and reducing initial load time.

By loading modules on demand, the application has a smaller initial bundle size, leading to faster load times and better user experience.

1. **Enable Lazy-loading with Routing** →

```
-------------------- recipes-routing.module.ts --------------------
const routes: Routes = [
  {
    path: '/',
    loadComponent: () => import('./recipes/recipes.component').then(m => m.RecipesComponent),
    canActivate: [AuthGurad], children: [
      { path: '', component: RecipeStartComponent },
      { path: ':id', component: RecipeDetailsComponent,
        resolve: [RecipesResolverService]
      }
    ]
  }
];
```

```
--------------------------- app.module.ts ---------------------------
const appRoutes: Routes = [
  // { path: '', component: HomeComponent },
  { path: '', redirectTo: '/recipes', pathMatch: 'full' },
  // Add below line to enable lazy-loading
  {
    path: 'recipes',
    loadChildren: './recipes/recipes.module#RecipesModule'
  }
  // Alternatively
  // For newer version of Angular
  {
    path: 'recipes',
    loadChildren: () => import('./recipes/recipes.module').then(m => m.RecipesModule)
  }
];
```

[Note: When implementing lazy-loading, there's no need to manually import the module into the **`imports`** array of the **`app.module.ts`** file. Angular handles the module loading dynamically when the relevant route is accessed.**]**

2. **Preloading Lazy-Loaded Code →**

```
--------------------- app-routing.module.ts ---------------------
import { RouterModule, PreloadAllModules } from '@angular/router';

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

[Note: The advantage of lazy-loading is that it keeps the initial download bundle small, as only the necessary code is loaded. This results in a faster initial loading phase and quicker subsequent loads, improving overall performance and user experience.**]**

Services & Module

`What's special about services and Modules?` → These are closely connected to lazy-loading in Angular. Let's explore where we can provide services and how it impacts the instances of the services we're working with.

• Services can be provided in the app component or other components or modules.

• Services can be added to the providers of an **eager-loaded** module (Like a Recipes Module before we added lazy-loading) or a **Lazy-loaded** module.

• Using **providedIn: 'root'** in the **@Injectable** decorator makes the service available application-wide and is recommended for services intended for the AppModule.

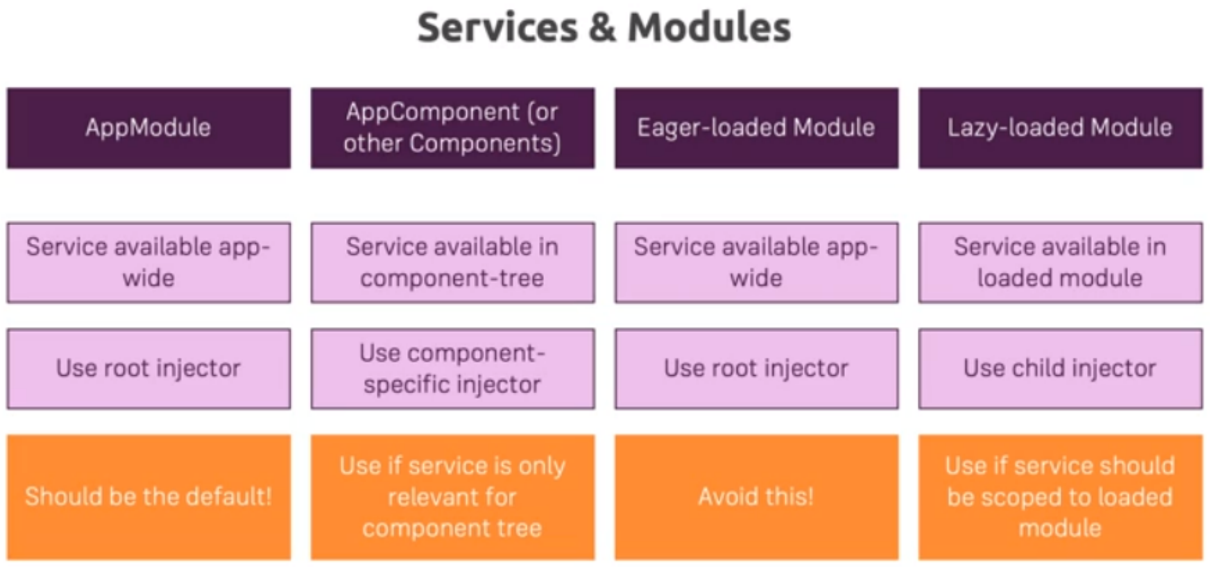
Service Instances Based on Location and how it affects service instances:

• **Eager-loaded Module**: If a service is added to an eager-loaded module's providers, it is only available inside that module.

• **Lazy-loaded Module**: If added to a lazy-loaded module, the service is available only in that module and will have its own instance, separate from the rest of the app.

• **App Module & Lazy-loaded Module**: If a service is provided both in the app module and a lazy-loaded module, the service becomes application-wide, but the lazy-loaded module will get a separate instance.

<u>`Important`</u>`:` Avoid adding a service in an eager-loaded module as it might lead to unexpected behavior or multiple instances of the service.



Services & Modules

AppModule	AppComponent (or other Components)	Eager-loaded Module	Lazy-loaded Module
Service available app-wide	Service available in component-tree	Service available app-wide	Service available in loaded module
Use root injector	Use component-specific injector	Use root injector	Use child injector
Should be the default!	Use if service is only relevant for component tree	Avoid this!	Use if service should be scoped to loaded module

S M HEMEL
Senior Software Engineer
Email: smhemel.eu@gmail.com

Deployment

1. Ensure all necessary environment variables are suitably set for the environment.
2. Build Angular apps for deployment by running → `ng build`
3. For Firebase deployment, execute → `npm install -g firebase-tool`. Afterward, log in to Firebase with → `firebase login`.
   Next, initialize Firebase for project → `firebase init` and choose **Hosting**.
   Set the public directory as **dist/project_directory_name**.
   Finally, deploy with → `firebase deploy`.
4. To address broken routes after deployment, refer to → Fix Broken Route.

Standalone Components

Angular 14 introduced **Standalone Components**, which became stable from Angular 15 onwards. This feature simplifies Angular development by reducing the boilerplate code required when setting up a new project or refactoring existing ones. Traditionally, Angular developers had to declare components, directives, and pipes in modules and manage imports and exports between them, which could be cumbersome in larger apps.

Standalone components allow us to build Angular apps **without the need for NgModules**. These components can now operate independently, with services, directives, and pipes being injected directly into them without the need to declare them in a module.

Key Benefits: Reduced Boilerplate Code, Simplified Structure, Backward Compatibility, Improved Developer Experience.

1. Create a Standalone Component →

```
import { Component } from '@angular/core';
@Component({
    standalone: true,
    selector: 'app-xyz',
    templateUrl: './xyz.component.html',
    styleUrls: ['./xyz.component.scss']
})

export class DemoComponent {}
```

Important: When creating a standalone component, it's crucial not to declare it in any **NgModules**. If the component has already been added to the **declarations** array, it should be removed. However, simply removing the declaration is not enough to integrate the component into the project. Angular still needs to be made aware of the component, and there are two ways to achieve this:

- ➔ Add the standalone component to the **imports** array of app.module.ts or any specific module.
- ➔ Include the standalone component in the **imports** array of another standalone component that will use it.

```
@Component({
    standalone: true,
    imports: [ DemoComponent ]
    selector: 'app-xyz',
    templateUrl: './xyz.component.html',
    styleUrls: ['./xyz.component.scss']
})
export class DemoParentComponent {}
```

2. After adding a standalone component to the **imports** array of an **NgModule** file or another standalone component file, the standalone component remains unaware of other components, directives, modules, and pipes, and therefore cannot utilize them. To enable the use of these elements, it is necessary to import the relevant components, directives, modules, and pipes into the **imports** array of the standalone component.

Note: If a component, directive, or pipe is declared within a module importing the module into the **imports** array of the standalone component is sufficient, eliminating the need to import each individual component, directive, or pipe.

```
@Component({
    standalone: true,
    imports: [SharedModule]
    selector: 'app-xyz',
    templateUrl: './xyz.component.html',
    styleUrls: ['./xyz.component.scss']
})
export class DemoComponent {}
```

3. Standalone Root Component →

```
------------------------ app.component.ts -----------------------
@Component({
    standalone: true,
    imports: [MainComponent, NavbarComponent]
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.scss']
})
export class AppComponent {}
------------------------- app.module.ts -------------------------
// For standalone root components, we don't need the app.module.ts. We just add this here, for example, the purpose of how to configure it.
@NgModule({
    declarations: [],
    imports: [BrowserModule, AppComponent],
    providers: [],
    bootstrap: [AppComponent],
})
---------------------------- main.ts ----------------------------
import { enableProdMode } from '@angular/core';
import { bootstrapApplication } from '@angular/platform-browser';
import { environment } from './environments/environment';

if (environment.production) { enableProdMode(); }
bootstrapApplication(AppComponent);
```

Important: After transforming the root component into a standalone component, there is no longer a need for the **app.module.ts** file. The **BrowserModule** is accessible through the **Bootstrap** application, rendering the **app.module.ts** file unnecessary. The **Bootstrap** application seamlessly manages all functionalities of the app module. This represents a significant advantage of using standalone components.

4. Services →

```
------------------------ demo.service.ts ------------------------
// @Injectable({providedIn: 'root'})
export class DemoService {}
---------------------------- main.ts ----------------------------
bootstrapApplication(AppComponent, { providers: [ DemoService ] });
```

Important: Instead of using **providedIn**, that is another way to provide a service globally when creating standalone root components.

5. Routing →

```
------------------------ app.component.ts ------------------------
@Component({
    standalone: true,
    imports: [MainComponent, NavbarComponent, RouterModule]
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.scss']
})
export class AppComponent {}
---------------------------- main.ts ----------------------------
import { importProvidersFrom } from '@angular/core';
import { bootstrapApplication } from '@angular/platform-browser';
bootstrapApplication(AppComponent, {
    providers: [ DemoService, importProvidersFrom(AppRoutingModule) ]
});
```

6. Load Component Lazily →

```
const appRoutes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'users/:id',
    loadComponent:() => import('./user/detail.component').then(m => m.DetailComponent)
  },
  { path: 'recipes',
    loadChildren: () => import('./recipes/recipes.module').then(m => m.RecipesModule)
  }
];
```

[Note: This loadComponent works only with the standalone component.**]**

Important: We can load individual components lazily without having to wrap them in a module.

7. We can add routing part of a feature without using a specific routing module →

```
---------------------------- routes.ts ----------------------------
// Create this routes.ts file after removing/instead of dashboard-routing.module.ts file
export const DASHBOARD_ROUTES: Route[] = [
  { path: '', component: DashboardComponent },
  { path: 'today', component: TodayComponent }
];
```

