Angular: Beyond Basics

Mimanshu Maheshwari

March 21, 2024

Contents

List of Figures

Chapter 1

Content Projection

1.1 Introduction

In this context, having data-driven DOM structures prove to be extremely helpful as one directive or component can be created such that it will allow to insert data dynamically. This brings in reusability, minimizing the time, effort and duplication of code thereby minimizing the need of re-designing any part of the application that is already created.

Content Projection in Angular is a powerful and flexible way to create reusable components with dynamic content. It facilitates passing content from a parent component to a child component. This makes it possible to insert content dynamically into a component's template from the outside. Content Projection is also known as **Transclusion**.

Content projection helps in padding the data types that cannot be passed with the @Input decorator:

- Inner HTML
- HTML Elements
- Styed HTML
- Other Components

Content Projection has three flavours:

- Single slot Content Projection
- Multi slot Content Projection
- Conditional Content Projection

1.2 Single Slot Content Projection

- In the Parent Component
  - Define a child component tag in the parent's tmeplate
  - Add content between the opening and closing tag of the child component. This content will be projected into the child component template

```
<app-child>
Projected content <!- content to be projected -->
</app-child>
```

Listing 1.1: app-parent.component.html

- In the child component
    - Use `<ng-content>` directive to specify where the projcted content should b placed.

```
<div>
<h2> Child Component content</h2>
<h3> Single Slot content projection</h3>
<ng-content></ng-content>
</div>
```

Listing 1.2: app-child.component.html

1.3 Multi Slot Content Projection

```
<app-child>
  <div slot="header">Header content</div>
  <div slot="body">Body content</div>
</app-child>
```

Listing 1.3: app-parent.component.html

```
<div>
  <h3>Multi slot content projection</h3>
  <ng-content></ng-content> <!- Display default content -->
  <div class="header">
    <ng-content select="[slot=header]"></ng-content> <!- Display specific content -->
  </div>
  <div class="body">
    <ng-content select="[slot=body]"></ng-content> <!- Display specific content -->
  </div>
</div>
```

Listing 1.4: app-child.component.html

1.4 Conditional Content Projection

If the component needs to conditionally render content or has to render content multiple times, then the component shoud accept a `<ng-template>` element which can contain the conditionally rendered content.

In these scenarios, usage of `<ng-content>` is not advisable. This is because, when the cosumer of the component delivers the content, the content gets initialized always, even if component doesn't have `<ng-content>` or the `<ng-content>` is present within an `ngIf` statement.

With `<ng-template>`, content is not renderd by default by Angular. It needs to be explicitly rendered. So you can render content based on any condition or how many ever times we want. Hence here, `<ng-template>` is more preferred over `<ng-content>`

```
@Component({
  selector: 'app-component',
  template: `
    <div *ngIf="isLoggedIn; else loggedOut"> Welcome back </div>
    <ng-template #loggedOut>Please Login!!</ng-template>
    `,
})
export class AppComponent{
  isLoggedIn = true;
}
```

Listing 1.5: app.component.ts

Chapter 2

Dynamic Components

2.1 Introduction

Consider a scenario where there is a need to build a dynamic ad banne. Various advertisements need to be displayed in the ad banner. Fresh ad components get added regularly. Sticking to a static component structure for AdComponent is hence close to impossible. Instead, there needs to be some way which can load a new offer component without a permanent mention to the component in the template of the AdComponent. This is why Angular introduced the concept of dynamic components. Dynamic components in Angular allows to generate and render components dynamically at runtime. This comes in extremely handy in the context of dynamic user interfaces or when there is a need for creation of components based on user interactions.
The anchor directive
Before considering to add components, an anchor point has to be defined which can act as a guide to Angular for informing where to insert components.

2.2 Demo

The ad banner uses a helper directive named `AdDirective` for marking insertion points in the template.

```
import {Directive, ViewContainerRef} from '@angular/core';
@Directive({
  selector: '[adHost]',
})
export class AdDirective {
  constructor(public viewContainerRef: ViewContainerRef){ }
}
```

Listing 2.1: ad.directive.ts

The directive will inject ViewContainerRef. This makes it possible for the directive to gain access to the view container of the element that will take care of hosting the dynamically added component.

```
export interface AdComponent {
  data:any;
}
```

Listing 2.2: ad.ts

A common `AdComponent` interface will be implemented by all components. This will help to standardize the API as it needs to pass data to the components.

```
import {Type} from '@angular/core';
export class AdItem{
  constructor(public component: Type<any>, public data: any){ }
}
```

Listing 2.3: ad-item.ts

`AdItem` assists in stating the type of component which has to be loaded and the data that has to be bound to the component.

```

import { Injectable } from '@angular/core';
import { AdItem } from './ad-item';
import { ProfileComponent } from './profile.component';
import { JobAdComponent } from './job-ad.component';
@Injectable()
export class AdService {
  getAds() {
    return [
    new AdItem(
    ProfileComponent,
    { name: 'Dr. Jack', bio: 'Awesome Cardiac' }
    ),
    new AdItem(
    ProfileComponent,
    { name: 'Dr. Mary', bio: 'Wonderful OB/GYN' }
    ),
    new AdItem(
    JobAdComponent,
    { headline: 'We are hiring for many positions', body: 'Bring in your resume today!' }
    ),
    new AdItem(
    JobAdComponent,
    { headline: 'We have openings in many departments', body: 'Apply today to make the best use of
        ↪ opportunities!' }
    )
    ];
  }
}
```

Listing 2.4: ad.service.ts

`AdService` will returns the actual ads which are required for making the ad campaign. The service's `getAds()` method return an array of `AdItem` objects.

```
import { Component, Input, OnDestroy, OnInit, ViewChild } from '@angular/core';
import { AdDirective } from './ad.directive';
import { AdItem } from './ad-item';
import { AdComponent } from './ad';
@Component({
  selector: 'app-ad-banner',
  template: `
  <div class="ad-banner-example">
  <h3>Advertisements Section</h3>
  <hr/>
  <ng-template adHost></ng-template>
  </div>
  `
})
export class AdBannerComponent implements OnInit, OnDestroy {
  @Input() ads: AdItem[] = [];
  currentAdIndex = -1;
  @ViewChild(AdDirective, {static: true}) adHost!: AdDirective;
  private clearTimer: VoidFunction | undefined;
  ngOnInit(): void {
    this.loadComponent();
    this.getAds();
  }
  ngOnDestroy() {
    this.clearTimer?.();
  }
  loadComponent() {
    this.currentAdIndex = (this.currentAdIndex + 1) % this.ads.length;
    const adItem = this.ads[this.currentAdIndex];
    const viewContainerRef = this.adHost.viewContainerRef;
    viewContainerRef.clear();
    const componentRef = viewContainerRef.createComponent<AdComponent>(adItem.component);
    componentRef.instance.data = adItem.data;
  }
  getAds() {
    const interval = setInterval(() => {
      this.loadComponent();
    }, 3000);
    this.clearTimer = () => clearInterval(interval);
  }
}
```

Listing 2.5: ad-banner.component.ts

Due to the use of `<ng-template>` in above code, Angular will know where it has to dynamically load components. The `<ng-template>` element is a preferred choice while creating dynamic components as it doesn't lead to rendering of any additional output.

```
import { Component, Input } from '@angular/core';
import { AdComponent } from './ad';
@Component({
  template: `
  <div>
  <h3>Job Opportunities:</h3>
  <h4>{{data.headline}}</h4>
  {{data.body}}
  </div>
  `
})
export class JobAdComponent implements AdComponent {
  @Input() data: any;
}
```

Listing 2.6: job-ad.component.ts

```
import { Component, Input } from '@angular/core';
import { AdComponent } from './ad';
@Component({
  template: `
  <div>
  <h3>Featured Profile:</h3>
  <h4>{{data.name}}</h4>
  <p>{{data.bio}}</p>
  </div>
  `
})
export class ProfileComponent implements AdComponent {
  @Input() data: any;
}
```

Listing 2.7: profile.component.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
import { AdBannerComponent } from './ad-banner.component';
import { AdDirective } from './ad.directive';
import { AdService } from './ad.service';
import { JobAdComponent } from './job-ad.component';
import { ProfileComponent } from './profile.component';
```

```
@NgModule({
  imports: [ BrowserModule ],
  providers: [ AdService ],
  declarations: [
  AppComponent,
  AdBannerComponent,
  JobAdComponent,
  ProfileComponent,
  AdDirective
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Listing 2.8: app.module.ts

```
import { Component, OnInit } from '@angular/core';
import { AdService } from './ad.service';
import { AdItem } from './ad-item';
@Component({
  selector: 'app-root',
  template: `
  <div>
  <app-ad-banner [ads]="ads"></app-ad-banner>
  </div>
  `
})
export class AppComponent implements OnInit {
  ads: AdItem[] = [];
  constructor(private adService: AdService) {}
  ngOnInit() {
    this.ads = this.adService.getAds();
  }
}
```

Listing 2.9: app.component.ts

Chapter 3

Stand alone components

3.1 Introduction

Consider the below use cases:

- A modal need to be created to display alerts, dialogs and other user interactions. This modal may be required to be part of many screens.
- Form controls like date pickers, checkboxes and radio buttons needs to be reused across many screens.
- Custom Charts or Graphs need to display custom content across many screens.
- Weather display widgets need to be displayed across many screens.

Usually when creating components in above scenarios, the components are usually attached to a certain module. If these components are however built as standalone components, the following benefits can be received:

- Get more flexibilty and control over their codebases.
- Being small, focused components, they become more easier to understand, test and maintain.
- Promote better code organization as grouping of components can be done based on purpose or functionality.

A Standalone component is a certain kind of component which is not necessarily part of any module in an Angular application. Prior to v14, if components are not attached to a module, Angular compilation would fail. Post v14, components can be created as standalone components which are independent of modules. It also possible to create stand alone pipes and directives.

Apart from creating standalone components, we can also create:

- Standalone pipes
- Standalone directives

Standalone components can be used with:

- Module-based components
- Other standalone components
- Loading routes
- Lazy loading

3.1.1 Generating standalone components in Angular

```
ng g c myStandAlone -standalone
```

update to app.module.ts is not mentioned as is usually mentioned when using ng g c command.