```
-------------------- app-routing.module.ts --------------------
const appRoutes: Routes = [
  { path: '', component: HomeComponent },
  {
    path: 'users/:id',
    loadComponent: () => import('./user/user-details.component').then(m =>
m.UserDetailsComponent)
  },
  {
    path: 'dashboard',
    loadChildren: () => import('./dashboard/routes').then(m => m.DASHBOARD_ROUTES)
  }
];
```

8. **Very Very Important:** When incorporating elements into a standalone component that relies on any module, component, directive, or pipe, it's imperative to import the relevant module, component, directive, or pipe into the **imports** array of the standalone component. For example, if we use **routerLink** in the HTML file of a standalone component, which relies on **RouterModule**, it is essential to include **RouterModule** to the **imports** array to ensure proper functionality.

Angular Universal / Angular SSR

What is Angular Universal, and Why does it matter?

When we build a standard angular app, we get a client-side app because Angular is a client-side JavaScript framework. This means that the code we write eventually runs in the browser. While this has some advantages, it also has potential drawbacks. One significant disadvantage is its impact on search engine optimization(SEO). When building an Angular app, web search crawlers (like Google) can't access the content of our website—they essentially see an empty page, resulting in poor SEO performance.

Another potential drawback is slower page loading, especially on slow networks or devices. The page and its content may take time to load because all the JavaScript code must be downloaded before anything is rendered on the screen.

Angular Universal helps address these issues. Essentially, Angular Universal is an additional package—a set of extra settings and features that add server-side functionality to our Angular app. With Angular Universal enabled, the app is still a client-side single-page application once downloaded from the server. However, the key difference is that the very first page loaded by the user is pre-rendered on the server. This means the fully-rendered page is served to the user right away, so we don't have to wait for the JavaScript code to download and execute before seeing content on the screen.

It's important to note that this optimization applies only to the initial page load. After that, subsequent navigations within the app are still handled by Angular on the client side, maintaining the single-page application experience with the added benefit of faster first-page load times.

1. With Angular 17, "**Angular Universal**" was essentially renamed to "**Angular SSR**" (SSR stands for Server-side Rendering). However, the core concept remains the same in both cases: the Angular app is pre-rendered on the server, and the finished HTML is sent to the client. Afterward, the app transitions back into a client-side SPA (Single Page Application).

Installation & Configuration

1. If we are working on an Angular 16 or older project, then run the following command to enable Angular Universal/SSR into our project.
   → **`ng add @nguniversal/express-engine`**

   But, if we're working on an Angular 17 project, then run
   → **`ng add @angular/ssr`**

2. After running the **Angular Universal** or **SSR** enable command, 4 new files will be created automatically and also updating the existing app-routing module file. Created files are → main.server.ts, app.server.module.ts, tsconfig.server.json, server.ts.
3. **Creating Angular SSR Apps with Angular 17+**, by running
   → **`ng new <project-name> --ssr`**
   Note: Using Angular 17, running an SSR-enabled project also is easier than doing so with Angular 16 or lower.

Deployment

In the Angular Universal app, we could create a REST API into the server.ts file, allowing us to build a full-stack app with a backend REST API that resides within the same project and directory. An Angular Universal/SSR app is essentially a Node.js application using the Express framework. Therefore, we can search for Node.js hosting providers and find a wide variety of options for deploying our application.

To deploy our app, follow these steps:

- Run the command → `npm run build:ssr`
- After running the command, we get a **dist** folder containing both **browser** and **server** subfolders, each named after our project. The entire **dist** folder needs to be uploaded to the hosting provider for deployment.
- Copy three essential items(package.json, angular.json and dist directory) into a new folder where we will prepare the project for deployment.
- Run the command → `npm install`
- Verify the project works properly by running the command: `npm run serve:ssr`
- Once verified, we can deploy it to the hosting provider's machine.

Important Note: we **MUST NOT** use any browser-specific APIs like **document.querySelector()** in our Angular code, as they will be executed on the server, where such APIs are unavailable. It is recommended to rely only on Angular's built-in features for server-side rendering.

Angular Signals

Angular 16 introduced a very important new feature called **Signals**, which is stable in **Angular 17**.

What's the purpose of Signals? → Signals provide a new way of handling data changes and updating the UI. Traditionally, Angular uses the **zone.js** library, which automatically detects changes and updates the UI accordingly. However, this approach can negatively impact app performance and increase bundle size.

That's why **Signals** come in. They offer an alternative method for managing and detecting changes, allowing Angular to eliminate the need of **`zone.js`** library. This change let's Angular stop monitoring the entire application, which can improve performance and reduce bundle size. With Signals, there is no automatic change detection. Instead, we explicitly tell Angular when data changes and where that data is being used.

1. Create & Update a new Signal →

```
--------------------- signals.component.ts ---------------------
import { Component, signal } from '@angular/core';
@Component({
    standalone: true,
    imports: [SharedModule]
    selector: 'app-signals,
    templateUrl: './signals.component.html',
    styleUrls: ['./signals.component.scss']
})

export class SignalsComponent {
    counter = signal(set_default_value_here);
    increment() { this.counter.update((oldCounter) => oldCounter + 1); }
}
--------------------- signals.component.html --------------------
<div class="container">
    <p> Output Value: {{ counter() }} </p>
</div>
```

2. Different update methods →

```
--------------------- signals.component.ts ---------------------
actions = signal<string[]>([]);
counter = signal(set_default_value_here);

increment() {
   this.counter.update((oldCounter) => oldCounter + 1);
   this.counter.set(5);
   this.counter.set( this.counter() + 5);
   this.counter.mutate((oldActions) => oldActions.push('Increment'));
}
--------------------- signals.component.html --------------------
<div class="container">
    <h2> Signals </h2>
    <li *ngFor="let action of actions()"></li>
</div>
```

[Note: Both set and update assign a new value to a Signal. The mutate allows us to modify existing values, effectively assigning a new value by editing the current one.**]**

3. Computed Values →

```
---------------------- signals.component.ts ---------------------
import { Component, signal, computed } from '@angular/core';
counter = signal(set_default_value_here);
doubleCounter = computed(() => this.counter() * 2);
-------------------- signals.component.html ---------------------
<div class="container">
    <h2> Signals </h2>
    <p> Output Value: {{ doubleCounter() }} </p>
</div>
```

[Note: The **doubleCounter** behaves like a signal, meaning it recomputes whenever the **counter** changes and updates the UI accordingly. This makes **doubleCounter** dependent on another signal, in this case, **counter**.**]**

4. Effects →

```
import { Component, signal, effect } from '@angular/core';
export class SignalsComponent {
    constructor() {
        effect(() => console.log(this.counter()));
    }
}
```

[Note: The effect is a function that could be executed in the constructor and we can use it in other places. The effect allows us to define code that depends on signals and Angular will automatically re-execute this function that's being passed to effect, whenever any signal is being used inside of the function.
Angular understands that this code uses the counter signal and whenever the counter changes, only then angular will go ahead and re-execute this code.**]**

Angular Animations

With the release of Angular 4, the general syntax for Angular Animations remained unchanged.

Installation & Configuration

1. First we need to install the animations package by running the command:
   → `npm install --save @angular/animations`
2. Add the **BrowserAnimationsModule** to the **imports[]** of the **AppModule**. This module needs to be imported from the **@angular/platform-browser/animations** package.

3. Now, we can import **trigger**, **state**, **style**, and other animation-related functions from the **@angular/animations**.

Implementation

1. **Uses of Animations Trigger and State** → To use animations in component templates, we first need to set them up in the app component decorator.

```
----------------------- demo.component.ts -----------------------
import {
    Component, trigger,
    state, transition, animate
} from '@angular/core';

@Component({
    selector: 'app-demo',
    templateUrl: './demo.component.html',
    styleUrls: ['./demo.component.scss'],
    animations: [
        // We could write multiple trigger here separating by `,`
        trigger('divState', [
            state('normal', style({
                'background-color': 'red', 'transform': 'translateX(0)'
            })),
            state('highlighted', style({
                'background-color': 'blue', 'transform': 'translateX(100px)'
            })),
            transition('normal => highlighted', animate(300)),
            transition('highlighted => normal', animate(800))
            // transition('normal <=> highlighted', animate(300))
            // This works in both(left-right, right-left) ways.
        ])
    ]
})
export class DemoComponent {
    state = 'normal';
    // Switching between states

    onAnimate() {
        this.state = this.state === 'normal' ? 'highlighted' : 'normal';
    }
}
```

```
---------------------- demo.component.html ----------------------
<div class="row">
    <div class="col-xs-12">
        <button (click)="onAnimate()">Animate!</button>
    </div>
    <div style="width: 100px; height: 100px" [@divState]="state"> </div>
</div>
```

2. Advance Transitions →

```
----------------------- demo.component.ts -----------------------
@Component({
    selector: 'app-demo',
    templateUrl: './demo.component.html',
    animations: [
        trigger('wildState', [
            state('normal', style({
                'background-color': 'red',
                'transform': 'translateX(0) scale(1)'
            })),
            state('highlighted', style({
                'background-color': 'blue',
                'transform': 'translateX(100px) scale(1)'
            })),
            state('shrunken', style({
                'background-color': 'green',
                'transform': 'translateX(0px) scale(0.5)'
            })),
            transition('normal => highlighted', animate(300)),
            transition('highlighted => normal', animate(800))
            transition('shrunken <=> *', animate(500)) // Use * means any
state, 'shrunken' to any state and any state to 'shrunken'
        ])
    ]
})
export class DemoComponent {
  wildState = 'normal';
  onAnimate() {
    this.wildState = this.wildState === 'normal' ? 'highlighted' : 'normal';
  }
  onShrink() { this.wildState = 'shrunken'; }
}
```

```
---------------------- demo.component.html ----------------------
<div class="row">
  <div class="col-xs-12">
    <button (click)="onShrink()">Shake!</button>
  </div>
  <div style="width: 100px; height: 100px" [@wildState]="state"> </div>
</div>
```

3. **Transition Phases** → We can write style properties into the **animate** method.

```
transition('shrunken <=> *', animate(50, style({ 'borderRadius': '50px' })))
```

4. **The "void" state →**

```
----------------------- demo.component.ts -----------------------
@Component({
  selector: 'app-demo',
  templateUrl: './demo.component.html',
  animations: [
    trigger('wildState', []),
    trigger('list1', [
      state('in', style({ opacity: 1, transform: 'translateX(0)' })),
      transition('void => *', [
        style({ opacity: 0, transform: 'translateX(-100px)' }), animate(300)
      ]),
      transition('* => void', [
        animate(300, style({ opacity: 0, transform: 'translateX(100px)' }))
      ]),
    ]),
  ]
})

---------------------- demo.component.html ----------------------
<ul class="list-group">
   <li [@list1] (click)="onDelete()" *ngFor="let item of items">
       {{ item }}
   </li>
</ul>
```

[Note: The **"void"** is a reserve state name. We can't use it. We shouldn't override it based on our requirement.**]**

5. Use keyframe & grouping for Animations →

```
---------------------- demo.component.ts ----------------------
import { Keyframes, group } from '@angular/core';
@Component({
  selector: 'app-demo',
  templateUrl: './demo.component.html',
  animations: [
    trigger('list2', [
      state('in', style({ opacity: 1, transform: 'translateX(0)' })),
      transition('void => *', [
         animate(1000, keyframes(
           style({ opacity: 0, transform: 'translateX(-100px)' }),
           style({ opacity: 0.5, transform: 'translateX(-50px)' }),
           style({ opacity: 1, transform: 'translateX(-20px)' })
         ))
      ]),
      transition('* => void', [
         group([
           animate(300, style({ color: 'red' })),
           animate(800, style({ opacity: 0, transform: 'translateX(100px)' }))
         ])
      ]),
    ]),
  ]
})
--------------------- demo.component.html ---------------------
<ul class="list-group">
  <li [@list2] (click)="del()" *ngFor="let item of items"> {{ item }} </li>
</ul>
```

6. **Use Animation Callback →**

```
--------------------- demo.component.html ---------------------
<div class="row">
  <button (click)="onShrink()">Shake!</button>
  <div style="width: 100px; height: 100px" [@divState]="state"
       (@divState.start)="animationStarted($event)"
       (@divState.done)="animationEnded($event)">
  </div>
</div>
```

Offline Capabilities with Service Workers

What are Service Workers? → Service workers can intercept outgoing requests for web assets like CSS, fonts, JavaScript, or data from an API. They can cache these responses in special cache storage and return the cached versions to the page, allowing the app to function offline if a cached version exists. Essentially, a service worker acts as a proxy between the front-end app and the HTTP requests sent to the backend. It intercepts requests, caches them, and can either let the request proceed or block it, depending on the conditions.

1. To enable service worker functionality, first, install the necessary package by running the command: **`ng add @angular/pwa`**.
   → This command will create new files and update some existing ones(particularly app.module.ts) to configure the service worker.
   → In the app.module.ts file, make sure to add the **ServiceWorkerModule** to the **imports[]** array.

```
import { ServiceWorkerModule } from '@angular/service-worker';
@NgModule({
  imports: [
    ServiceWorkerModule.register('/ngsw-worker.js', { enabled:
environment.production })
  ]
})
```

2. **Caching assets for Offline Use** → To cache and load data for offline use, we need to ensure that key assets, such as the `index.html` file (which sits at the root of the server folder), are cached. In the `ngsw-config.json` file, we find two asset group arrays. These asset groups define which static assets should be cached and how they should be cached. Dynamic assets, such as data from an API, are different—they change frequently, so they are not considered static.
   Within the **`assetGroups`** array, you'll see **`installMode`** and **`updateMode`** properties, where you can set values like **`"prefetch"`** or **`"lazy"`** to control how assets are loaded. There is also a **`resources`** object, which contains an array specifying the files you want to cache.
   If you want to cache assets like fonts via a link, you can add a `urls` array inside the **`resources`** object, alongside the **`files`** array, and include the link in the **`urls`** array.

3. **Caching Dynamic assets and urls** → To cache dynamic assets and urls, we need to create a **dataGroups** object array just like **assetGroups**.

```
"dataGroups": [
    {
        "names": "posts",
        "urls": [ "https://xyz.com/posts" ],
        "cacheConfig": {
           "maxSize": "5",
           "maxAge": "6h",
           "timeout": "10s",
           "strategy": "freshness", // Or "performance",
        }
    }
]
```

4. **Angular Service Worker Docs** → https://angular.io/guide/service-worker-intro

Unit Testing and E2E Testing

Always write testing-related code in files with a **.spec** extension. The **.spec** extension is essential because it allows the Angular CLI to recognize and compile the tests. After defining a test suite, we use Jasmine's **describe** function. A single test suite can contain multiple specifications, which define individual test cases.

```
---------------------- calculator.service.spec.ts ---------------------
describe('CalculatorService', () => {
    it('should add two numbers', () => { });
    it('should subtract two numbers', () => { });
});
------------------------ calculator.service.ts ------------------------
export class CalculatorService {
    add(n1: number, n2: number) {
        this.logger.log("Addition operation called");
        return n1 + n2;
    }

    subtract(n1: number, n2: number) {
        this.logger.log("Subtraction operation called");
        return n1 - n2;
    }
}
```

Command

1. To test our app run → **`ng test`**
2. Run test without hot loader → **`ng test --no-watch`**

Jasmine Testing Utility Methods

```
---------------------- calculator.service.spec.ts ---------------------
import { CalculatorService } from "./calculator.service";
import { LoggerService } from "./logger.service";

describe("CalculatorService", () => {
  it("should add two numbers", () => {
    const calculator = new CalculatorService(new LoggerService());
    const result = calculator.add(2, 2);
    expect(result).toBe(4);
  });

  it("should subtract two numbers", () => {
    const calculator = new CalculatorService(new LoggerService());
    const result = calculator.subtract(2, 2);
    expect(result).toBe(0);
  });
});
------------------------ calculator.service.ts ------------------------
export class CalculatorService {
  constructor(private logger: LoggerService) { }

  add(n1: number, n2:number) {
    this.logger.log("Addition operation called");
    return n1 + n2;
  }
  subtract(n1: number, n2:number) {
    this.logger.log("Subtraction operation called");
    return n1 - n2;
  }
}
-------------------------- logger.service.ts --------------------------
export class LoggerService {
  log(message:string) { console.log(message); }
}
```

S M HEMEL
Senior Software Engineer
Email: smhemel.eu@gmail.com

spyOn & createSpyObj

If we want to track how many times certain logger functions are called, we can use Jasmine's **spyOn** utility.

Alternatively, we can create fake dependencies instead of using actual service dependencies. When we use a fake dependency, there's no need to use **spyOn**, as the logger automatically tracks calls. This is the preferred approach.

Important Note: when using **createSpyObj**, we must include any function we intend to use in its parameter array; otherwise, it will result in an error.

```
---------------------- calculator.service.spec.ts ---------------------
describe("CalculatorService", () => {
  it("should add two numbers", () => {
    // const logger = new LoggerService();
    const logger = jasmine.createSpyObj("LoggerService", ["log"]);

    // spyOn(logger, "log");
    const calculator = new CalculatorService(logger);
    const result = calculator.add(2, 2);

    expect(result).toBe(4);
    expect(logger.log).toHaveBeenCalledTimes(1);
  });
});
```

beforeEach

To streamline our tests and avoid repeating code and execute necessary logic before each test, we should implement a **beforeEach** block. In this block, we include all the setup logic required for the individual **it()** test blocks. The code in **beforeEach** will execute before every **it()** block in the test suite.

```
---------------------- calculator.service.spec.ts ---------------------
describe("CalculatorService", () => {
  let calculator: CalculatorService, loggerSpy: any;

  beforeEach(() => {
    loggerSpy = jasmine.createSpyObj("LoggerService", ["log"]);
    calculator = new CalculatorService(loggerSpy);
  });

  it("should add two numbers", () => {
    const result = calculator.add(2, 2);
    expect(result).toBe(4);
    expect(loggerSpy.log).toHaveBeenCalledTimes(1);
  });
  it("should subtract two numbers", () => {
    const result = calculator.subtract(2, 2);
    expect(result).toBe(0);
  });
});
```

afterEach

To avoid code duplication and execute necessary cleanup after each test, we should implement an **`afterEach`** block. In this block, we include any logic that needs to run after the individual **`it()`** blocks. The code in **`afterEach`** will execute after every **`it()`** block in the test suite.

```
----------------------- courses.service.spec.ts -----------------------
describe("CoursesService", () => {
  afterEach(() => {
    httpTestingController.verify();
  });
});
```

[Note: The **verify()** method of the **HttpTestingController** ensures that only the HTTP requests specified within the **it()** block, using the expected APIs from the testing controller, are made by the service being tested.**]**

toBeTruthy()

This is used after the **`expect()`** block to ensure that a variable holds a truthy value. If the variable exists and has a value, **`toBeTruthy()`** will validate it.

```
----------------------- courses.service.spec.ts -----------------------
coursesService.findLessons(12)
  .subscribe((lessons) => {
     expect(lessons).toBeTruthy();
  });
```

TestBed

It allows us to configure services without directly assigning a service instance to a variable. It enables us to provide dependencies to services using Angular's dependency injection, eliminating the need to call constructors explicitly.

```
--------------------- calculator.service.spec.ts ---------------------
import { TestBed } from "@angular/core/testing";

beforeEach(() => {
  loggerSpy = jasmine.createSpyObj("LoggerService", ["log"]);
  TestBed.configureTestingModule({
    providers: [
      CalculatorService,
      { provide: LoggerService, useValue: loggerSpy },
    ],
  });
  calculator = TestBed.inject(CalculatorService);
});
```

Angular Service Testing

1. To add an Angular service, such as **`HttpClient`**, to a testing file, we must include the module-specific testing module(**`HttpTestingModule`**). This testing module provides a mock implementation of the service, which includes all the same methods as the real service.

```
-------------------- courses.service.spec.ts --------------------
describe('CoursesService', () => {
  let coursesService: CoursesService, httpTestController: HttpTestingController

  beforeEach(() => {
     TestBed.configureTestingModule({
        imports: [HttpClientTestingModule],
        declarations: [CoursesService]
     });

     coursesService = TestBed.inject(CoursesService);
     httpTestController = TestBed.inject(HttpTestingController);
  });
  it('Should retrieve all courses', () => {});
})
```

2. **req.flush()** - It triggers the mock HTTP call and simulates a response with mock data that resembles the original response.

```
-------------------- courses.service.spec.ts --------------------
describe('CoursesService', () => {
  it("should save the course data", () => {
    const changes: Partial<Course> = {
      titles: { description: "Testing Course" },
    };

    coursesService.saveCourse(12, changes)
      .subscribe((course) => { expect(course.id).toBe(12); });

    const req = httpTestingController.expectOne("/api/courses/12");
    expect(req.request.method).toEqual("PUT");

    expect(req.request.body.titles.description)
      .toEqual(changes.titles.description);

      req.flush({ ...COURSES[12], ...changes });
   });
})
```

Angular Component Testing

1.

Must know

1. To skip a test suite or a specific test block, we use **`xdescribe`** and **`xit`**. Conversely, to focus on a specific test suite or test block, we can use **`fdescribe`** and **`fit`**.

RxJS

RxJS stands for **Reactive Extensions for JavaScript**. It is a Third-party powerfull library used in Angular for handling asynchronous operations and managing events through observables and provides a wide range of JavaScript functions and classes, including **`Observable`**, **`map`**, **`filter`**, **`merge`**, **`concat`**, **`catchError`**, **`interval`**, etc.

Observable

It is a class interface provided by the RxJS library. It is used to manage asynchronous data streams, allowing developers to handle events, asynchronous operations, and data changes efficiently.

```
export class AppComponent {
    count$ = new Observable((observer) => {
        observer.next('Hello');
        observer.error('An error occurred');
        observer.complete();
    }).subscribe({
        next(value) { console.log(value); },
        error(err) { console.log(err); }
    });
}
```

1. Creating & Using an Observable →

```
------------------------ messages.service.ts ------------------------
export class MessagesService {
  messages$ = new BehaviorSubject<string[]>([]);
  private messages: string[] = [];
  addMessage(mes: string) {
    this.messages = [...this.messages, mes];
    this.messages$.next(this.messages);
  }
}
--------------------- messages-list.component.ts --------------------
export class MessagesListComponent implements OnInit {
  private messagesService = inject(MessagesService);
  ngOnInit() {
    this.messagesService.messages$.subscribe((message: string[]) => { });
  }
}
                        - INTERVAL OBSERVABLE -
ngOnInit() {
  const subscription = interval(1000).subscribe(() => {
    next: (val: number) => console.log(val);
  });
}
```

[Note: It will emit value after 1 second.]

S M HEMEL
Senior Software Engineer
Email: smhemel.eu@gmail.com

2. Creating & Using A Custom Observable From Scratch →

```
import { Observable } from 'rxjs';

export class AppComponent implements OnInit {
  customInterval$ = new Observable((subscriber) => {
    let timesExecuted: number = 0;
    const interval = setInterval(() => {
       if (timesExecuted > 3) {
          clearInterval(interval);
          subscriber.complete();
          return;
       } else if (timesExecuted < 1) {
          subscriber.error();
          return;
       }

       subscriber.next({ message: 'New Value' });
       timesExecuted++;
    }, 2000);
  })

  ngOnInit() {
    this.customInterval$.subscribe(() => {
       next: (val: any) => console.log(val);
       complete: () => console.log('Completed!!');
       error: () => console.log('Error!!');
    })
  }
}
```

Operator

1. **of()** → It allows us to create an Observable that emits a sequence of values.

```
-------------------------- demo.service.ts --------------------------
export class DemoService {
    constructor(private http: HttpClient) { }
    loadAllCourses(): Observable<Course[]> {
       return this.http.get<Course[]>(url)
           .pipe(shareReplay());
    }
}
```

S M HEMEL
Senior Software Engineer
Email: smhemel.eu@gmail.com

```
------------------------- demo.component.ts -------------------------
allCourses: Observable<Course[]>;
ngOnInit() {
   const allCourses = this.demoService.loadAllCourses();
   allCourses.subscribe(val => console.log(var));
}
```

2. **map()** → It transforms the values emitted by an observable. It takes a function as an argument, which is executed on every value emitted by the observable, allowing you to modify the data as it flows through the stream.

```
ngOnInit() {
    const subscription = interval(1000).pipe(
      map((val: number) => val * 2)
    ).subscribe(() => {
      next: (val: number) => console.log(val); // 0 2 4 6 8
    });
}
```

3. **tap()** → It executes code as we would do in subscribe, but without subscribing to the observable. It accepts an object where we define **next**, **complete**, **error** callblocks, allowing us to perform actions on emitted values, completion or errors without altering the data stream.

```
------------------------- demo.service.ts -------------------------
import { tap } from 'rxjs';

userPlaces: any = []
loadUserPlaces() {
  this.httpClient.get<{places: Place[]}>(API_URL)
     .pipe(map((resData: any) => resData.places),
         catchError((error) => throwError(() => new Error(error))
     ))
     .pipe(tap({ next: (userPlaces) => this.userPlaces = userPlaces; }));
}
```

4. **shareReply** → When we implement API call functionality in a method that returns an **`Observable`**, and we call this method in a component to retrieve data, subsequent subscriptions to the same variable (either directly or using an async pipe) can trigger additional API calls. To prevent this and ensure we can access the previously fetched data without making another API call, we use the **`shareReplay`** operator from RxJS. This operator shares the most recent

emitted value among multiple subscribers, effectively caching the result and replaying it to any new subscribers.

```
-------------------------- demo.service.ts --------------------------
export class DemoService {
    constructor(private http: HttpClient) { }
    loadAllCourses(): Observable<Course[]> {
       return this.http.get<Course[]>(url).pipe(shareReplay());
    }
}
------------------------- demo.component.ts -------------------------
allCourses: Observable<Course[]>;
ngOnInit() {
   const allCourses = this.demoService.loadAllCourses();
   allCourses.subscribe(val => console.log(var));
}
```

NgRx State Management

It's a state management solution. It is a package that can be installed to help manage application-wide state effectively. NgRx provides a standardized and clearly defined approach for handling application state, allowing developers to maintain a predictable state management system using **actions**, **reducers**, and **selectors**.

Project Setup & Install

1. Add NgRx → **`ng add @ngrx/store`**
2. After executing the command, the **StoreModule** will be automatically added to the app.module.ts file, establishing a global store for managing application wide state using **`StoreModule.forRoot({}, {})`** in the **imports[]** array.
3. When using NgRx, we will always have a single global store that manages our application data. For standalone components, we can run the same command, which will set up the **`StoreModule`** in the **`main.ts`** file.