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
@Component({
  selector: 'app-my-stand-alone',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './my-stand-alone.component.html',
  styleUrls: ['./my-stand-alone.component.css']
})
export class MyStandAloneComponent {
}
```

Listing 3.1: my-stand-alone.component.ts

3.1.2 Using standalone component inside module

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { MyStandAloneComponent } from './my-stand-alone/my-stand-alone.component';
@NgModule({
  declarations: [
  AppComponent
  ],
  imports: [
  BrowserModule,
  AppRoutingModule,
  MyStandAloneComponent
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Listing 3.2: app.module.ts

3.1.3 Bootstrapping Angular Standalone component

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';
import { bootstrapApplication } from '@angular/platform-browser';
import { MyStandAloneComponent } from './app/my-stand-alone/my-stand-alone.component';
// platformBrowserDynamic().bootstrapModule(AppModule)
// .catch(err => console.error(err));
bootstrapApplication(MyStandAloneComponent, {
  providers:[]
})
```

Listing 3.3: main.ts

```
<body>
  <!-- <app-root></app-root> -->
  <app-my-stand-alone></app-my-stand-alone>
</body>
```

Listing 3.4: index.html

3.1.4 Routing with Standalone Component

```
{
  path: 'standalone',
  loadComponent: () =>
  import ('./my-stand-alone/my-stand-alone.component').then((m) => m.MyStandAloneComponent),
}
```

Listing 3.5: app-router.module.ts

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { MyStandAloneHomeComponent } from '../my-stand-alone-home/my-stand-alone-home.component';
@Component({
  selector: 'app-my-stand-alone',
  standalone: true,
  imports: [CommonModule, MyStandAloneHomeComponent],
  templateUrl: './my-stand-alone.component.html',
  styleUrls: ['./my-stand-alone.component.css']
})
export class MyStandAloneComponent {
}
```

Listing 3.6: my-stand-alone-home.component.ts

Chapter 4

Resolvers

4.1 Introduction

Consider a scenario where if there is a delay in fetching data from the server and the component is loaded empty. This kind of delay can lead to a bad user experience(UX). To improve this behavior, pre-fetching of the data from the server can be done and then should load the component.

This can be implemented using Resolvers in Angular. Using resolvers, pre-fetch the data so that data is ready to be displayed before a component is loaded. This also allows to handle errors efficiently before routing to a component.

A Resolver in Angular is a service class that should inherit the Resolve interface and overrides the resolve() method. A resolver can return the response of type observable or promise or any custom type.

Class based resolvers have been deprecated now and currently Angular supports functional resolvers. To inject a class, inject function can be used. So there is no need to generate a resolver(service file) using the Angular CLI. Instead a simple Typescript file can be created and a function of type ResolveFn need to be exported. This can then be used it in the route mapping.

```
export const ordersResolver : ResolveFn<any>=
(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) : Observable<any> => {
  const ordersService= inject(OrdersService)
  return ordersService.getOrders();
}
```

Listing 4.1: resolver.ts

Any number of resolvers can be configured. They will be executed in the mentioned order. If there are route guards configured, then first the guards execute and if fine to proceed, the resolvers get executed.

4.2 Demo

```
import { inject } from "@angular/core";
import { ActivatedRouteSnapshot, ResolveFn, RouterStateSnapshot } from "@angular/router";
import { Observable } from "rxjs";
import { OrdersService } from "./orders.service";
export const ordersResolver : ResolveFn<any>=
(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) : Observable<any> => {
  const ordersService= inject(OrdersService)
  return ordersService.getOrders();
}
```

Listing 4.2: orders-resolver.ts

```
import { inject } from "@angular/core";
import { ActivatedRouteSnapshot, ResolveFn, RouterStateSnapshot } from "@angular/router";
import { Observable } from "rxjs";
import { OrdersService } from "./orders.service";
export const ordersDetailsResolver : ResolveFn<any>=
(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) : Observable<any> => {
  const ordersService= inject(OrdersService)
  return ordersService.getOrderById(Number(route.paramMap.get('orderid')));
}
```

Listing 4.3: order-detail-resolver.ts

```
  import { Injectable } from '@angular/core';
import {of, Observable, delay} from 'rxjs';
const orders=[
  {orderid:1, bookName:"Brain Games for Clever Kids", author: 'Gareth Moore', publisher: 'Buster Books', price
      ↪ : 5.99, quantity: 1, total: 5.99},
  {orderid:2, bookName:"Brain Games for Clever Kids", author: 'Gareth Moore', publisher: 'Buster Books', price
      ↪ : 5.99, quantity: 1, total: 5.99},
  {orderid:3, bookName:"Grandparents' Bag of Stories", author: 'Sudha Murty', publisher: 'Penguin Random House
      ↪ ', price: 5.99, quantity: 1, total: 5.99},
  {orderid:4, bookName:"The Kids' Book of Awesome Riddles", author: 'Helen Rosenberg', publisher: 'Rockridge
      ↪ Press', price: 5.99, quantity: 1, total: 5.99},
  {orderid:5, bookName:"Great Stories for Children", author: 'Ruskin Bond', publisher: 'Rupa Publications',
      ↪ price: 5.99, quantity: 1, total: 5.99},
]
@Injectable({
  providedIn: 'root'
})
export class OrdersService {
  constructor() { }
  getOrders():Observable<any[]>{
    return of(orders).pipe(delay(5000))
  }
  getOrderById(orderid: number):Observable<any>{
    let order=orders.find(order=>order.orderid==orderid);
    return of(order).pipe(delay(5000))
  }
}
```

Listing 4.4: orders.service.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { AppComponent } from './app.component';
import { AppRoutingModule } from './app-routing.module';
import { PageNotFoundComponent } from './page-not-found/page-not-found.component';
import { LoginComponent } from './login/login.component';
import { OrdersComponent } from './orders/orders.component';
import { OrderDetailComponent } from './order-detail/order-detail.component';
@NgModule({
  imports: [BrowserModule, HttpClientModule, ReactiveFormsModule,FormsModule, AppRoutingModule],
  declarations: [AppComponent, LoginComponent, PageNotFoundComponent, OrdersComponent, OrderDetailComponent
      ↪ ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Listing 4.5: app.module.ts

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

Listing 4.6: app.component.html

```typescript
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { LoginComponent } from './login/login.component';
import { OrdersComponent } from './orders/orders.component';
import { PageNotFoundComponent } from './page-not-found/page-not-found.component';
import { OrderDetailComponent } from './order-detail/order-detail.component';
import { ordersResolver } from './orders-resolver';
import { ordersDetailsResolver } from './order-detail-resolver';
const appRoutes: Routes = [
{ path: '', redirectTo: '/login', pathMatch: 'full' },
{ path: 'login', component: LoginComponent },
{ path: 'orders', component: OrdersComponent, resolve:{orders: ordersResolver} },
{ path: 'order/:orderid', component: OrderDetailComponent, resolve: {order: ordersDetailsResolver} },
{ path: '**', component: PageNotFoundComponent }
];
@NgModule({
  imports: [
  RouterModule.forRoot(appRoutes)
  ],
  exports: [
  RouterModule
  ]
})
export class AppRoutingModule { }
```

Listing 4.7: app-routing.module.ts

```typescript
import { Component, OnInit } from '@angular/core';
import { OrdersService } from '../orders.service';
import { ActivatedRoute } from '@angular/router';
@Component({
  selector: 'app-orders',
  templateUrl: './orders.component.html',
  styleUrls: ['./orders.component.css']
})
export class OrdersComponent implements OnInit {
  orders!: any[];
  constructor(private activatedRoute:ActivatedRoute) { }
  ngOnInit(): void {
    this.activatedRoute.data.subscribe((data)=>{
      this.orders=data['orders'];
    });
  }
}
```

Listing 4.8: orders.component.ts

```html
<h3>Your orders</h3>
<hr/>
<table *ngIf="orders">
<thead>
<th>Order ID</th>
<th>Order Details</th>
</thead>
<tbody>
<tr *ngFor="let order of orders">
<td>{{order.orderid}}</td>
<td><a [routerLink]="['/order',order.orderid]">{{order.bookName}}</a> by {{order.author}} </td>
</tr>
</tbody>
</table>
```

Listing 4.9: orders.component.html

```css
table, tr, th, td { border: 1px solid black; padding: 3px; }
table{margin-top:20px}
```

Listing 4.10: orders.component.css

```typescript
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { OrdersService } from '../orders.service';
@Component({
  selector: 'app-order-detail',
  templateUrl: './order-detail.component.html',
  styleUrls: ['./order-detail.component.css']
})
export class OrderDetailComponent implements OnInit {
  order!:any;
  constructor(private route: ActivatedRoute){}
  ngOnInit(): void {
    this.route.data.subscribe((data)=>{
      this.order=data['order'];
    });
  };
}
```

Listing 4.11: order-detail.component.ts

```html
<h3>Order orders</h3>
<table *ngIf="order">
<thead>
<th>Order ID</th>
<th>Book Name</th>
<th>Author</th>
<th>Publisher</th>
<th>Price</th>
<th>Quantity</th>
<th>Total</th>
</thead>
<tbody>
<tr >
<td>{{order.orderid}}</td>
<td>{{order.bookName}}</td>
<td>{{order.author}} </td>
<td>{{order.publisher}}</td>
<td>{{order.price | currency }}</td>
<td>{{order.quantity}}</td>
<td>{{order.total | currency}}</td>
</tr>
</tbody>
</table>
```

Listing 4.12: order-detail.component.html

```typescript
import { Component } from '@angular/core';
import { FormBuilder, FormGroup } from '@angular/forms';
import { Router } from '@angular/router';
@Component({
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css'],
})
export class LoginComponent {
  invalidCredentialMsg!: string;
  loginForm!: FormGroup;
  constructor(
  private router: Router,
  private formbuilder: FormBuilder
```

```
    ) {
      this.loginForm = this.formbuilder.group({
        username: [],
        password: [],
      });
    }
    onFormSubmit(): void {
      const uname = this.loginForm.value.username;
      const pwd = this.loginForm.value.password;
      if (uname == pwd)
      this.router.navigate(['/orders']);
      else
      this.invalidCredentialMsg = 'Invalid Credentials. Try again.';
    }
  }
```

Listing 4.13: login.component.ts

```
<h3 style="position: relative; left: 60px">Login Form</h3>
<div *ngIf="invalidCredentialMsg" style="color: red">
{{ invalidCredentialMsg }}
</div>
<br />
<div style="position: relative; left: 20px">
<form [formGroup]="loginForm" (ngSubmit)="onFormSubmit()">
<p>User Name <input formControlName="username" /></p>
<p>
Password
<input
type="password"
formControlName="password"
style="position: relative; left: 10px"
/>
</p>
<p><button type="submit">Submit</button></p>
</form>
</div>
```

Listing 4.14: login.component.html

Chapter 5

Debouncing

5.1 Demo

```
<h2>Enter the text to search for: </h2>
<input type="text" (input)="onInput($event)"/>
```

Listing 5.1: search-component.html

```
import { Component } from '@angular/core';
import { Subject, debounceTime } from 'rxjs';
@Component({
  selector: 'app-search',
  templateUrl: './search.component.html',
  styleUrls: ['./search.component.css']
})
export class SearchComponent {
  private searchSubject=new Subject<String>();
  constructor(){
    this.searchSubject.pipe(
    debounceTime(300)).subscribe(searchTextValue=>{
      this.search(searchTextValue)
    });
  }
  onInput(event: Event){
    const inputElement=<HTMLInputElement>event.target;
    this.searchSubject.next(inputElement.value);
  }
  search(searchTextValue:String){
    // do search
    console.log('Searching for '+searchTextValue);

  }
}
```

Listing 5.2: search.component.ts

```
<app-search></app-search>
```

Listing 5.3: app.component.html

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { AppComponent } from './app.component';
import { AppRoutingModule } from './app-routing.module';
import { SearchComponent } from './search/search.component';
@NgModule({
  imports: [BrowserModule, HttpClientModule, ReactiveFormsModule,FormsModule, AppRoutingModule],
  declarations: [AppComponent, SearchComponent],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Listing 5.4: app.module.ts

Chapter 6

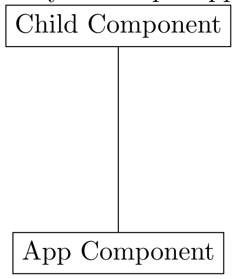
State Management

6.1 Introduction

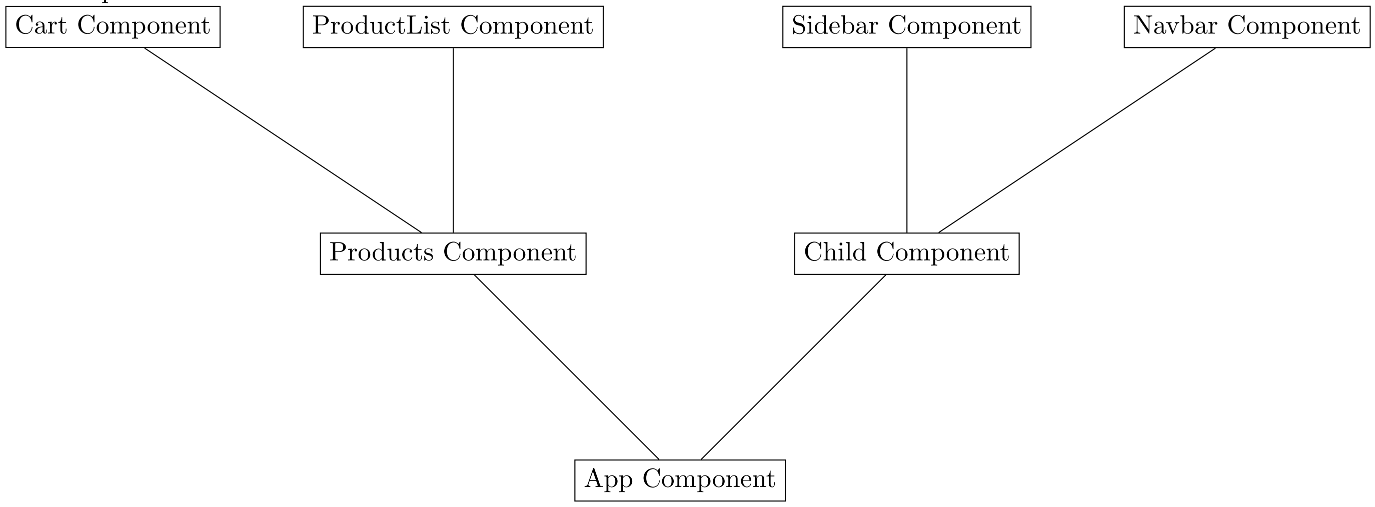
We have already covered most of the topics with respect to developing SPAs using angular, but we never used the term state management anywhere in the application. So, before we learn how to manage state, its important to know what is state ?

We will find a lot of definitions for State, but in simple words, "State is the data or the behavior of the app at a given instant". It includes both, the state of the UI as well as the state of the variables in the code. So, any change in either of these is a change in state of the application.

In angular, the application has a component-based architecture and each component maintains its own state and one component does not have any information about the state of other components until it is passed from one component to other using `@Input` or `@Output`. It is easier to exchange data between components in this way for simple apps as shown.



But with complex app architecture as shown, it gets very difficult and painful to exchange data between components. Consider if we had to pass some data from cart component to the navbar component.



State management is a technique that we use to handle the state of the application effectively to ensure the consistency of data across the application. We will be using ngRx for state management in angular.

Next we will learn about ngRx in detail.

6.2 ngRx

ngRx stands for Angular Reactive Extensions.

It is an open source library that helps to build reactive applications using angular. It is a great design pattern for predictable state management, which is inspired by Redux (a state management library for React). It aims to bring reactive extensions to angular and redux like a single store for the state in an application.

Redux Patter + RxJs + Angular = ngRx

Some of the features provided by ngRx are:

- State management
- Isolation of side effects
- Router bindings
- Entity collection management

ngRx should be used in medium/complex applications which involve a lot of user interactions and multiple data sources.

A good way to decide whether we need ngRx in our angular applications is SHARI principle of ngRx.

- Shared: state that is accessed by many components and services.
- Hydrated: state that is persisted and rehydrated from external storage.
- Available: state that needs to be available when re-entering routes.
- Retrieved: state that must be retrieved with a side-effect.
- Impacted: state that is impacted by actions from other sources

Let's consider an event that we all are likely to be familiar with - visiting a library to borrow a few books.

Let's say, one morning you wake up and want to read some books.

Rather than buying them, you decide to visit your local library to issue them instead. While going to the library there's just one intention or action you've got in mind i.e. BORROW (issue/borrow a book).

Here's where things will get interesting, in the above scenario, there are three main tasks that are happening:

- A Library which has all the books stored for not only you, but many others.
- The Customer (you) wants to access the books in library without much hassle
- You relied on someone else (a Librarian), to smoothly and efficiently let you borrow some books.
- And definitely not to forget the Book itself

You are being the customer of the Library, looking to borrow some books. You won't go directly to the Library's book racks and take a book on your own. Rather you need to rely upon the library system that includes a librarian to handover your required books to you. Thats where Redux pattern plays a role in ngRx for state management.

The previous scenario is very common, and is faced by almost all of us.

Your issued books in the library signifies your 'state' with respect to the Library. Library as institution manages everyone else's books along with managing your books specifically. For which, the Library has created different accounts (or library cards) to preserve the 'state' of each of its customers.

The library company can be thought of as ngRx as a whole. This scenario has four main actors involved:

- Book racks: This is the only single source of books, which is being used for everyone's transactions (borrow/returning/extension etc).
- You: You are the one who want to borrow (or return) the books. The library has your existing state with it, and you are relying on the library system to get your book out of the pool of books sitting inside the library.
- A Librarian: They are the agents for the actual management of money. A librarian will validate if you have the authority and the necessary availability of books for your transaction.
- The Book: The book rack is going to have a lot of books placed. We don't need the entire book rack, rather we want only the book of our choice.

A library transaction results in the manipulation of books or the 'state' in the Library. Hence, the librarian is the person who is responsible for changing from one 'state' to another.

Likewise, ngRx is similar to the library company. ngRx also has four main parts:

- Store - Store is the place where all the possible states of all the components of your application are stored. Store in ngRx can be understood as the book racks of a library where the books for all the customers are kept.
- Actions - Action is the one which initiates a request to change the 'state'. In our example, Action is analogous to you. As Action initiates state change, similarly, you are the one who initiates any book borrow or book return from your account.
- Reducers - These are the agents that change the state of app, in a smooth and predictable manner. Reducers are analogous to the librarian who are responsible for any borrow or return of books happening in the library.
- Selectors - When the customer borrows a book, the library does not provide the entire book rack to the customer. It just gives the required book to customer. The same job is done by the Selector in ngRx. It provides only the required states from the store.

Hence, the main actors in ngRx for any state management are Store, Reducers, Selectors and Actions. Next, lets see the principles of redux based on which ngRx works.

6.3 Redux Principles of ngRx

ngRx is based on the three fundamental principles of Redux:

- Single source of truth
- State is immutable
- Changes are made with pure functions

1. Single source of truth The state of complete application is stored in a single state object tree within a store. Having single store in a application makes it easy to create, debug and test. It also means components should read data from this single source and not keep their own version of the same state separately. The state of whole application is centralized and stores in a single state object. In ngRx, store is the single source of truth.
2. State is immutable State is immutable. The only way to change a state is by dispatching an action. Since whole application is stored in single state tree, having immutable state make sure that no views or network callbacks wrote directly to state. Instead, they dispatch an action, which have type property which indicates the type of action being performed.
3. Changes are made with pure functions Pure functions are those functions which do not change input value and always provide same output for same input. In Redux/ngRx, reducers are pure functions which takes previous state and an action as parameter and return new state.Since reducers are pure functions, it will never change the previous state, it will always return new state object.

6.4 ngRx Core Concepts

Then, based on the action we create, the changes must be applied into the state available in the **ngRx Store**. To apply those changes into the state based on the action, we use Reducers.

Once the new application state is produced, we need to select the state/data from the store that are required in the components and then pass it on to them, this job of selecting the data from the store and then passing on to the components is done by a Selector.

So, In Short, we can say, A component creates an **action** based on user **events**, which then goes to the reducer, based on the action the reducer updates the **store** and then **selector** passes on the required state to the respective components.

Now, the application might not be that simple always, some user events might also involve asynchronous http calls. To manage such asynchronous operations, we use **Effects**. These **Effects** will help us invoke the required services to interact with the database and hence bring back appropriate response which will be later used to update the store.

6.5 Action

Actions are one of the main building blocks in NgRx. Actions represent unique events in ngRx based angular application and each action updates the state differently. There are many ways in which an action can be created such as:

- user interaction through clicks, mouseovers, form submissions, etc...
- external interaction through network requests,
- direct interactions with device APIs, etc..

An action is made up of a simple interface containing a property type.