```
                         Normal Component
------------------------- app.module.ts -------------------------
import { storeModule } from '@ngrx/store';
@NgModule({
    imports: [storeModule.forRoot({},{})]
})
```

```
                        Standalone Component
---------------------------- main.ts ----------------------------
import { provideStore } from '@ngrx/store';
bootstrapApplication(AppComponent, {
    providers: [ provideStore() ]
});
```

4. Create a **store** directory into the app directory where all the store-related files we will create.

Implementation

1. Add a First Reducer & Store Setup →

```
----------------------- counter.reducer.ts ---------------------
import { createReducer } from '@ngrx/store';

//We Can set any type of value that we need here
const initialState = 0;
export const counterReducer = createReducer(initialState);
                        Normal Component
------------------------ app.module.ts ------------------------
@NgModule({
    imports: [storeModule.forRoot({
       counter: counterReducer,
       // auth: authReducer
    })]
})
                        Standalone Component
---------------------------- main.ts ----------------------------
bootstrapApplication(AppComponent, {
    providers: [
      provideStore({
        counter: counterReducer,
        // auth: authReducer
      })
    ]
});
```

[Note: If we have multiple **reducers**, we can add them here by using key-value structure.**]**

2. An alternative way to create a reducer(**It is the older way**) →

```
--------------------- counter.reducer.ts ---------------------
//We Can set any type of value that we need here
const initialState = 0;

export function counterReducer(state = initialState) {
    return state;
}
```

[Note: It also worked on the standalone component.**]**

3. Read data from the store →

```
----------------- counter-output.component.ts -----------------
import { Store } from '@ngrx/store';

export class CounterOutputComponent {
    count$: Observable<number>;
    constructor(private store: Store<any>) {
        this.count$ = store.select('counter');

        this.value = this.count$.subscribe();
       // Must unsubscribe this inside the ngOnDestroy method. Additionally,
we could use the async pipe to get value instead of subscribing to the
observable variable.
    }
}

---------------- counter-output.component.html ----------------
<p> Output Value: {{ count$ | async }} </p>
```

[Note: It's common practice, though not required, to use a dollar sign (**$**) at the end of property names (e.g., **count$**) that store observables. This is a convention followed in many high-quality projects to indicate that the variable holds an **observable**.

We retrieve an observable from the store in **count$**, which automatically updates whenever the data in the store changes.

In the **select()** method, we pass a key (e.g., **'counter'**) corresponding to a specific reducer that is added using the **StoreModule**.

4. Actions and State-Changing Reducers →

```
---------------------- counter.actions.ts ---------------------
import { createAction } from '@ngrx/store';
export const increment = createAction(
    '[counter] Increment'
);
```

[Note: We can use anything inside the createAction method, but this is the correct convention.**]**

```
---------------------- counter.reducer.ts ---------------------
import { createReducer, on } from '@ngrx/store';
import { increment } from './counter.actions';

const initialState = 0;
export const counterReducer = createReducer(
    initialState,
    on(increment, (state) => state + 1)
);
```

5. Dispatch Actions →

```
---------------- counter-controls.component.ts ----------------
import { Store } from '@ngrx/store';
import { increment } from './counter.actions';

export class CounterControlsComponent {
    count$: Observable<number>;
    constructor(private store: Store<any>) {
        this.count$ = store.select('counter');
    }

    increment() { this.store.dispatch(increment()) }
}
```

6. Attach Data to Actions →

```
---------------------- counter.actions.ts ---------------------
import { createAction, props } from '@ngrx/store';
export const increment = createAction(
    '[counter] Increment',
    // props<{ email: string; password: string;}>()
    props<{ x: number }>()
);
```

```
---------------------- counter.reducer.ts ---------------------
import { createReducer, on } from '@ngrx/store';
import { increment } from './counter.actions';

const initialState = 0;
export const counterReducer = createReducer(
    initialState,
    on(increment, (state, action) => state + action.x)
    // if we have multiple actions then we can write another or more on
block here For example-
    on(decrement, (state, action) => state - action.x)
);
---------------- counter-controls.component.ts ----------------
import { Store } from '@ngrx/store';
import { increment } from './counter.actions';

increment() { this.store.dispatch(increment({x: 2})) }
```

7. Handle Actions without createReducer →

```
---------------------- counter.reducer.ts ---------------------
const initialState = 0;
export function counterReducer(state = initialState, action: any) {
    if (action.type === '[counter] Increment') {
        return state + action.x;
    }
    return state;
}
```

8. An Alternative way to define Actions →

```
---------------------- counter.actions.ts ---------------------
import { Action } from '@ngrx/store';
export const INCREMENT = '[counter] Increment';
export class IncrementAction implements Action {
    readonly type = INCREMENT;
    constructor(public x: number) { }
}

export type CounterActions = IncrementAction;
---------------- counter-controls.component.ts ----------------
import { IncrementAction } from './counter.actions';
increment() { this.store.dispatch(new IncrementAction(2)); }
```

S M HEMEL
Senior Software Engineer
Email: smhemel.eu@gmail.com

```
--------------------- counter.reducer.ts ---------------------
import { Action } from '@ngrx/store';
import { CounterActions, IncrementAction, INCREMENT } from './counter.actions';

const initialState = 0;
export function counterReducer(state = initialState, action: CounterActions | Action) {
    if (action.type == INCREMENT)
        return state + (action as IncrementAction).x;
    return state;
}
```

Selectors

1. We've seen how the **`select()`** method retrieves values from the store using a selector (e.g., **`select('counter')`**). However, a more recommended approach is to create custom selectors. Selectors can be triggered for specific actions and provide a more structured way to access the store's state.

```
-------------------- counter.selectors.ts ---------------------
import { createSelector } from '@ngrx/store';
export const selectCount = (state: {x: number}) => state.x;
// We can write multiple selectors here and we can combine selectors here
with the help of createSelector of the NgRx library.
export const selectDoubleCount = createSelector(
    selectCount,
    (state: number) => state * 2 // Here, we will get value from selectCount
);
----------------- counter-output.component.ts -----------------
import { selectCount, selectDoubleCount } from './counter.selectors';

export class CounterOutputComponent {
    count$: Observable<number>;
    doubleCount$: Observable<number>;
    constructor(private store: Store<any>) {
        this.count$ = store.select(selectCount);
        this.doubleCount$ = store.select(selectDoubleCount);
    }
}
---------------- counter-output.component.html ----------------
<p> Output Value: {{ count$ | async }} </p>
<p> Output Value: {{ doubleCount$ | async }} </p>
```

Effects

1. An effect is triggered by specific actions and is typically used for handling side effects, rather than directly updating the UI. To use effects in our project, first need to install the NgRx effects package by running: **`ng add @ngrx/effects`**
   After installation, the **EffectsModule** will automatically be added to the imports array in the app.module.ts file.
   Note: For the **Standalone** components, the **`EffectsModule`** will be added to the **providers'** array in the main.ts file.
2. Define a Effect →

```
--------------------- counter.effects.ts ----------------------
import { tap } from 'rxjs';
import { Actions, createEffect, ofType } from '@ngrx/effects';
import { decrement, increment } from './counter.actions;

@Injectable()
export class CounterEffects {
  constructor(private actions: Actions) { }
  saveCount = createEffect(() =>
    this.actions.pipe(
      ofType(increment, decrement),
      tap((action) => { localStorage.setItem('count', action.value.toString()); })
    ),
    { dispatch: false }
  );
}
                 - - - - - - - OLD WAY - - - - - - -
import { tap } from 'rxjs';
import { Actions, Effect, ofType } from '@ngrx/effects';
export class CounterEffects {
  constructor(private actions: Actions) { }

  @Effect({ dispatch: false })
  saveCount = this.actions.pipe(
    ofType(increment, decrement),
    tap((action) => { localStorage.setItem('count', action.value.toString()); })
   );
}
```

[Note: It is no longer available in NgRx/effects recent versions.**]**

3. Register an Effects →

```
                        Normal Component
------------------------ app.module.ts ------------------------
@NgModule({
    imports: [
       storeModule.forRoot({counter: counterReducer, // auth: authReducer}),
       EffectsModule.forRoot([CounterEffects])
    ]
})
                     Standalone Component
--------------------------- main.ts ---------------------------
bootstrapApplication(AppComponent, {
    providers: [
       provideStore({ counter: counterReducer, // auth: authReducer}),
       provideEffects([CounterEffects])
    ]
});
```

4. Use Store Data In Effects →

```
---------------------- counter.effects.ts ----------------------
import { Actions, createEffect, ofType } from '@ngrx/effects';
import { tap, withLatestFrom } from 'rxjs';
import { store } from '@ngrx/store';
import { decrement, increment } from './counter.actions';
import { selectCount } from './counter.selectors';

@Injectable()
export class CounterEffects {
    constructor(private actions: Actions, private store: Store) { }
    saveCount = createEffect(() =>
        this.actions.pipe(
            ofType(increment, decrement),
            withLatestFrom(this.store.select(selectCount)),
            tap(([action, counter]) => {
                console.log(action);
                localStorage.setItem('count', counter.toString());
            })
        ),
        { dispatch: false }
    );
}
```

S M HEMEL
Senior Software Engineer
Email: smhemel.eu@gmail.com

5. Add Another Effect →

```
---------------------- counter.actions.ts ---------------------
import { createAction, props } from '@ngrx/store';

export const init = createAction('[counter] Init');
export const set = createAction( '[counter] Set', props<{value: number}>());
export const increment = createAction(
    '[counter] Increment',
    props<{ x: number }>()
);
---------------------- counter.reducer.ts ---------------------
import { createReducer, on } from '@ngrx/store';
import { increment, set } from './counter.actions';

const initialState = 0;
export const counterReducer = createReducer(
    initialState,
    on(increment, (state, action) => state + action.x)
    on(set, (state, action) => action.value)
);
---------------------- counter.effects.ts ----------------------
import { Actions, createEffect, ofType } from '@ngrx/effects';
import { of, tap, withLatestFrom, switchMap } from 'rxjs';
import { store } from '@ngrx/store';
import { decrement, increment, init, set } from './counter.actions';
import { selectCount } from './counter.selectors';

@Injectable()
export class CounterEffects {
  constructor(private actions: Actions, private store: Store) { }

  loadCount = createEffect(() => this.actions.pipe(
      ofType(init),
      switchMap(() => {
        const storedCount = localStorage.getItem('count');
        if (storedCount) return of(set({ value: +storedCount }));
        return of(set({ value: 0 }));
        // By the help of `of`, we return observable from here
      })
    )
  );
// Here we should not add the dispatch: false configuration object to create
an effect. That was only required for where we did indeed not yield a new
action object.

  saveCount = createEffect(() => this.actions.pipe(
       ofType(increment, decrement),
       withLatestFrom(this.store.select(selectCount)),
       tap(([action, counter]) => {
          localStorage.setItem('count', counter.toString());
       })
    ),
    { dispatch: false }
  );
}
---------------------- app.component.ts -----------------------
ngOnInit() { this.store.dispatch(init()); }
```

[Note: After doing these, whenever we reload/restart our app, we will start with the counter that was already stored in the local storage. If we delete that value from local storage and then restart, the app should start with zero.**]**

Angular 17

1. Server-Side Rendering project → **`ng new new_project --ssr`**
2. New syntax →

```
<section>
    @if (user.loggedIn) { // Our code }
    @else if (user.role === 'admin') { // Our code }
    @else { // Our code }
</section>
-----------------------------------------------------------------
<section>
    @for (user of userList; track user) {
      <app-card [data]="user" />
    } @empty {
      <p>No item available</p>
    }
</section>
```

[Note: The track is now required. It will help our loop be faster and more efficient. We can also specify a fallback when there are no items in the list by using @empty.**]**

S M HEMEL
Senior Software Engineer
Email: smhemel.eu@gmail.com

```
------------------------------------------------------------
<section>
    @switch (membershipStatus) {
      @case ('gold') { <p>Your discount is 20%</p> }
      @case ('silver') { <p>Your discount is 10%</p> }
      @default { <p>Keep earning rewards</p> }
    }
</section>
```

3. **Angular Deferred/lazy loading** → Angular has supported native deferred/lazy loading for individual routes, an effective way to reduce the initial bundle size. This technique allows us to separate chunks of content to be deferred or loaded lazily, meaning components can be loaded later and not be part of the initial bundle. Each deferred block requires a specific condition or trigger to determine exactly when the deferred loading should occur.

   `@defer built-in triggers` →

   `idle` → loads as soon as the browser reports it is in idle state.

   `interaction` → loads when an element is clicked, focused, or similar behavior.

   `viewport` → loads when the content enters the client's viewport window.

   `hover` → loads when the mouse is hovering over a designated area.

   `timer` → loads after a specific timeout.