```
interface Action {
  type: string;
}
```

Listing 6.1: action.ts

The property type describes the action to be dispatched. The value for type property will be in the form '[Source] Event' where we use Source to specify source from where action was created. A simple of action is:

```
{ type: '[Counter Component] Increment' }
```

Listing 6.2: example-action

We can also pass additional data (also known as payload) with the action as given below

```
{ type: '[Login Page] Login', username: string; password: string;}
```

Listing 6.3: example-action

payload: payload is not compulsory to be passed with every action. The data type of the payload depends on the type of data that the actions needs to send to the reducer.

6.5.1 Action Creators

Actions can be created using the createAction method available from the @ngrx/store library. Following are few examples for createAction method:

```
import { createAction } from '@ngrx/store';
export const increment = createAction('[Counter Component] Increment');
/* increment() returns { type: '[Counter Component] Increment'} */
```

Listing 6.4: without payload

```
import { createAction, props } from '@ngrx/store';
export const login = createAction(
'[Login Component] Login',
props<{ username: string; password: string }>()
);
/* login('user1', 'user1') returns {type: '[Login Component] Login', username:'user1', password:'user1' } */
```

Listing 6.5: with payload

createAction function will return a function that when called will return an object as shown before in this page. props() method helps to payload data.

createAction just returns an action object which then must be dispatched based on any event that occurs due to user interaction. An action can be dispatched from the components using the dispatch method of store as given below:

```
this.store.dispatch(increment()); /* Without Payload */
this.store.dispatch(login("user1", "user1")); /* With Payload */
```

Listing 6.6: example

6.6 Reducer

Reducers are the pure function that handles the state of the application based on the type of the action triggered. It takes the previous state and an action as an argument and returns a new state and hence, it transitions the application from one state to another. There are a few consistent parts of every state managed by a reducer which includes:

- An Interface or a data type to define the type of state
- It will always have two parameters i.e. initial/current state and current action
- Function to handle the state changes for the the current action

6.6.1 example

First thing we need to do is define the type of the State using an interface as given below:

```
export interface State {
  counter : number;
}
```

Listing 6.7: state.ts

Once the state's type has been defined, we also need to create an initial state object of the type State.

```
export const initialState: State = {
  counter: 0
};
```

Listing 6.8: initial-state.ts

After defining the initial state, We need to define the reducer function that will handle the different actions. Reducers can be created using the createReducer method available from the @ngrx/store library. Following is an example to use createReducer method:

```
import { createReducer, on } from '@ngrx/store';
import { increment, decrement, reset } from './counter.actions'; /* This file contains all the actions */
const _counterReducer = createReducer(initialState,
    on(increment, state => { return { ...state, counter: state.counter + 1 } }),
    on(decrement, state => { return { ...state, counter: state.counter - 1 } }),
    on(reset, state => initialState),
);
export function counterReducer(state, action) {
    return _counterReducer(state, action);
}
```

Listing 6.9: counter-reducer.ts

Note: The exported counterReducer function is necessary as function calls are not supported by the AOT compiler.

In the above example, counterReducer handles 3 actions: '[Counter Component] Increment', '[Counter Component] Decrement' and '[Counter Component] Reset'. Each of these actions are strongly typed and handle the state changes immutably i.e. state transition does modify the original state instead it returns a new state object using spread operator.

When an action is dispatched, ngRx will go through all the reducers by passing the previous state and the Action till it finds a reducer to handle the action. Whether the reducer handles the action or not, is determined by the on function.

6.7 Store

Store is a container that holds the entire application's state.
The store helps us to keep the application data in one place, and also which enables us to use the store as a single source of truth, i.e. we can reliably access the application's state from the store rather than components of the app holding their own state and passing data between then on need. This reduces the communication between the components, which helps to scale the application without adding more complexity.

To access and manipulate the global store, we must first register it within our angular application along with the reducer function that should change the state of the application. To register the global store within our application, we use `StoreModule.forRoot()` method with a map of key-value pairs that define our state.

Let us see how to set up an application store:

We can configure the `app.module` with `@ngrx/store` module and reducer as follows, by importing the application reducer into the `StoreModule`:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
import { StoreModule } from '@ngrx/store';
import { counterReducer } from './counter.reducer';

@NgModule({
  declarations: [AppComponent],
  imports: [
    BrowserModule,
    StoreModule.forRoot({ count : counterReducer })
  ],
  providers: [],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

Listing 6.10: app.module.ts

Registering states with `StoreModule.forRoot()` ensures that the states are defined upon application startup. In general, we register root states that always need to be available to all areas of your application immediately. We can also configure the reducers and state for lazy loaded feature modules using `forFeature()` method. The store also provide us with a method dispatch which is used to dispatch actions to update the state.

6.7.1 Advantages of Store based application:

The primary advantages of a store based application are:

1. Centralized and Immutable State: As all the relevant data/state is kept at a single place, it makes the tracking of issues/bugs in the applications easier.

2. Performance: As the entire state of the application is centralized, the components can access the updated data directly from there and this improves the performance of the application as the data exchange is much simpler and faster.

3. Testability: Reducer functions are pure functions and they are the ones which handle all the state updates. Pure functions are also easier to test as it simply involves input in, assert against output.

4. Tooling and Ecosystem: As the state is centralized and immutable, it also empowers us with better tooling. for example, ngRx developer tools, which maintain a history of all the dispatched actions and state changes done which allows time travel during development.

6.8 Selector

Selectors are pure functions used for obtaining slices of store state. `@ngrx/store` provides a few helper functions for optimizing this selection. Selectors provide many features when selecting slices of state such as **Portability**, **Memoization**, **Composition**, **Testability** and **Type Safety**. We can create selectors using the functions `createSelector` or `createFeatureSelector`.

The `createSelector` function can be used to select only the required data for a component from all the entire state. We can pass upto 8 selector functions as parameter to `createSelector` method for better state selection. Lets see a simple example with a single selector function:

```
import { createSelector } from '@ngrx/store';
export interface FeatureState {
   counter: number;
}
export interface AppState {
   count: FeatureState;
}
export const selectFeature = (state: AppState) => { return state.count };
export const selectFeatureCount = createSelector(
   selectFeature,
   (state: FeatureState) => state.counter
);
```

Listing 6.11: create-selector-example.ts

Then to fetch the value of counter function in respective components, we need to use select method by passing the selector method i.e. `selectFeatureCount` as given below:

```
this.count$ = this.store.pipe(select(fromRoot.selectFeatureCount))
```

Listing 6.12: use the selector created

When these functions are used to create the selectors, `@ngrx/store` keeps track of the latest arguments in which the selector function was invoked. As Selectors are pure functions, it can return the last result without re invoking the selector function, if the arguments match with the previous one. This can help in boosting the performance of the application and is also called as **Memoization**.

6.9 ngRx Counter App - Demo

Problem Statement: Create an angular app with counter component as shown using ngRx core concepts i.e. Store, Actions, Reducers and Selectors.

1. Create an angular app and install `@ngrx/store` in the application using the below command

```
      npm install @ngrx/store --save
```

2. Create *app/Store/actions/counter.actions.ts* within src with the following code:

```
import { createAction, props } from '@ngrx/store';
export const increment = createAction('[Counter Component] Increment');
export const decrement = createAction('[Counter Component] Decrement');
export const multiply = createAction('[Counter Component] Multiply', props<{ multiplyBy: number
    ↪ }>());
export const reset = createAction('[Counter Component] Reset');
```

Listing 6.13: app/Store/actions/counter.actions.ts

3. Create *app/Store/reducers/counter.reducer.ts* within src with the following code:

```
import { createReducer, on } from '@ngrx/store';
import { increment, decrement, reset, multiply } from '../actions/counter.action';
export const initialState = {
    counter: 0
};
const _counterReducer = createReducer(initialState,
    on(increment, state => { return { ...state, counter: state.counter + 1 } }),
    on(decrement, state => { return { ...state, counter: state.counter - 1 } }),
    on(multiply, (state, { multiplyBy }) => { return { ...state, counter: state.counter *
        ↪ multiplyBy } }),
    on(reset, state => initialState),
);
export function counterReducer(state, action) {
    return _counterReducer(state, action);
}
```

Listing 6.14: app/Store/reducers/counter.reducer.ts

4. Create *app/Store/selectors/counter.selector.ts* within src with the following code:

```
import { createSelector } from '@ngrx/store';
export interface FeatureState {
    counter: number;
}
export interface AppState {
    count: FeatureState;
}
export const selectFeature = (state: AppState) => { return state.count };
export const selectFeatureCount = createSelector(
    selectFeature,
    (state: FeatureState) => state.counter
);
```

Listing 6.15: app/Store/selectors/counter.selector.ts

5. Create a component **"counter"** and add the following code to *counter.component.ts*

```
import { Component, OnInit } from '@angular/core';
import { Store, select } from '@ngrx/store';
import { Observable } from 'rxjs';
import { increment, decrement, reset, multiply } from '../Store/actions/counter.action';
import * as fromRoot from '../Store/selectors/counter.selector';
@Component({
  selector: 'app-counter',
  templateUrl: './counter.component.html',
  styleUrls: ['./counter.component.css']
})
export class CounterComponent implements OnInit {
  count$: Observable<number>;
  constructor(private store: Store<fromRoot.AppState>) {
  }
  ngOnInit(): void {
    this.count$ = this.store.pipe(select(fromRoot.selectFeatureCount))
  }
  increment() {
    this.store.dispatch(increment());
  }
  decrement() {
    this.store.dispatch(decrement());
  }
  reset() {
    this.store.dispatch(reset());
  }
  multiplyByTwo() {
    this.store.dispatch(multiply({ multiplyBy: 2 }))
  }
  multiplyByThree() {
    this.store.dispatch(multiply({ multiplyBy: 3 }))
  }
}
```

Listing 6.16: counter.component.ts

6. Add the following code to *counter.component.html*