```
-------------------------- OLD WAY --------------------------
export const routes: Route[] = [
  {
    path: 'user',
    loadComponent: () => import('./user.component').then(m => m.UserComponent)
  }
];
------------------------ New defer Syntax ----------------------
                        For Developer Preview
<section #trigger>
    <navbar-component />
    @defer (on viewport(trigger)) { // on idl -OR-
      <large-component />          // on interaction(showContent)
    }                              // on hover(showUser)
    <small-component />            // on immediate
    <footer-component />           // on timer
</section>
```

```
// Use Condition
<button (click)="load = true"> Load Component </button>
@defer (when load == true) {
  <recommended-movies />
}

// Use Trigger and Condition
<button #trigger (click)="load = true"> Load Component </button>
@defer (on viewport(trigger); when load === true) {
  <recommended-movies />
}

// Prefetch
@defer (prefetch on immediate; prefetch when val === true) {
    // @defer (on interaction(showContent); prefetch on idle)
       <recommended-movies />
}

// If we want to show a component/loader before defer load
@defer (on interaction(trigger)) { <recommended-movies /> }
@placeholder (minimum 500ms) { <img src="placeholder-image.png" /> }
@loading (after 500ms; minimum 1s) { <spinner /> }
@error { <p>Oh, something went wrong.</p> }
```

4. In Angular, the **Standalone** feature is now the default for components, directives, and pipes. To create a project, component, directive, or pipe without the standalone feature, use the following commands:
   `ng new new_project --standalone false`
   `ng generate component my_component --standalone false`
5. @Input →

```
import { Component, signal, booleanAttribute  } from '@angular/core';
export class TextInput {
   @Input({ transform: booleanAttribute }) disabled: boolean = false;
}

<!-- Before -->
<text-input [disabled]="true"></text-input>
<!-- After -->
<text-input disabled></text-input>
```

[Note: The **transform** converts its value to boolean and no longer binds the **disabled** to a boolean literal.**]**

Lazy Loading

`What is @defer?` → It is an Angular template syntax that enables loading parts of a template or components only when they are needed. It provides two levels of control:

1. **Prefetching:** Controls when code or data should be fetched from the backend and loaded into memory.
2. **Rendering**: Allows code or components to be rendered separately on the page only when required.

```
@defer() { <large-component /> }
@defer() { <h1>Heading</h1> }
```

Signal Input (Release on 17.1)

1. Required Signal Input →

```
---------------------- main.component.html ----------------------
<app-demo [id]="25" />
----------------------- demo.component.ts -----------------------
import { Component, input, computed } from '@angular/core';
export class DemoComponent {
   id = input.required<number>();
   nextId = computed(() => this.id() + 1);
}
---------------------- demo.component.html ----------------------
<p>Pokemon id: {{ id() }}, Next Pokemon id: {{ nextId() }}</p>
```

[Note: nextId is a computed signal that increments id signal input by 1.**]**

2. Signal Input with initial value →

```
---------------------- main.component.html ----------------------
<app-demo [id]="25" backgroundColor="yellow" />

----------------------- demo.component.ts -----------------------
import { Component, input } from '@angular/core';
export class DemoComponent {
   bgColor = input('cyan', { alias: 'backgroundColor' });
}
---------------------- demo.component.html ----------------------
<p [style.background]="bgColor()"> Background color: {{ bgColor() }} </p>
```

[Note: we declared a signal input with an initial value 'cyan', aliased as backgroundColor. If the component does not receive a backgroundColor input, the value defaults to 'cyan'. Otherwise, backgroundColor is overwritten by the provided input value.**]**

3. Transformed Signal Input →

```
---------------------- main.component.html ----------------------
<app-demo transformedText="green" />

----------------------- demo.component.ts -----------------------
import { Component, input } from '@angular/core';
export class DemoComponent {
   text = input<string, string>('', {
      alias: 'transformedText',
      transform: (v) => `transformed ${v}!`,
   });
}
---------------------- demo.component.html ---------------------
<p>Transformed: {{ text() }}</p>
```

[Note: we declare a signal input with an empty string, aliased as transformedText, along with a **transform** function. The transform function takes a string, prepends 'transformed', and appends an exclamation mark to it. For example, the demo component displays "transformed green!".**]**

Release on 17.2

1. Two-way binding with Signal Inputs**(Model Inputs)** → Read Me, Read Me.

Important Things

1. To disable strict mode in a project, set **strict: false** in our **tsconfig.json** file.
2. To create a **data model,** first declare the data fields in a class and use the constructor to initialize values for these fields.

```
export class Ingredient {
    public name: string;
    public amount: number;
    constructor(name: string, amount: number) {
        this.name = name;
        this.amount = amount;
    }
}
```

Shortcut way →

```
export class Ingredient {
    constructor(public name: string, public amount: number) {}
}
```

3. When subscribing to any observable in a component, the subscription persists even after leaving the component. To manage this, we import **`Subscription`** from the RxJS library and create a **`Subscription`**-type variable. Then, in the **`onDestroy`** lifecycle method, we unsubscribe from all subscriptions to ensure they're properly cleaned up when the component is destroyed.
4. **<ng-template></ng-template>** is an Angular directive that doesn't render anything directly to the DOM but remains accessible in Angular's templating. It's commonly used to reuse and organize code within an HTML file.

```
<ng-template #show let-msg="showMesssage">
    <p>{{ msg }}</p>
</ng-template>
<ng-container
    *ngTemplateOutlet="show;
    context : {showMessage: 'This is the context message.'}"
>
</ng-container>

// Using $implicit
<ng-template #show let-person>
    <p>My name is {{ person.name }}, I am {{ person.age }} years old.</p>
</ng-template>
<ng-container
   *ngTemplateOutlet="show; context : {$implicit: showDetails}">
</ng-container>

const showDetails = {
    name: 'Mr. X',
    age: 45
};
```

5. **Understanding Config files →**
   `.editorconfig` → Override user default IDE settings to ensure consistent coding style in a project for all users, regardless of the editor.
   `.gitignore` → Specifies files for Git to ignore (e.g., frequently regenerated files like `node_modules`).
   `angular.json` → Manages the project's configuration. Contains project type (**`"application"`** or **`"library"`**) and prefixes for components. If we look into it, we can easily understand every property.
   `browserslist` → Used by Angular CLI during production builds and tells the Angular CLI which browsers we want to support with your project.
   `karma.conf` → Configures the Karma testing framework for unit tests.
   `package-lock.json` → This file is generated based on our package.json file, which manages the packages that we use for our project and the versions of those packages. It stores the exact versions of your third-party dependencies because in the package.json we specify some minimum requirements and some flexibility. After that, once we run the npm install, we lock in certain exact versions which are saved here last but not least.
   `tsconfig.json` → Configures the TypeScript compiler. Includes options like **`strict: true`** for stricter typing, which can lead to cleaner code but may vary based on project needs.
   `tslint.json` → This simply is used for linting to check our code quality and tells us whether we're following best practices and we wrote our code in a good way. If you get errors here, these are not real errors. These are just improvement recommendations on how we could improve your code.
6. **Managing Multiple projects in one Folder →**
   Create another project by run - **`ng generate application another_app`**
   Run a specific project - **`ng serve --project=another_app`**

Learn Modern Angular(16 - Higher)

Project Setup

1. Install Angular with npm → **`npm install -g @angular/cli@latest`**
2. Creating app → **`ng new my-app`** and providing all the questions answered.

Basics

Component

1. Structure of a Component(**component.ts**) →

```
----------------------- demo.component.ts -----------------------
import { Component } from '@angular/core';

@Component({
  selector: 'app-demo',
  standalone: true,
  imports: [HeaderComponent], // Component list that use under this demo component
  // template: '<p>Hello</p>' // We can write HTML code here
  templateUrl: './demo.component.html',
  styleUrls: ['./demo.component.scss'],
  // styles: [`h3{color: white}`] // We can write CSS code here
})
export class DemoComponent {}
```

[Note: We cannot use `` `template` `` and `` `templateUrl` `` simultaneously, and the same rule applies to `` `styles` `` and `` `styleUrls` ``.**]**

2. To use files (such as images or SVGs) from the **assets** directory in a component file, ensure **`"src/assets"`** is included in the **assets** array of **architect > build > options** object of the angular.json file. Without this, these images won't load.

Content Projection & ng-content

1. General way of content projection →

S M HEMEL
Senior Software Engineer
Email: smhemel.eu@gmail.com

```
------------------ dashboard-item.component.html -----------------
<div class="dashboard-item">
    <article>
        <header>
            <img [src]="image().src" [alt]="image().alt" />
            <h2>{{ title() }}</h2>
        </header>
        <ng-content />
    </article>
</div>
--------------------- parent.component.html ---------------------
<app-dashboard-item
   [image]="{ src: 'status.png', alt: 'A signal symbol'}"
   title="Server Status"
>
  <p>Servers are online</p>
</app-dashboard-item>
```

2. Component Projection with Multiple slots →

```
---------------------- button.component.html --------------------
<span> <ng-content /> </span>
<span> <ng-content select=".icon" /> </span>

--------------------- parent.component.html ---------------------
<button appButton>
    Logout
    <span class="icon">→</span>
</button>
```

3. Advance Projection →

```
---------------------- button.component.html --------------------
<span> <ng-content /> </span>
<span>
    <ng-content select="icon" />
</span>

--------------------- parent.component.html ---------------------
<button appButton>
    Logout
    <span ngProjectAs="icon">→</span>
</button>
```

S M HEMEL
Senior Software Engineer
Email: smhemel.eu@gmail.com

4. Content Projection Fallbacks →

```
---------------------- button.component.html --------------------
<span> <ng-content /> </span>
<span>
    <!-- Important: FallBack Content -->
    <!-- If we don't send any content for projection then the fallback
content will show. Here, --- is the fallback content -->
    <ng-content select="icon"> *** </ ng-content
</span>
--------------------- parent.component.html ---------------------
<button appButton> Logout </button>
```

5. Multi-element Content Projection →

```
---------------------- control.component.html --------------------
<p>
    <label>{{ label() }}</label>
    <ng-content select="input, textarea" />
</p>
--------------------- parent.component.html ---------------------
<form>
    <app-control label="Title">
        <input name="title" id="title" />
    </app-control>

    <app-control label="Request">
        <textarea name="request" id="request" rows="3"></textarea>
    </app-control>
    <p>
        <button appButton>
            Logout
            <span ngProjectAs="icon">→</span>
        </button>
    </p>
</form>
```

Host Element

1. Component Host Element →

```
--------------------- button.component.html ---------------------
<span> <ng-content /> </span>
<span> <ng-content select="icon" /> </span>
```

```
--------------------- button.component.css ---------------------
:host {
    text-align: center;
    cursor: pointer;
}
:host:hover { background-color: #551b98; }
--------------------- parent.component.html ---------------------
<li>
    <button appButton>
        Logout
        <span ngProjectAs="icon">→</span>
    </button>
</li>
```

2. Interacting with Host Elements from Component →

   The class name of the **'host'** object will be added with the host tag.

```
-------------------- control.component.html ---------------------
<label>{{ label() }}</label>
<ng-content select="input, textarea" />
--------------------- control.component.ts ---------------------
@Component({
    selector: 'app-control',
    standalone: true,
    templateUrl: './control.component.html',
    styleUrl: './control.component.css',
    encapsulation: ViewEncapsulation.None,
    // It will make this css component as a global component.
    // We can avoid this if we add a host in the css file.
    host: { class: 'control' },
})

export class ControlComponent {
    label = input.required<string>();
}
---------------------      Alternatively     ---------------------
export class ControlComponent {
    @HostBinding('class') className = 'control';
    // We can use it by commenting out the host property inside the
Component Decorator. But, it’s not the recommended way.
    label = input.required<string>();
}
```

```
--------------------- parent.component.html ---------------------
<form>
    <app-control label="Title">
        <input name="title" id="title" />
    </app-control>

    <app-control label="Request">
        <textarea name="request" id="request" rows="3"></textarea>
    </app-control>
</form>
```

3. Interacting with Host Elements vis **@HostBinding** and **@HostListener** → The class name of the **'host'** object will be added with the host tag.