```
<div class="card shadow row col-md-6 offset-md-3 mt-5 text-center">
  <div class="card-body">
      <h3 style="font-family: cursive;">Current Count: {{ count$ | async }}</h3>
      <button class="btn btn-outline-danger mr-2" id="increment" (click)="increment()">Increment</button>
      <button class="btn btn-outline-success mr-2" id="decrement" (click)="decrement()">Decrement</button>
      <button class="btn btn-outline-dark mr-2" id="reset" (click)="reset()">Reset Counter</button>
      <button class="btn btn-outline-warning mr-2" id="multiply" (click)="multiplyByTwo()">Multiply By 2</button>
      <button class="btn btn-outline-info" id="multiply" (click)="multiplyByThree()">Multiply By 3</button>
  </div>
</div>
```

Listing 6.17: counter.component.html

7. *app.module.ts*

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { CounterComponent } from './counter/counter.component';
import { StoreModule } from '@ngrx/store';
import { counterReducer } from './Store/reducers/counter.reducer';
@NgModule({
  declarations: [
    AppComponent,
    CounterComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    StoreModule.forRoot({ count: counterReducer })
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Listing 6.18: app.module.ts

6.10 Meta Reducers

@ngrx/store helps us to compose all the reducers together into a single reducer. Meta Reducer is a higher order reducer function that accepts reducers and returns a new reducer.

As developers, we can consider meta reducer as a hook between action → reducer pipeline and it helps pre process the action before the normal reducers are invoked. Some of the common use cases for meta reducers are:

- Reset State once a user session terminates
- Loggers to help developers in debugging
- Rehydrating once the app restarts

Meta reducers are the reducer functions which should be executed irrespective of the action type.

Meta reducers are very similar to the **middlewares** in redux.

6.10.1 Demo

Problem Statement: Log all the dispatched actions and state changes in browsers console

Update the code in *app.module.ts* with the code given below in the counter-app used in previous demo:

```
1  import { BrowserModule } from '@angular/platform-browser';
2  import { NgModule } from '@angular/core';
3  import { AppRoutingModule } from './app-routing.module';
4  import { AppComponent } from './app.component';
5  import { CounterComponent } from './counter/counter.component';
6  import { StoreModule, ActionReducer, MetaReducer } from '@ngrx/store';
7  import { counterReducer } from './Store/reducers/counter.reducer';
8  /* console.log all actions and state changes */
9  export function debug(reducer: ActionReducer<any>): ActionReducer<any> {
10   return function (state, action) {
11     console.log('previous state', state);
12     console.log('action', action);
13     let nextState = reducer(state, action);
14     console.log('current state', nextState);
15     return nextState;
16   };
17 }
18 export const metaReducers: MetaReducer<any>[] = [debug];
19 @NgModule({
20   declarations: [
21     AppComponent,
22     CounterComponent
23   ],
24   imports: [
25     BrowserModule,
26     AppRoutingModule,
27     StoreModule.forRoot({ count: counterReducer }, { metaReducers })
28   ],
29   providers: [],
30   bootstrap: [AppComponent]
31 })
32 export class AppModule { }
```

Listing 6.19: app.module.ts

Line 6: ActionReducer and MetaReducers are imported from **@ngrx/store**. `ActionReducer` is a function that takes an Action and a state and return another State. It is used to ensure, debug function takes a reducer function as a parameter.

Line 9: debug is a reducer function that takes a `ActionReducer` as a parameter and returns another `ActionReducer` ↪ . It returns another function that logs the dispatched action, state before and after the action was handled by the corresponding reducer.

Line 18: metaReducer is an array of meta-reducers to be used, in this case we are using only one i.e. debug meta-reducer.
Line 27: metaReducer has been configured to the application using `StoreModule.forRoot()`.

6.11 ngRx Dev Tools

Store Devtools provides developer tools and instrumentation for Store. To use ngrx devtools, we first need to install @ngrx/store-devtools:

```
npm install @ngrx/store-devtools --save
```

Setup
Follow the given steps to configure the store devtools for chrome:

1. Add the Redux Devtools Extension to your chrome from the chrome web store.

2. In app.module.ts, add the module imports using StoreDevtoolsModule.instrument as shown below:

```
  /* Add the following import statements to the file */
import { StoreDevtoolsModule } from '@ngrx/store-devtools';
import { environment } from '../environments/environment';
/* Add the following code to imports array of AppModule */
StoreDevtoolsModule.instrument({
      maxAge: 25,
      logOnly:environment.production
    })
```

Listing 6.20: app.module.ts

maxAge:25 will ensure that the last 25 state changes data is maintained in the devtools. logOnly will connect to the devtools extension in log-only mode.By default it is false, which enables all the extension features. The devtools has many more other options available.

6.12 Demo : NgRx/store

Understanding the core modules of ngRx through a counter-demo Creating action, reducer, and store

1. Create an angular application and install ngrx/store in the same folder:

```
    npm install @ngrx/store
```

2. Create *app/actions/counter.actions.ts* within src as shown below:

```
    import { Action } from '@ngrx/store';
export const INCREMENT_COUNT = 'INC_COUNT';
export const DECREMENT_COUNT = 'DEC_COUNT';
export class IncrementCounter implements Action {
  readonly type = INCREMENT_COUNT;
  constructor(public payload: number) {
  }
}
export class DecrementCounter implements Action {
  readonly type = DECREMENT_COUNT;
  constructor(public payload: number) {
  }
}
export type Actions = IncrementCounter | DecrementCounter;
```

Listing 6.21: app/actions/counter.actions.ts

3. Create *app/reducers/counter.reducer.ts* within src as shown below:

```
    import * as counterActions from './../actions/counter.actions';
const initialState = {
  counter: 10
};
export function counterReducer(state = initialState, action: counterActions.Actions) {
  switch (action.type) {
    case counterActions.INCREMENT_COUNT: {
      console.log(state);
      return {
        ...state,
        counter: state.counter + action.payload
      };
    }
    case counterActions.DECREMENT_COUNT: {
      console.log('DECREMENT_COUNT');
      console.log(state);
      return {
        ...state,
        counter: state.counter - action.payload
      };
    }
    default: {
      return state;
    }
  }
}
```

Listing 6.22: app/reducers/counter.reducer.ts

4. Create *app/counter/counter.component.ts* within src as shown below:

```
    import { Component, OnInit } from '@angular/core';
import { Store, select } from '@ngrx/store';
import * as counterActions from './../actions/counter.actions';
@Component({
  selector: 'app-counter',
  templateUrl: './counter.component.html',
  styleUrls: ['./counter.component.css']
})
export class CounterComponent implements OnInit {
  constructor(private store: Store<any>) { }
  count: number = 0;
  ngOnInit() {
    this.store.select('counterReducer').subscribe(
      counter => {
        console.log("sub", counter);
        if (counter) {
          this.count = counter.counter;
        }
      }
    );
  }
  onAddCount(counter?: number) {
    console.log(counter);
    this.store.dispatch(new counterActions.IncrementCounter(counter),
    );
  }
  onDecrementCount(counter?: number) {
    console.log(counter);
    this.store.dispatch(new counterActions.DecrementCounter(counter),
    );
  }
}
```

Listing 6.23: app/counter/counter.component.ts

5. Create *app/counter/counter.component.html* within src as shown below:

```
  <div style="background-color:skyblue ;width:500px;"><br/>
  <h2>&nbsp;&nbsp;Counter</h2><br/>
  <div>
      &nbsp; &nbsp;
  <button (click)="onAddCount(2)" value="click">Increment</button>
&nbsp;&nbsp;&nbsp;
  <button (click)="onDecrementCount(2)" value="click">Decrement</button>
<br/><br/>
 <h5>&nbsp;&nbsp;&nbsp;Counted: {{count}}</h5> <br/><br/>
</div>
<div style="background-color:skyblue ;width:500px;"><br/>
  <h2>&nbsp;&nbsp;Counter</h2><br/>
  <div>
      &nbsp; &nbsp;
  <button (click)="onAddCount(2)" value="click">Increment</button>
&nbsp;&nbsp;&nbsp;
  <button (click)="onDecrementCount(2)" value="click">Decrement</button>
<br/><br/>
 <h5>&nbsp;&nbsp;&nbsp;Counted: {{count}}</h5> <br/><br/>
</div>
```

Listing 6.24: app/counter/counter.component.html

6. Create *app/app.component.ts* within src as shown below:

```
  import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: `<div >
  <h4>
  Counter demo which explains NgRx modules
  </h4>
</div>
<app-counter></app-counter>`,
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'ngrx demo app';
}
```

Listing 6.25: app/app.component.ts

7. Create *app/app.module.ts* within src as shown below:

```
  import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
import { CounterComponent } from './counter/counter.component';
import { counterReducer } from './reducers/counter.reducer';
import { StoreModule } from '@ngrx/store';
@NgModule({
  declarations: [
    AppComponent,
    CounterComponent
  ],
  imports: [
    BrowserModule,
    StoreModule.forRoot({counterReducer: counterReducer})
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Listing 6.26: app/app.module.ts

6.13 ngRx Effects

In any service based application which interacts with a database server, component is the one which interacts with external resources directly through services. So, the component is responsible to manage the view dynamically as well as interacting with external sources.

Effects help us handle this more efficiently. Effects provide us a way to interact with the services and isolate them from the components. Effects are an RxjS powered side effect model for Store. Effects uses streams and provide us with new sources of action to handle the state based on external interactions such as long running tasks which produce multiple events, web-socket messages and network requests to fetch, add or modify data. The use of Effects becomes much more important in larger applications as they may have multiple sources of data with multiple services required to fetch those data

Features of Effects

- It helps to have more pure components that can select the state from the store and dispatch actions, by isolating the side effects.
- Effects are long running services which listens to every action dispatched.
- Effects use RxJS operators to filter the actions based on the type of action.
- Effects can handle synchronous as well as asynchronous tasks and return a new action.