```
---------------------- control.component.ts ---------------------
@Component({
    selector: 'app-control',
    standalone: true,
    templateUrl: './control.component.html',
    styleUrl: './control.component.css',
    host: { class: 'control', '(click)': 'onClick()' },
})
export class ControlComponent {
    label = input.required<string>();
    onCLick() { }
}
---------------------      Alternatively     ---------------------
export class ControlComponent {
    @HostBinding('class') className = 'control'; // We can use it by
commenting out the host property inside the Component Decorator.
    @HostListener('click') onCLick() { }
}
```

[Note: It is not the recommended way for **HostBinding.]**

4. Accessing Host Elements Programmatically →

```
---------------------- control.component.ts ---------------------
export class ControlComponent {
  label = input.required<string>();
  private el = inject(ElementRef);

  onCLick() {}
}
```

Binding

1. Class Binding →

```
<div [class.status] = "currentStatus === 'offline'"></div>
----------------------      Alternatively      ----------------------
<div [class]="{
    status: true,
    'status-online': currentStatus === 'online',
    'status-offline': currentStatus === 'offline',
    'status-unknown': currentStatus === 'unknown'
}"> </div>
```

2. Style Binding →

```
<div [style.fontSize] = "'64px'"></div>
<div [style.height]="(dataPoint.value / maxTraffic) * 100 + '%'"> </div>
----------------------      Alternatively      ----------------------
<div [style] ="{ fontSize: '64px' }"></div>
```

Lifecycle Methods

https://angular.dev/guide/components/lifecycle

An modern alternative of **OnDestroy()** →

```
export class ServerStatusComponent implements OnInit {
  currentStatus: 'online' | 'offline' | 'unknown' = 'offline';
  private destroyRef = inject(DestroyRef);

  ngOnInit() {
    const interval = setInterval(() => {
      if (Math.random() < 0.5) {
        this.currentStatus = 'online';
      }
    }, 5000);

    this.destroyRef.onDestroy(() => {
      clearInterval(interval);
    });
  }
}
```

AfterRender and AfterNextRender LifeCycle Function

Introduced in Angular 16, **AfterRender** and **AfterNextRender** lifecycle functions can be registered within the constructor.

- **AfterRender** executes repeatedly whenever there is any change in the entire Angular application, making it suitable for tasks that need continual updates in response to any change.
- **AfterNextRender**, in contrast, triggers only once for the next change and does not execute repeatedly.

The key difference is that **AfterRender** continually listens to all subsequent changes, whereas **AfterNextRender** is limited to triggering on the next single change in the application.

```
constructor() {
    afterRender(() => { console.log('afterRender'); });

    afterNextRender(() => { console.log('afterNextRender'); });
}
```

Component Instance by using Template Variable

When a template variable (e.g., **#btn**) is used with a component selector like **<appButton>**, it enables direct access to all the methods and properties of that component instance. Here, **`#btn`** acts as a reference to the **`appButton`** component instance and allows us to call methods or access properties of **`appButton`** directly from the template or within the component's TypeScript code, making it straightforward to interact with specific component instances in a flexible way.

```
---------------------- demo.component.html ----------------------
<form (ngSubmit)="onSubmit()" #form>
    <app-control label="Title">
        <input name="title" id="title" #titleInput />
    </app-control>
    <p>
        <button appButton #btn>
            Logout
            <span ngProjectAs="icon">→</span>
        </button>
    </p>
</form>
```

```
------------------------ demo.component.ts ------------------------
@ViewChild('form') form?: ElementRef<HTMLFormElement>;
```

ViewChild Signal Function

```
------------------------ demo.component.ts ------------------------
export class DemoComponent {
  private buttonRef = viewChild(ButtonComponent);
  private form = viewChild.required<ElementRef<HTMLFormElement>>('form');

  onSubmit() {
    this.form().nativeElement.reset();
    // If we don't use required, then we should use optional after form().
  }
}
```

ContentChild Signal Function

```
---------------------- shared.component.ts ----------------------
export class ControlComponent {
  private control = contentChild<ElementRef<HTMLInputElement |
HTMLTextAreaElement>>('input');

  onCLick() { console.log(this.control()); }
}
---------------------- shared.component.html ----------------------
<label>{{ label }}</label>
<ng-content select="input, textarea" />
---------------------- parent.component.ts ----------------------
<form (ngSubmit)="onSubmit()" #form>
  <app-control label="Title">
    <input name="title" id="title" #input />
  </app-control>

  <app-control label="Request">
    <textarea name="request" id="request" rows="3" #textInput #input></textarea>
  </app-control>
</form>
```

Signal

```
----------------------- demo.component.ts -----------------------
export class DemoComponent {
  selectedUser = signal(Users[0]);
  imagePath = computed(() => 'assets/users/' + this.selectedUser().avatar);

  onSelectedUser() { this.selectedUser.set(Users[randomIndex]); }

  // Update Signal Value
  detailsVisible = signal(false);
  onToggleDetails() {
    this.detailsVisible.update((wasVisible) => !wasVisible);
  }
}
```

Signal Input & Output

```
// @Input() avatar!: string;
// @Input({ required: true }) name!: string;
// @Output() select = new EventEmitter<string>();

avatar = input<string>();
name = input.required<string>();
select = output<string>();

imagePath = computed(() => { return 'assets/users/' + this.avatar() });
onSelectedUser() { this.select.emit('Clicked'); }
```

Signal Effects

Use Case: we may want to trigger code within the TypeScript class whenever a Signal changes, beyond just updating the template automatically. This is helpful when we need specific logic to run in response to Signal updates, not directly tied to the UI.

Solution: Modern angular provides us a special function called **effect**, which we can execute in our constructor and **effect** takes a function as an argument. If we use a Signal in that function that's passed to **effect**, Angular will set up a subscription. The subscription will automatically clean up if that component should get removed from the DOM.

```
export class ServerStatusComponent implements OnInit {
  currentStatus = signal<'online' | 'offline' | 'unknown'>('offline');
  private destroyRef = inject(DestroyRef);

  constructor() {
    effect(() => { console.log(this.currentStatus()); });
    effect(() => { console.log('1'); });
  }

  ngOnInit() {
    const interval = setInterval(() => {
      if (Math.random() < 0.5) {
        this.currentStatus.set('online');
      } else { this.currentStatus.set('unknown'); }
    }, 3000);
    this.destroyRef.onDestroy(() => { clearInterval(interval); });
  }
}
```

Signal Effects CleanUp Functions

When working with Signal effects in Angular, there are times when cleanup tasks are necessary before the effect re-runs. For this, Angular's **effect** function provides an **onCleanup** hook, which allows us to specify cleanup actions before the effect is re-triggered. This is particularly useful for tasks like unsubscribing from services, resetting variables, or freeing up resources.

Here's how it works:

- Inside the **`effect`** function, use **`onCleanup`** to define a function with any actions that should occur before the effect re-runs.
- This cleanup function will automatically execute each time the effect function is about to run again.

```
constructor() {
  effect((onCleanup) => {
    const tasks = getTasks();
    const timer = setTimeout(() => { console.log(tasks().length); }, 1000);

    onCleanup(() => { clearTimeout(timer); });
  });
}
```

@for & @if

```
<main>
  <ul>
    @for (user of users; track user.id) {
      <li>
        <app-user [user]="user" [selected]="user.id === selectedUserId" />
      </li>
    }
  </ul>

  @if (selectedUser) {
    <app-tasks [userId]="selectedUser.id" />
  } @else {
    <p id="fallback">Select a user to see their tasks</p>
  }
</main>
```

An Easier Way of Setting Up Custom Two-Way Binding

```
--------------------- child.component.html ---------------------
<div [style]="{ width: size().width + 'px', height: size().height + 'px' }">
</div>
---------------------- child.component.ts ----------------------
export class RectComponent {
    size = model.required<{ width: string; height: string }>();
    onReset() { this.size.set({ width: '200', height: '100' }); }
}
--------------------- parent.component.html ---------------------
<app-react [(size)]="rectSize" />
```

Directive

In modern Angular, there are no built-in structural directive left.

1. Custom Structural directive →

```
-------------------- safe-link.directive.ts --------------------
@Directive({
    selector: 'a[appSafeLink]',
    standalone: true,
    host: { '(click)': 'onConfirm($event)' }
})
```

S M HEMEL
Senior Software Engineer
Email: smhemel.eu@gmail.com

```
export class SafeLinkDirective {
  queryParam = input('myapp'); // 'myapp' is the default value
  onConfirm(event: MouseEvent) {
    const address = (event.target as HTMLAnchorElement).href;
    (event.target as HTMLAnchorElement).href = address+'?from='+this.queryParam;
  }
}

---------------- HTML file where we use the directive ----------------
<ul>
  <li>
    <a href="https://angular.dev" appSafeLink queryParam="myapp-docs-link"></a>
  </li>
  <li>
    <a href="https://angular.dev" appSafeLink queryParam="myapp-course-link"></a>
  </li>
</ul>

----------------- If we use alias with input field ------------------
queryParam = input('myapp', { alias: 'appSafeLink'});

<ul>
  <li>
    <a href="https://angular.dev" appSafeLink="myapp-docs-link"></a>
  </li>
  <li>
    <a href="https://angular.dev" appSafeLink="myapp-course-link"></a>
  </li>
</ul>
```

2. Dependency Injection →

```
-------------------- safe-link.directive.ts ---------------------
export class SafeLinkDirective {
  queryParam = input('myapp', { alias: 'appSafeLink' });
  private hostElementRef = inject<ElementRef<HTMLAnchorElement>>(ElementRef);

  onConfirm(event: MouseEvent) {
    const address = (event.target as HTMLAnchorElement).href;
    this.hostElementRef.nativeElement.href = address+'?from='+this.queryParam;
  }
}
```

3. Structural Directive →

```
-------------------- safe-link.directive.ts ---------------------
@Directive({
    selector: '[appAuth]',
    standalone: true
})
export class AuthDirective {
  userType = input.required<'user' | 'guest' | 'default'>({ alias: 'appAuth' });
  private authService = inject(AuthService);
  private templateRef = inject(TemplateRef);
  // When building a structural directive, it is super important
  private viewContainerRef = inject(ViewContainerRef);
  // This view container ref works as a reference to the place in the DOM

  constructor() {
    effect(() => {
       if (this.authService.activePermission() === this.userType()) {
          this.viewContainerRef.createEmbeddedView(this.templateRef);
       } else {
          this.viewContainerRef.clear();
       }
    });
  }
}
---------------- HTML file where we use the directive ----------------
<ng-template appAuth="admin">
    <p>Hello World!</p>
</ng-template>
```

4. **Host Directive** → Imagine we have a directive that contains a click event in the host object which is linked to a method that logs the host element ref. Typically, we use the directive with HTML tags where needed. However, there is another way to use it that works effectively. Modern angular provides a method called **hostDirectives**, which we can add to the decorator object.

```
------------------------ log.directive.ts ------------------------
@Directive({
    selector: '[appLog]',
    standalone: true,
    host: { '(click)': 'onLog()' }
})
```

```
export class LogDirective {
    private elementRef = inject(ElementRef);
    onLog() { console.log(this.elementRef.nativeElement); }
}
-------------------- app-ticket.component.ts --------------------
@Component({
    selector: 'app-ticket',
    standalone: true,
    templateUrl: './ticket.component.html',
    styleUrl: './ticket.component.css',
    hostDirectives: [LogDirective]
})
```

Services & Dependency Injection

1. Inject service into a component →

```
------------------------ task.service.ts ------------------------
export class Tasks Service {
  private tasks = signal<Task[]>([]); Accessible only in this class
  allTasks = this.tasks.asReadOnly(); Return a read-only version of a signal

  addTask(taskData: any) {
    const newTask: Task = {
       ...taskData,
       id: Math.random().toString(),
       status: 'OPEN'
    };
    this.tasks.update((oldTasks) => [...oldTasks, newTask]);
  }

  updateTaskStatus(taskId: string, status: string) {
    this.tasks.update((oldTasks) =>
       oldTasks.map((task) => task.id === taskId ? {...task, status: status} : task)
    );
  }
}
--------------------- account.component.ts ---------------------
export class AccountComponent {
    private tasksService = inject(TasksService);
}
```

S M HEMEL
Senior Software Engineer
Email: smhemel.eu@gmail.com

2. Using Custom DI Tokens & Providers →

```
---------------------------- main.ts ----------------------------
export const TasksServiceToken = new
InjectionToken<TasksService>('tasks-service-token');

bootstrapApplication(AppComponent, {
    providers: [{ provide: TasksServiceToken, useClass: TasksService}]
}).catch((err) => console.error(err));
--------------------- account.component.ts ----------------------
export class AccountComponent {
    private tasksService = inject(TasksServiceToken);
}
```

Change Detection

How does the Angular change detection Mechanism work by default?

Let's imagine a component tree(where one component is nested within another), and angular wraps this entire tree within a **'zone'**. This zone is a feature provided by **zone.js**(a library maintained by the angular team). It informs Angular about potential events happening on the pages that may need to be reflected on the screen.
For instance, if a user clicks a button, Angular is notified of this event, triggering a change detection process. Angular then proceeds to visit every component in the application, regardless of where the event occurred. It examines all templates and template bindings, such as property bindings and string interpolations, checking if any bindings produce a new value. If a new value is detected, Angular updates the real DOM to reflect this change. This is how Angular's change detection works by default.

In development mode, Angular runs change detection twice by default. If code within a block whose value changes after every change detection cycle, we get an error called **ExpressionChangedAfterItHasBeenCheckedError**.

1. **Avoiding Zone Pollution →**
   To prevent zone.js from monitoring specific sections of code, we can use the **`runOutsideAngular`** method. By passing a function to this method, we can wrap code that should execute outside of Angular's change detection or **`zone.js`** monitoring. This approach helps avoid unnecessary change detection cycles.

```
---------------------- demo.component.ts ------------------------
export class DemoComponent implements OnInit {
    private zone = inject(NgZone);

    ngOnInit() {
        this.zone.runOutsideAngular(() => {
            setTimeout(() => {
                console.log('Time expired');
            }, 5000);
        });
    }
}
```

2. **OnPush strategy →**

   To avoid expensive function calls, Angular offers a powerful strategy that can significantly improve application performance: the **OnPush** change detection strategy. This alternative change detection approach can be applied on a per-component basis to ensure that change detection runs less frequently for selected components.

   When the **OnPush** strategy is enabled on a component, Angular will only reevaluate that component and its child components if an event occurs within that component or one of its child components, or if an input value changes. However, if the **OnPush** strategy is enabled only on a child component, that child component will not reevaluate when events occur in the parent component.

   This selective reevaluation process helps improve application performance by reducing unnecessary change detection cycles.

```
---------------------- demo.component.ts ------------------------
@Component({
    selector: 'app-demo',
    standalone: true,
    templateUrl: './demo.component.html',
    styleUrl: './demo.component.scss',
    changeDetection: ChangeDetectionStrategy.OnPush
})
```

3. **Problem with OnPush →**

   Consider an application with a service that stores a **`messages`** array and three components: **`add-message`**, **`display-message`**, and a **`parent-component`**

that contains both. When a message is added via the **`add-message`** component, it won't appear in the **`display-message`** component if the OnPush strategy is enabled on **`display-message`**.
To resolve this, we have two options: using RxJS's **`subscribe`** method or the **`AsyncPipe`**.
<u>Note</u>: If using Angular **Signals**, no additional setup is required. **Signals** will handle the reactivity as usual.

```
---------------------- messages.service.ts ----------------------
export class MessagesService {
    message$ = new BehaviorSubject<string[]>([]);
    private messages: string[] = [];

    get allMessages() {
        return [...this.messages];
    }

    addMessage(message: string) {
        this.messages = [...this.messages, message];
        this.message$.next([...this.messages]);
    }
}
                          ***Subscribe Way***
------------------ message-display.component.ts ------------------
export class MessagesListComponent implements OnInit {
    private messagesService = inject(MessagesService);
    private cdRef = inject(ChangeDetectorRef);
    private destroyRef = inject(DestroyRef);

    messages: string[] = [];
    ngOnInit() {
        const subscription = this.messagesService.message$
           .subscribe((messages) => {
              this.messages = messages;
              this.cdRef.markForCheck();
        });

        this.destroyRef.onDestroy(() => subscription.unsubscribe(); });
    }
}
```

```
                         ***Async Way***
----------------- message-display.component.ts -----------------
@Component({
    selector: 'display-message',
    standalone: true,
    imports: [AsyncPipe],
    templateUrl: './display-message.component.html',
    styleUrl: './display-message.component.scss',
    changeDetection: ChangeDetectionStrategy.OnPush
})
export class MessagesListComponent {
    private messagesService = inject(MessagesService);
    message$ = this.messagesService.messages$;
}
---------------- message-display.component.html ----------------
<ul>
    @for (message of messages$ | async; track message) {
        <li>{{ message }}</li>
    }
</ul>
```

4. **Zoneless App** -

   To make an Angular app zoneless, first, remove **`zone.js`** from the **`polyfills`** array in the **`angular.json`** file. Then, add **`provideExperimentalZonelessChangeDetection`** to the **`bootstrapApplication`** function within the providers array.

   Note: This is an experimental feature available in Angular 18. It works best when Signals are used consistently throughout the project.

```
-------------------------- main.ts ---------------------------
import { bootstrapApplication } from '@angular/platform-browser';
import { provideExperimentalZonelessChangeDetection } from '@angular/core';
import { AppComponent } from './app/app.component';

bootstrapApplication(AppComponent, {
    providers: [provideExperimentalZonelessChangeDetection()]
}).catch((err) => console.error(err));
```

RxJS observables

In the previous section, we explored RxJS. Now, we are working with Signals, which offer similar features to RxJS in some respects.

RxJS also includes Subjects, which are a specific type of observable with unique properties. In modern Angular, however, we have a new built-in concept called Signals, which functions differently from observables. Now, let's explore Signals, focusing on the similarities and differences between Observables and Signals. We'll also cover how to convert a Signal to an Observable and vice versa.

Creating and Using an Observable -

```
----------------------- messages-list.component.ts -----------------------
export class MessagesListComponent {
    clickCount = signal(0);
    constructor() {
        effect(() => { console.log(this.clickCount()); });
    }
    onCLick() { this.clickCount.update(prevCount => prevCount + 1); }
}
```

[Note: For every count update, the console.log will print the value.]

Signals vs Observables →

When comparing Signals to Observables (especially Subjects), they may initially seem quite similar. For instance, when using a Subject in a service to share state between different components, we could use a Signal instead—this might even be a better approach in modern Angular.

The key difference is that Observables provide a pipeline of values emitted over time, while Signals act as value containers. Signals allow you to change values, and any observers are notified immediately. A significant advantage of Signals is that you can access the current value directly at any time without needing a subscription.

```
                            - Using RxJS -
ngOnInit() {
    const subscription = interval(1000)
        .pipe(map((val: number) => val * 2))
        .subscribe(() => {
            next: (val: number) => console.log(val); // 0 2 4 6 8
        });
}
```

```
                       – Using Signal –
interval = signal(0);
doubleInterval = computed(() => this.interval() * 2);

constructor() {
  effect(() => { console.log(this.doubleInterval()); });
}
ngOnInit() {
  setInterval(() => {
    this.interval.update(prevVal => prevVal + 1);
  }, 1000)
}
```

Convert Signals To Observables →

```
import { toObservable } from '@angular/core/rxjs-interop';
export class AppComponent implements OnInit {
 clickCount = signal(0);
 clickCount$ = toObservable(this.clickCount);
 private destroyRef = inject(DestroyRef);

 constructor() {
   // effect(() => { console.log(this.clickCount()); });
 }

 ngOnInit() {
  // setInterval(() => { this.clickCount.update(prevVal => prevVal + 1); }, 1000);
  const subscription = this.clickCount$.subscribe({
    next: (val) => console.log(val)
  });
  this.destroyRef.onDestroy(() => { subscription.unsubscribe(); });
 }

 onCLick() { this.clickCount.update(prevCount => prevCount + 1); }
}
```

Convert Observables To Signals →

```
import { toSignal } from '@angular/core/rxjs-interop';
import { interval } from 'rxjs';
export class AppComponent implements OnInit {
 interval$ = interval(1000);
 intervalSignal = toSignal(this.interval$, { initialValue: 0 });
```

```
  constructor() {
    effect(() => { console.log(this.intervalSignal()); });
  }
  ngOnInit() {
   // setInterval(() => { this.clickCount.update(prevVal => prevVal + 1); }, 1000);
  }
}
```

Http

1. **Handling HTTP Errors →**

```
import { catchError } from 'rxjs';
export class AppComponent implements OnInit {
    isFetching = signal<boolean>(false);
    places = single<Place[]>();

    ngOnInit() {
        this.isFetching.set(true);
        this.httpClient.get<{ places: Place[] }>(API_URL)
            .pipe(map(
                (resData: any) => resData.places),
                catchError((error) => throwError(() => new Error(error))
            )).subscribe({
                next: (places) => { this.places.set(places); },
                error: (error) => { console.log(error); },
                complete: () => { this.isFetching.set(false); }
            });
    }
}
```

2. **Introducing HTTP Interceptors →**

```
------------------- interceptor.service.ts --------------------
import { HttpEvent, HttpHandler, HttpInterceptor, HttpRequest } from '@angular/common/http';
import { Observable } from 'rxjs';
@Injectable()
class LoggingInterceptor implements HttpInterceptor {
  intercept(req: HttpRequest<unknown>, handler: HttpHandler): Observable<HttpEvent<any>> {
    console.log('Request URL: ' + req.url);
    return handler.handle(req);
  }
}
```

```
--------------------------- main.ts ---------------------------
bootstrapApplication(AppComponent, {
    providers: [
      provideHttpClient(withInterceptorsFromDi()),
      {
          provide: HTTP_INTERCEPTORS,
          useClass: LoggingInterceptor, multi: true
      }
    ]
  }).catch((err) => console.error(err));
```

3. **HTTP Response Interceptors →**

```
------------------- interceptor.service.ts --------------------
import { HttpEvent, HttpEventType, HttpHandler, HttpInterceptor, HttpRequest
} from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Observable, tap } from 'rxjs';

@Injectable()
class LoggingInterceptor implements HttpInterceptor {
    intercept(req: HttpRequest<unknown>, handler: HttpHandler):
Observable<HttpEvent<any>> {
      console.log('Outgoing  ' + req.url);
      return handler.handle(req).pipe(
        tap({
          next: (event: any) => {
            if (event.type === HttpEventType.Response) {
              console.log('Incoming Response.');
              console.log(event.status, event.body);
            }
          }
        })
      );
    }
}
```

Angular Forms

Template Driven Approach

1. **Accessing form with signal viewChild() →**

```
----------------------- demo.component.html -----------------------
<form  #form="ngForm" (ngSubmit)="onSubmit(form)"> </form>
----------------------- demo.component.ts -----------------------
private form = viewChild.required<NgForm>('form');
  private destroyRef = inject(DestroyRef);

  constructor() {
    afterNextRender(() => {
      const subscription = this.form().valueChanges?.pipe(debounceTime(500))
           .subscribe({
              next: (value) => window.localStorage.setItem(
                'search-value', JSON.stringify({ email: value.email })
              ),
           });

      this.destroyRef.onDestroy((() => subscription?.unsubscribe()));
    });
}
```

[**NOTE:** Here we do some work that is useful for some tasks. For example, we have a search field which is a form. We are seeing that when we use **afterNextRender** it will trigger once when the component is loaded and we subscribe form value when the value changes inside it. While we are typing value in the search field this method will trigger every time. But we use a RxJS method called debounceTime which will prevent triggers if we don't wait 500ms after keyboard press.]

Routing

Setting & Config

```
--------------------------app.routes.ts --------------------------
export const routes: Routes = [
    { path: 'tasks', component: TasksComponent }
];
```

```
------------------------- app.config.ts -------------------------
export const appConfig: ApplicationConfig = {
    providers: [ provideRouter(routes) ]
};
---------------------------- main.ts ----------------------------
bootstrapApplication(AppComponent, appConfig).catch((err) =>
console.error(err));
```

Parameter

1. Extracting Dynamic Route Parameters via Inputs →

```
------------------------- app.config.ts -------------------------
export const appConfig: ApplicationConfig = {
    providers: [
        provideRouter(routes, withComponentInputBinding())
    ]
};
-------------------------app.routes.ts -------------------------
export const routes: Routes = [
    { path: 'tasks', component: TasksComponent },
    { path: 'user/:userId', component: UsersComponent }
];
----------------------- user.component.ts -----------------------
export class UsersComponent {
    userId = input.required<string>(); // Here we get userId from route parameter
}
```

2. Pass Dynamic Data using Resolver →

```
-------------------------app.routes.ts -------------------------
export const routes: Routes = [
  { path: 'tasks', component: TasksComponent },
  { path: 'user/:userId', component: UsersComponent,
    children: [
      { path: 'tasks', component: TasksComponent },
      { path: 'tasks/new' component: NewTaskComponent }
    ],
    resolve: {
     userName: resolveUserName // Can add multiple resolver here for multiple field
    }
  }
];
```

```
----------------------- tasks.component.ts -----------------------
export class TasksComponent {
  userName = input.required<string>(); // Here we will get value from resolver
}
export const resolveUserName: ResolveFn<string> = (activatedRoute:
ActivatedRouteSnapshot, routerState: RouterStateSnapshot) => {
  const usersService = inject(UsersService);
  const userName = usersService.users.find(
    (u: any) => u.id === activatedRoute.paramMap.get('userId'))?.name || '';
  return userName;
};
```

3. Reading Page using Router Programmatic Navigation →

```
------------------------ tasks.component.ts ----------------------
export class TasksComponent {
    tasks = input.required<Task[]>();
    private router = inject(Router);
    private activatedRoute = inject(ActivatedRoute);

    onCompleteTask() {
        this.tasksService.removeTask(this.tasks().id);
        this.router.navigate(['./'], {
            relativeTo: this.activatedRoute, onSameUrlNavigation: 'reload'
        });
    }
}
```

[**Note:** Ensure that we set runGuardsAndResolvers: 'always' for this page route.]

Nested Route

1. Accessing Parent Route Data from Inside Nested Routes →