To use ngRx Effects in our angular applications to perform the side effects, we need to install it using the below command:

```
npm install @ngrx/effects --save
```

6.13.1 Writing ngRx Effects

We can use Effects to isolate the side effects/external interactions from the components by creating a Effect class which listens to events and performs taskst. Effect class is an injectable service class and has the following distinct parts:

- An injectable Actions service which provides an observable of all the dispatched actions after the latest state changes.

```
    /* Example */
import { Injectable } from '@angular/core';
import { Actions } from '@ngrx/effects';
@Injectable()
export class ProductEffects {
  constructor(private actions$: Actions) { }
}
```

Listing 6.27: product-effects.ts

- Metadata is attached to the observable of dispatched actions using `createEffect` function of @ngrx/-effects. The `createEffect` function registers the streams that are subscribed to the store. Any action returned from the effect stream is then dispatched back to the Store.

```
    /* Example */
import { Injectable } from '@angular/core';
import { Actions, createEffect } from '@ngrx/effects';
@Injectable()
export class ProductEffects {
    constructor(private actions$: Actions) { }
    loadProducts$ = createEffect(() => this.actions$.pipe());
}
```

Listing 6.28: product-effects.ts

- Actions are filtered using the ngRx effects pipeable ofType operator. It takes one or more action types as argument to filter those on which it needs to act upon.

```
    /* Example */
import { Injectable } from '@angular/core';
import { Actions, createEffect, ofType } from '@ngrx/effects';
@Injectable()
export class ProductEffects {
    constructor(private actions$: Actions, private productService: ProductService) { }
    loadProducts$ = createEffect(() => this.actions$.pipe(
        ofType('[ProductList Component] GET_ALL_PRODUCTS')
    ));
}
```

Listing 6.29: product-effects.ts

- Effects are subscribed to the Store observable and Services are injected into the effects to interact with the external API's.

```
    import { Injectable } from '@angular/core';
import { Actions, createEffect, ofType } from '@ngrx/effects';
import { of } from 'rxjs';
import { ProductService } from '../../product-list/product.service';
import { map, mergeMap, catchError } from 'rxjs/operators';
@Injectable()
export class ProductEffects {
    constructor(private actions$: Actions, private productService: ProductService) { }
    loadProducts$ = createEffect(() => this.actions$.pipe(
        ofType('[ProductList Component] GET_ALL_PRODUCTS'),
        mergeMap(() => this.productService.getAllProducts()
            .pipe(
                map(products => ({ type: '[ProductList Component] GET_ALL_PRODUCTS SUCCESS', allProducts:
                     ↪ products })),
                catchError(() => of({ type: '[ProductList Component] GET_ALL_PRODUCTS ERROR', errorMessage
                     ↪ : 'No Products Found' }))
            ),
        ),
    )
    );
}
```

Listing 6.30: product-effects.ts

The flow Explanations:

- The `loadProducts$` effect is listening for all dispatched actions through the Actions stream, but is only interested in the '[ProductList Component] GET_ALL_PRODUCTS' event using the `ofType` operator.
- The stream of actions is then flattened and mapped into a new observable using the `mergeMap` operator.
- The `productService#getAllProducts()` method returns an observable that maps the products to a new action on success, and currently returns an empty observable if an error occurs.

- The action is dispatched to the Store where it can be handled by reducers when a state change is needed.
- It's also important to handle errors when dealing with observable streams so that the effects continue running.

Angular App flow with ngRx Effects

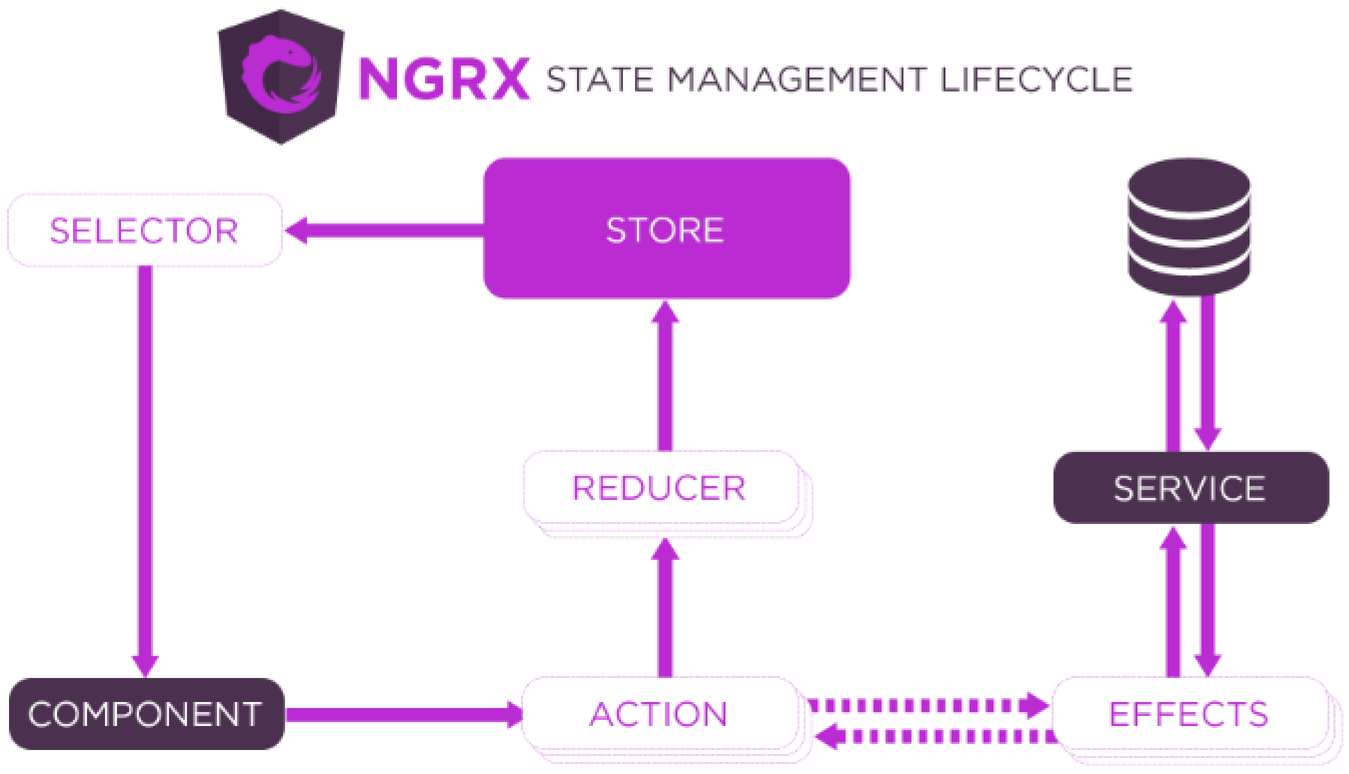


Figure 6.1: Angular App flow with ngRx Effects

1. Component C Dispatches an action A.
2. The action A invokes an Effect E.
3. The Effect E then invokes a service Srv to interact with the database. After service call the Effect E again returns an action, either A1 or A2 depending upon service call success or error.
4. A1/A2 should not be the same as A (otherwise, this will lead to infinite loop).
5. These actions A1/A2 then go to the reducer R and update the store S.
6. Once the store S has been updated, the component C can fetch the value using a Selector SS.

6.13.2 ngRx Effects - Demo

Problem Statement: Create a component product-list that uses ngRx effects to make API call for loading the list of products is success case and displays error message is if data not found

To achieve the above requirement, Follow the below steps:

1. Create an angular app and install @ngrx/effects in the angular application:

```
npm install @ngrx/effects --save
```

2. Define a model class for the Products data in the file *product-list/products.ts*:

```
export class Product {
  id: number=0;
  productName: string="";
  productPrice: number=0;
  productImg: string="";
  productBrand: string="";
}
```

Listing 6.31: product-list/products.ts

3. Define the actions for handling both success and error cases of API calls in the file *Store/actions/products.actions.ts*:

```
import { createAction, props } from '@ngrx/store';
import { Product } from '../../product-list/products';
export const getAllProducts = createAction('[ProductList Component] GET_ALL_PRODUCTS')
export const getAllProductsAPISuccess = createAction('[ProductList Component] GET_ALL_PRODUCTS
    ↪ SUCCESS', props<{ allProducts: Product[] }>())
export const getAllProductsAPIError = createAction('[ProductList Component] GET_ALL_PRODUCTS ERROR'
    ↪ , props<{ errorMessage: String }>())
```

Listing 6.32: Store/actions/products.actions.ts

`getAllProducts` is the action to be dispatched on load of product-list component, whereas `getAllProductsAPISuccess` ↪ and `getAllProductsAPIError` are the actions dispatched by the Effect once it completes the API/service call with success or error response

4. Define the **Initial state** for the application in the file *Store/states/products.store.ts*:

```
export const initialProductState = {
  allProducts: [],
  errorMessage: ""
}
```

Listing 6.33: Store/states/products.store.ts

allProducts is an array that will store an array of product objects fetched from service call whereas errorMessage will store the error response in case the service call does not return the required data.