```
-------------------------- app.config.ts --------------------------
export const appConfig: ApplicationConfig = {
  providers: [
    provideRouter(routes, withComponentInputBinding(), withRouterConfig({
      paramsInheritanceStrategy: 'always', // This will ensure dynamics routes
parameter values are injected into child routes
    }))
  ]
};
```

S M HEMEL
Senior Software Engineer
Email: smhemel.eu@gmail.com

```
-------------------------app.routes.ts -------------------------
export const routes: Routes = [
    { path: 'tasks', component: TasksComponent },
    { path: 'user/:userId', component: UsersComponent,
        children: [
            { path: 'tasks', component: TasksComponent },
            { path: 'tasks/new' component: NewTaskComponent }
        ]
    }
];
----------------------- user.component.ts -----------------------
export class UsersComponent {
    userId = input.required<string>();
    // Here we get userId from route parameter
}
```

Guard

1. **CanMatch**(Function based and Class based)→

```
------------------------- app.routes.ts -------------------------
// We can write this function anywhere in our project
const dummyCanMatchs: CanMatchFn = (route, segments) => {
    const router = inject(Router);
    const shouldGetAccess = Math.random();

    if (shouldGetAccess < 0.5) return true;
    return new RedirectCommand(router.parseUrl('/unauthorized'));
}

const routes: Routes = [
    { path: '', credirectTo: 'tasks', pathMatch: 'full' },
    {
      path: 'tasks',
      component:TasksComponent,
      runGuardsAndResolvers: 'always',
      canMatch: [dummyCanMatchs], // Can add multiple function here
      resolve: { userName: resolveUserName },
      title: resolveTitle
    }
];
```

```
                        CLASS BASED CanMatch GUARD
--------------------- auth-guard.service.ts ---------------------
@Injectable({ providedIn: 'root' })
class CanMatchTeamSection implements CanMatch {
    constructor(private router: Router) { }

    canMatch(route: Route, segments: UrlSegment[]) {
        const shouldGetAccess = Math.random();
        if (shouldGetAccess < 0.5)
            return true;
        return new RedirectCommand(this.router.parseUrl('/unauthorized'));
    }
}
------------------------- app.routes.ts -------------------------
const appRoutes: Routes = [
    { path: 'users/:userId', component: UserTasksComponent,
      runGuardsAndResolvers: 'always',
      canMatch: [CanMatchTeamSection],
      resolve: { userName: resolveUserName },
      title: resolveTitle
    }
];
```

2. **CanDeactivateFn**(Function base) →

```
----------------------- tasks.component.ts ----------------------
// We can write this function anywhere in our project
export const canLeaveEditPage: CanDeactivateFn<NewTaskComponent> = (component) => {
    if (component.enteredTitle() || component.enteredSummary())
        return window.confirm('Are you sure you want to leave?');
    return true;
}
------------------------- app.routes.ts ------------------------
const routes: Routes = [
    { path: '', redirectTo: 'tasks', pathMatch: 'full' },
    { path: 'tasks', component: TasksComponent,
      runGuardsAndResolvers: 'always', canMatch: [dummyCanMatchs],
    },
    { path: 'tasks/new', component: NewTaskComponent,
      canDeactivate: [canLeaveEditPage]
    }
];
```

Lazy Loading & Defer

Route-based lazy loading is a key feature in Angular, allowing modules to load only when needed. In Angular 17 and newer, there's an additional approach called *deferrable view* for lazy loading. In this section, we'll explore lazy loading in general and dive into the capabilities of deferrable views.

1. **Lazy-Loading With Routing →**

```
------------------------- tasks.routes.ts -------------------------
const routes: Routes = [
  { path: '', redirectTo: 'tasks', pathMatch: 'full' },
  {
    path: 'tasks',
     loadComponent: () => import('./tasks/tasks.component').then((mod) =>
mod.TasksComponent),
    runGuardsAndResolvers: 'always',
  },
  {
    path: 'tasks/new',
    component: NewTaskComponent,
    canDeactivate: [canLeaveEditPage]
  }
];

-------------------------- app.routes.ts --------------------------
const appRoutes: Routes = [
  {
    path: 'users/:userId',
    component: UserTasksComponent,
    runGuardsAndResolvers: 'always',
    loadChildren: () => import('./users/tasks.routes').then((mod) => mod.routes),
    canMatch: [CanMatchTeamSection],
    resolve: { userName: resolveUserName },
    title: resolveTitle
  }
];
```

2. **Lazy-Loading Services →**

   To enable lazy loading for a service, first, remove **{ providedIn: 'root' }** from the service file if it is present.

```
------------------------ tasks.routes.ts ------------------------
const routes: Routes = [
 {
   path: '', providers: [TasksService],
   children: [
    { path: '', redirectTo: 'tasks', pathMatch: 'full' },
    {
      path: 'tasks',
      loadComponent: () => import('./tasks/tasks.component').then((mod) =>
mod.TasksComponent),
      runGuardsAndResolvers: 'always',
    },
    {
      path: 'tasks/new', component: NewTaskComponent,
      canDeactivate: [canLeaveEditPage]
    }
   ]
 }
];
```

Deferrable Views

```
// Now, the app-offer-preview component will load lazily
@defer { <app-offer-preview /> }
```

Some Functionality with Defer →

```
// Now, the app-offer-preview component will load lazily
@defer(on viewport) { <app-offer-preview /> }
@placeholder { <p>We might have an offer...</p> }
```

[**NOTE**: By default, the viewport option triggers the deferred block when specific content enters the viewport.]

Other deferral triggers include:

on idle: Triggers once the browser enters an idle state.

on interaction: Triggers when the user interacts with the element (e.g., click or keydown).

on hover: Triggers when the mouse hovers over the element.

on immediate: Triggers immediately after the client finishes rendering.

on timer: Triggers after a specified duration, e.g., **`on timer(500ms)`**.

Prefetching with Defer →

```
// Now, the app-offer-preview component will load lazily
@defer(on interaction; prefetch on hover) {
    <app-offer-preview />
} @placeholder {
    <p>We might have an offer...</p>
}
```

[**NOTE**: When hovering over a specific element, the deferrable component's necessary resources load lazily and display upon interaction.]

For more details - https://angular.dev/guide/defer

Deploying Angular Apps - CSR, SSR, SGA

To deploy an Angular application, first build the app by running ng build or **npm run build**. This process will generate a directory called **dist**, which contains the files needed for deployment. Now, let's explore the different build options: Single Page Application (SPA), Server-Side Rendering (SSR), and Static Site Generation (SSG).

Single Page Application(SPA)

When we run **`npm run build`** without any additional configurations, we build our application as a Single Page Application(SPA). This means we are creating a client-side web application, where the architecture relies on a single HTML file hosted on a web server. This file loads all the Javascript code required to run our application.

In an SPA, the JavaScript code handles the rendering of the UI within the browser. Consequently, all UI rendering occurs on the client side through JavaScript served by the web host.

Disadvantages of SPA:

- The entire UI is rendered in users' browsers, which can lead to an initially empty page while content is loading.
- If rendering takes time due to a slow browser or other issues, users may encounter empty websites or missing content.
- Search engine optimization (SEO) can be challenging, as search engine crawlers may detect an empty site and not wait for the client-side JavaScript to render the content.

SPAs do not require a dynamic web server; a static host is sufficient to serve the built application. For SPA deployment, Follow the link - deployment

Server-Side Rendering(SSR)

In this mechanism, an Angular app's routes are rendered on-demand by a dynamic web server. When a user requests to visit a specific page, the server first handles the request and pre-renders the requested page. This pre-rendered page is then sent back to the user.

Advantages of SSR:

- Users receive a fully rendered HTML file with all the content, rather than an empty or nearly empty HTML file.

Disadvantages of SSR:

- If rendering a component takes longer due to data fetching, the user may experience delays in receiving a response.
- SSR introduces additional complexity to the deployment process.

Setting Up SSR →

1. To prepare our Angular application for server-side rendering, run the following command: **`ng add @angular/ssr`**.
2. Alternatively, when creating a new Angular project, you can add the SSR flag: **`ng new project-name --ssr`**
3. After executing the first command, you will be prompted with several questions to complete the setup for server-side rendering. This process updates TypeScript and Angular configuration files and adds various settings to support SSR. It also creates a **server.ts** file, which contains the Node.js server code used to serve your application.
4. Then run - **`npm run build`**. Angular will recognize that it should build the project for server-side rendering. Upon completion, you will find both a **`browser`** and a **`server`** directory within the **`dist/routing`** directory. Note that for SPA builds, only the **`browser`** directory is generated.

For deployment - https://firebase.google.com/docs/app-hosting/get-started

Local Storage Error

When using **`localStorage`** in an Angular project, you may encounter an error when running your application with the command: **`npm run serve:ssr:routing`**.

This error occurs because **`localStorage`** is a browser-side feature and is not available on the server side, leading to issues during server-side rendering.

<u>Resolving the Issue</u>:

To address this, you can use the **`afterNextRender`** function. This function is executed by Angular after the next component render cycle. While Angular prepares the HTML code for the components on the server, the actual rendering to the DOM occurs on the client side, where the DOM exists.

By using **`afterNextRender`**, you can ensure that any code relying on **`localStorage`** runs only after the client-side render cycle is complete. This provides a safe space for executing code that should only run in the browser.

Static Site Generation(SSG)

This technique is a combination of the two other options. Because the idea here is that we do have some server-side rendered application but that Angular routes are also pre-rendered at build time. So, when a user visits a certain page, that page might already exist because it was pre-rendered during the build process, it doesn't need to be rendered dynamically on the server when the user sends their request. Therefore, the page can be served quicker than on demand on the server.

WIth this technique, at least some pages are pre-rendered at build time. It's a page that wasn't pre-rendered and was dynamically rendered on the server but just as before, the page is hydrated on the client side in the browser and becomes a single-page application after it was received by the browser.

Using SSG, we still need a dynamic web server, unless we really pre-render all our application pages.

The disadvantage is that if we pre-render a page that fetches data and that data is fetched during the build process and never thereafter. So we might be showing some outdated data on our website until the application is built again and the pages are pre-rendered again.

<u>Setting Up SSR</u> →

1. Ready for server side rendering by running **`ng add @angular/ssr`**.
2. Now, we see **`prerender: true`** in the **angular.json** file. We know that our application must have some static pages which are not depend on userId or this type of dynamic data(e.g. NotFound page, AboutUs page). But if we want to prerender some components which depend on a dynamic route(For example,

user task page where user tasks are dependent on **userId**). To prerender the specific page we can declare the route in a txt file and then, we must add the file into the **angular.json** file.

```
------------------------ angular.json ------------------------
"options": {
   "server": "src/main.server.ts",
   "prerender": {
      "routesFile": "user-routes.txt"
   },
   "ssr": { "entry": "server.ts" }
}
------------------------ user-routes.ts ------------------------
/users/u1/tasks
/users/u2/tasks
```

3. Then run - `npm run build`.

Client Hydration

Client Hydration If you open the **src/app/app.config.ts** in our new SSR application, we'll notice one interesting **`provider: provideClientHydration()`**. This provider is crucial for enabling hydration - a process that restores the server-side rendered application on the client.

Hydration improves application performance by avoiding extra work to re-create DOM nodes. Angular tries to match existing DOM elements to the application's structure at runtime and reuses DOM nodes when possible. This results in a performance improvement measured using Core Web Vitals statistics, such as reducing the Interaction To Next Paint (INP), Largest Contentful Paint (LCP), and Cumulative Layout Shift (CLS). Improving these metrics also positively impacts SEO performance.

Without hydration enabled, server-side rendered Angular applications will destroy and re-render the application's DOM, which may result in visible UI flicker, negatively impact Core Web Vitals like LCP, and cause layout shifts. Enabling hydration allows the existing DOM to be reused and prevents flickering.

Event Replay and Improved Debugging

With Angular 18, event replay, a feature that significantly enhances the SSR experience, came into developer preview. Event replay captures user interactions during the SSR phase and replays them once the application is hydrated. This ensures that user actions, such as adding items to a cart, are not lost during the hydration process, providing a seamless user experience even on slow networks.

To enable event replay, you can add **`withEventReplay`** to your **`provideClientHydration`** call:

```
bootstrapApplication(App, {
    providers: [provideClientHydration(withEventReplay())],
  });
```

Furthermore, Angular 18 takes another step towards making SSR a first-class citizen by enhancing the debugging experience with Angular DevTools. Now, DevTools include overlays and detailed error breakdowns, empowering developers to visualize and troubleshoot hydration issues directly in the browser.

Constraints

However, hydration imposes a few constraints on your application that are not present without hydration enabled. Your application must have the same generated DOM structure on both the server and the client. The hydration process expects the DOM tree to have the same structure in both environments, including whitespaces and comment nodes produced during server-side rendering.

S M HEMEL
Senior Software Engineer
Email: smhemel.eu@gmail.com