5. Define the reducer for handling both success and error cases of API calls in the file *Store/reducers/products.reducer.ts*:

```
import { createReducer, on } from '@ngrx/store';
import { getAllProducts, getAllProductsAPISuccess, getAllProductsAPIError } from '../actions/
    ↪ products.actions';
import { initialProductState } from '../states/products.store';
const _productReducer = createReducer(initialProductState,
on(getAllProducts, (state) => state),
on(getAllProductsAPISuccess, (state:any, { allProducts }) => { return { ...state, allProducts:
    ↪ allProducts, errorMessage: "" } }),
on(getAllProductsAPIError, (state:any, { errorMessage }) => { return { ...state, errorMessage:
    ↪ errorMessage, allProducts: [] } })
);
export function productReducer(state:any, action:any) {
  return _productReducer(state, action);
}
```

Listing 6.34: Store/reducers/products.reducer.ts

productReducer is the reducer function that handles actions dispatched by the components as well as the effects after making the API calls

6. Define the selectors in the file *Store/selector/products.selector.ts* to ensure the component gets only the required data:

```
import { createSelector } from '@ngrx/store';
import { Product } from '../../product-list/products';
export interface FeatureState {
  allProducts: Product[];
  errorMessage: String;
}
export interface AppState {
  products: FeatureState;
}
export const selectFeature = (state: AppState) => { return state.products };
export const selectError = (state: AppState) => { return state.products };
export const selectFeatureProduct = createSelector(
selectFeature,
(state: FeatureState) => state.allProducts
);
export const selectFeatureError = createSelector(
selectError,
(state: FeatureState) => state.errorMessage
);
```

Listing 6.35: Store/selector/products.selector.ts

FeatureState defines the structure of the data stored in the store object selectFeatureProduct will return the products array from the global array whereas selectFeatureError will return the errorMessage set by the reducer

7. Create the Effects that invokes the appropriate service method to fetch all the products from server:

```
import { Injectable } from '@angular/core';
import { Actions, createEffect, ofType } from '@ngrx/effects';
import { ProductService } from '../../product-list/product.service';
import { map, mergeMap, catchError } from 'rxjs/operators';
@Injectable()
export class ProductEffects {
  constructor(private actions$: Actions, private productService: ProductService) { }
  loadProducts$ = createEffect(() => this.actions$.pipe(
  ofType('[ProductList Component] GET_ALL_PRODUCTS'),
  mergeMap(() => this.productService.getAllProducts()
  .pipe(
  map(products => ({ type: '[ProductList Component] GET_ALL_PRODUCTS SUCCESS', allProducts:
      ↪ products })),
  catchError(() => of({ type: '[ProductList Component] GET_ALL_PRODUCTS ERROR', errorMessage: 'No
      ↪ Products Found' }))
  ))));
}
```

Listing 6.36: product-effects.ts

createEffect is the method used to define an Effect, as discussed previously.

8. Create a Service product with the following code to fetch all the products from the server:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { Product } from './products';
@Injectable({
  providedIn: 'root'
})
export class ProductService {
  constructor(private http: HttpClient) { }
  getAllProducts(): Observable<Product[]> {
    return this.http.get<Product[]>("http://localhost:4000/products");
  }
}
```

Listing 6.37: product.service.ts

`getAllProducts()` is a service method which makes an API call to fetch all the products data from the service

9. Create a product-list component and add the following code to dispatch an action to fetch all the products on load of the component:

```
import { Component, OnInit } from '@angular/core';
import { Product } from './products';
import { Store, select } from '@ngrx/store';
import { Observable } from 'rxjs';
import * as fromRoot from '../Store/selector/products.selector';
@Component({
  selector: 'app-product-list',
  templateUrl: './product-list.component.html',
  styleUrls: ['./product-list.component.css']
})
export class ProductListComponent implements OnInit {
  constructor(private store: Store<fromRoot.AppState>) { }
  products$: Observable<Product[]> = this.store.pipe(select(fromRoot.selectFeatureProduct))
  errorMessage$: Observable<String> = this.store.pipe(select(fromRoot.selectFeatureError))

  ngOnInit(): void {
    this.store.dispatch({ type: '[ProductList Component] GET_ALL_PRODUCTS' });
  }
}
```

Listing 6.38: product-list.component.ts

`products$` is the property that stores the products array fetched from the global store. `errorMessage$` is the property that stores the error message fetched from the global store. An action of type '[ProductList Component] GET_ALL_PRODUCTS' is dispatched on load of component using `ngOnInit()` method.

10. Add the following code to *product-list.component.html* file:

```
<div class="row mt-4 text-center" style="font-family: cursive;">
<div class="col-md-3" *ngFor="let product of products$ | async; let i = index">
<div class="card shadow">
<img src="{{product.productImg}}" alt="{{product.productName}}">
<div class="card-body">
<h5>{{product.productName}}</h5>
<h6>Brand: {{product.productBrand}}</h6>
<h6 class="text text-success">Price: {{product.productPrice | currency:'INR'}}</h6>
</div>
</div>
</div>
<span *ngIf="errorMessage$">{{errorMessage$ | async}}</span>
</div>
```

Listing 6.39: product-list.component.html

asyc pipe is used for both products*anderrorMessage* as it unwraps a value from an asynchronous primitive

11. Configure the effects in *app.module.ts* file:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { ProductListComponent } from './product-list/product-list.component';
import { HttpClientModule } from '@angular/common/http';
import { StoreModule, ActionReducer, MetaReducer } from '@ngrx/store';
import { productReducer } from './Store/reducers/products.reducer';
import { EffectsModule } from '@ngrx/effects';
import { ProductEffects } from './Store/effects/products.effects';
export function debug(reducer: ActionReducer<any>): ActionReducer<any> {
  return function (state, action) {
    console.log('previous state', state);
    console.log('action', action);
    let nextState = reducer(state, action);
    console.log('current state', nextState);
    return nextState;
  };
}
export const metaReducers: MetaReducer<any>[] = [debug];
@NgModule({
  declarations: [
  AppComponent,
  ProductListComponent
  ],
  imports: [
  BrowserModule,
  AppRoutingModule,
  HttpClientModule,
  StoreModule.forRoot({ products: productReducer }, { metaReducers }),
  EffectsModule.forRoot([ProductEffects])
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Listing 6.40: app.module.ts

`EffectsModule.forRoot()` takes an array of effects to be used in the angular application to make API calls

12. Define the routes in *app-routing.module.ts* and load the product-list component by adding router-outlet in *app.component.html*:

```
const routes: Routes = [
{ path: '', redirectTo: 'products', pathMatch: 'full' },
{ path: 'products', component: ProductListComponent }
];
```

Listing 6.41: app-routing.module.ts

13. To access the server and fetch the data from the server, follow the below steps: [This is just for demonstration purpose, you can use any other server such as express web service as well, in that case below steps are not required to be followed]

    - Install json-server globally and locally:

```
npm install json-server -g
npm install json-server --save
```

- Create a *db.json* file in project root folder and and add the following data:

```
{
  "products": [
  {
    "id": 1001,
    "productName":"iPhone 7s",
    "productPrice":39999,
    "productImg":"iphone.jpg",
    "productBrand":"Apple"
  },
  {
    "id": 1002,
    "productName":"Nokia 8",
    "productPrice":24499,
    "productImg":"nokiasmartphone.png",
    "productBrand":"Nokia"
  },
  {
    "id": 1003,
    "productName":"Redmi Note 8Pro",
    "productPrice":39999,
    "productImg":"redminote8.PNG",
    "productBrand":"Xiaomi"
  },
  {
    "id": 1004,
    "productName":"Samsung Galaxy s10",
    "productPrice":39999,
    "productImg":"samsunggalaxy.jpg",
    "productBrand":"Samsung"
  }
  ]
}
```

Listing 6.42: db.json

- Run the json-server

```
json-server --watch db.json --port 4000
```

Chapter 7

Single Sign On using Angular

7.1

Single Sign-On

When you log in to G-mail, you automatically gain access to you-tube, Google drive, Google photos, and other Google applications. Have you ever wondered how this is happening? This is done by Single Sign On.

What is Single Sign-On(SSO)?

With Single Sign-On, the user can log in to multiple services with a one-time login. SSO will get the login credentials from directory server as Authentication and provides access to multiple applications by passing the authentication token to configured applications. Thus we can avoid the problem of maintaining the username and password for each site separately in their servers. This will also help the users to get access to the different application, without maintaining separate username and password for each application. The typical logic flow of an web application/ service with Single Sign-On will be as follows

SSO using Node-SSPI

Why we need Single Sign-On(SSO)?

- Using SSO we can eliminate the time spent on entering the user name and password for each site
- No need to remember the password for different applications
- As the SSO will get the credentials from the domain server, we can reduce the problem of storing the password in separate servers, invalid credentials, password reset issues.
- Minimize the fraudulent attempt to obtain the user name, password and some other sensitive information.

How Single Sign-On works?

Let us first understand how does an application work without SSO. Normally when you want to access some web app/service, the authentication process will be as follows.

Without SSO:

- When a user enters a website, first it will check whether the user is already logged in
- If the user is logged in, it will take him/her to the home page else it will redirect to the login page.
- User has to enter his/her username and password to the website. After authentication, the user will be redirected to the login or home page

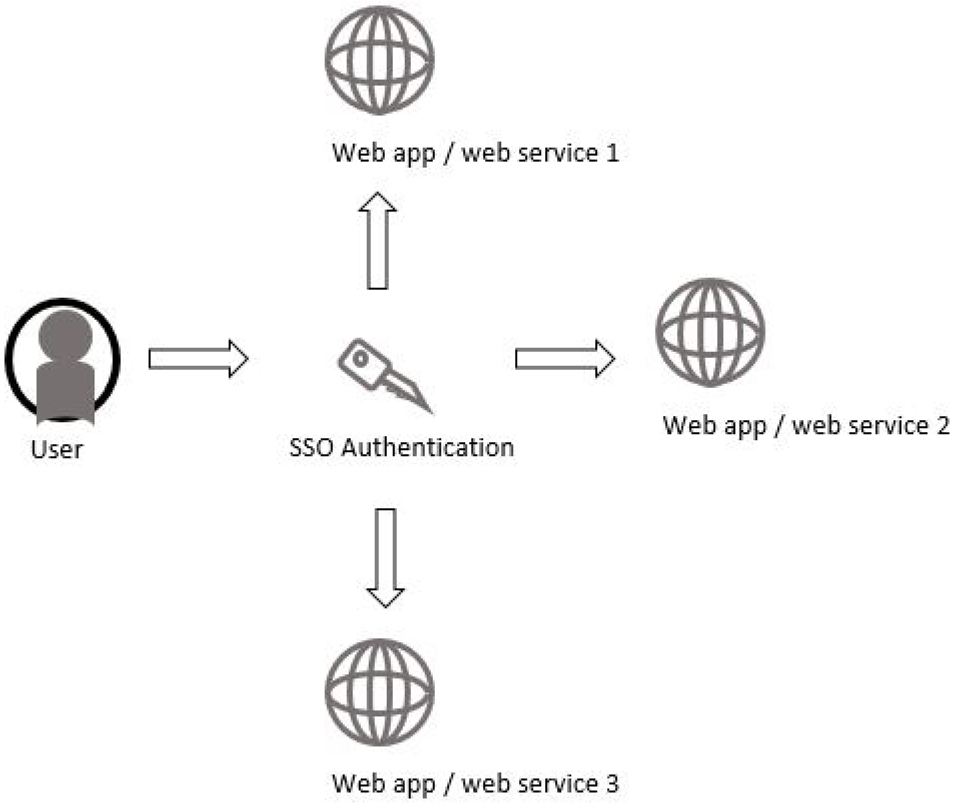


Figure 7.1: SSO Example

If we do the same with the single sign-on, the steps will be like :

With SSO:

- When a user enters a website, first it will check whether you are already authenticated by the SSO
- If yes, then it will take the user to the website's home page else it will redirect him to the login page to enter username and password

Now the user can access all the sites associated with the company without entering the username and password again. Single Sign-On can be implemented using various platforms. One such Platform is using Node-SSPI

Node-SSPI

For the server side, we are using node-SSPI for authentication.

- It stands for the Security Support Provider Interface.
- NodeSSPI is modeled to perform Windows authentication through native Windows SSPI
- After successful authentication, NodeSSPI can optionally retrieve a list of groups the user belongs to, facilitating to perform authorization.
- Node.js is customizable through JavaScript which offers much more flexibility.

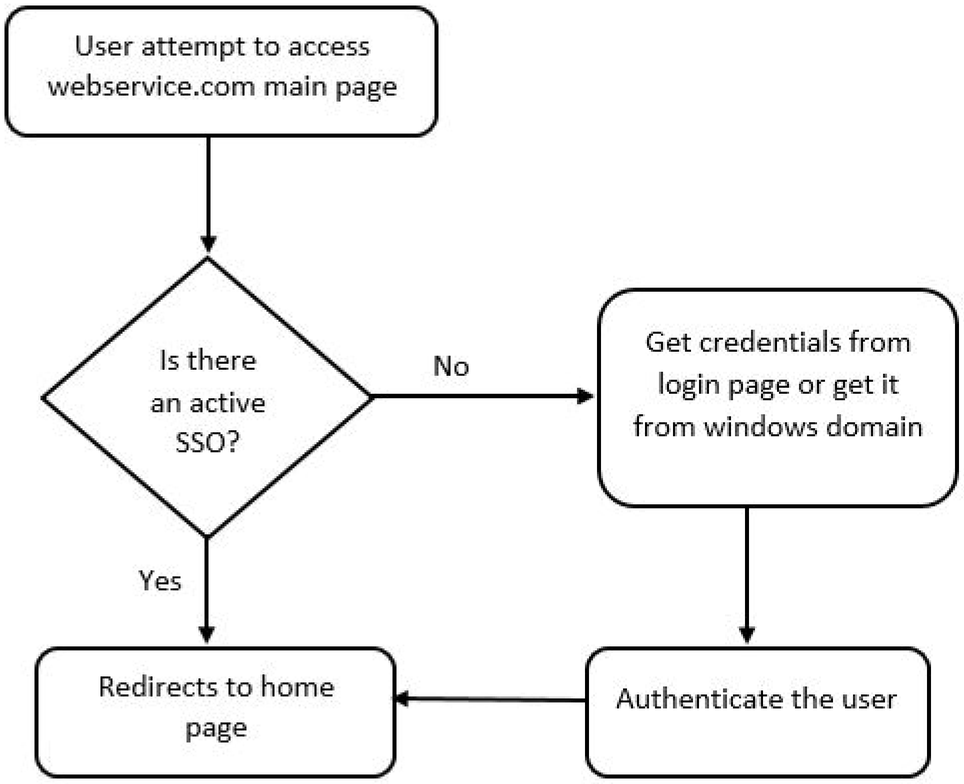

User attempt to access webservice.com main page
Is there an active SSO?
No
Get credentials from login page or get it from windows domain
Yes
Redirects to home page
Authenticate the user

Figure 7.2: Logic flow of an web application/service with Single Sign-On

7.2 Demo to fetch the username - Node

Create an Angular application that renders the username of the logged-in user.

- Create a new Express application and run the below command to install sspi package.

```
npm install node-expose-sspi
```

- Copy the below code in your *app.js* file

```
const express = require('express');
const { sso } = require('node-expose-sspi');
const cors = require('cors')
const bodyParser= require('body-parser')
const app = express();
const corsOptions = {
    origin: ['http://localhost:4200'],
    optionsSuccessStatus: 200,
    credentials: true
}
app.use(bodyParser.urlencoded({ extended: true }))
app.use("/", cors(corsOptions))
app.use(sso.auth());
app.use((req, res, next) => {
    res.json({
        sso: req.sso,
    });
```

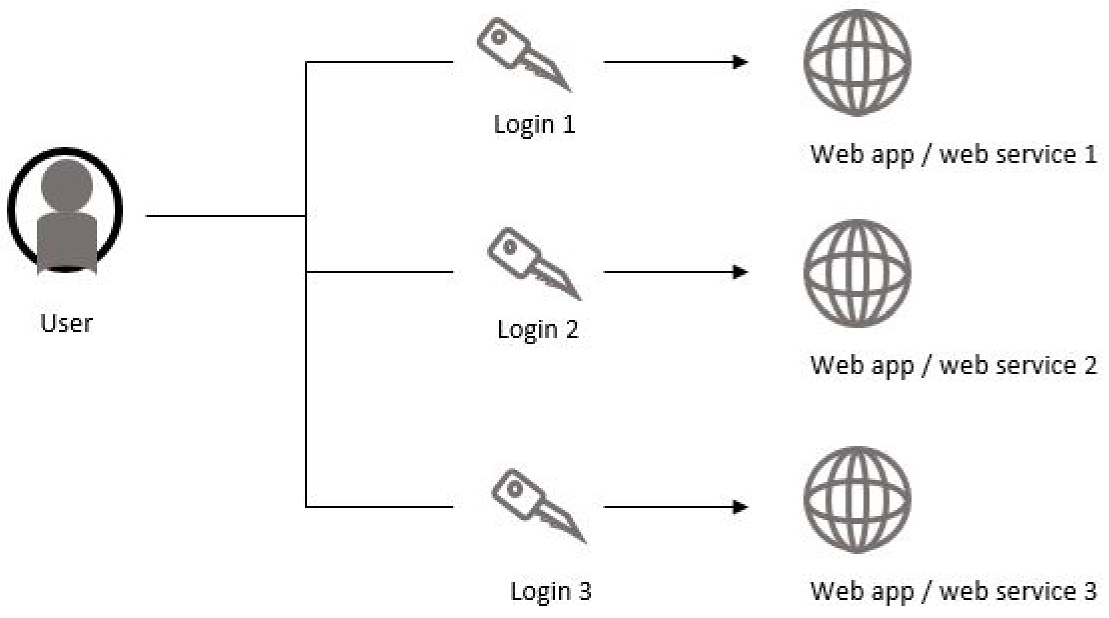


Figure 7.3: Without SSO

```
});
app.listen(3000, () => console.log('Server started on port 3000'));
```

Listing 7.1: app.js

- Run the command to start the Webservice

```
 node app.js
```

Once this is done, you can hit the URL `http://localhost:3000/` in browser, this will display all the available data related to the client. We have now created the backend application, next let us understand the series of step required for using or presenting this data in the front end angular application.

Demo to fetch the username - Angular

Follow the below steps to create your front-end application. Step4: Create your angular application.

```
ng new sso-app-ui
```

```
<div class="container-fluid"
   *ngIf="successmessage && employeeId && location && name && emailId && department && mobileNo; else nodata"
       ↪ >
   <div class="row mb-3 mt-2">
      <div class="col-md-6 offset-md-3">
         <div class="card shadow">
            <div class="card-header">
               <h2 class="text text-center">Hi {{successmessage}}!!</h2>
            </div>
            <div class="card-body">
               <table class="table table-hover">
```

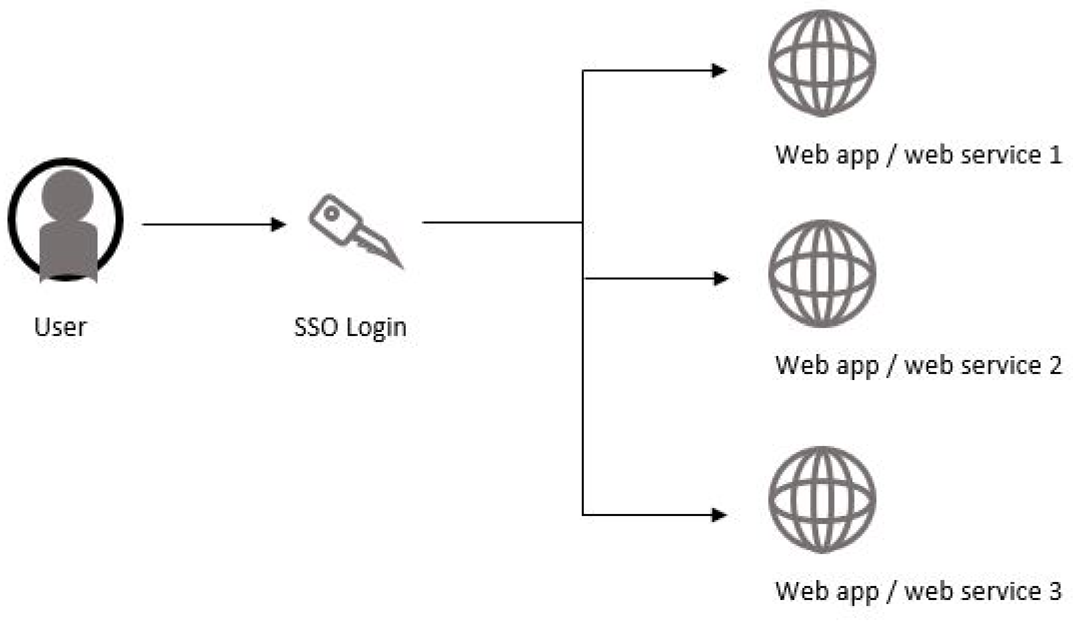


Figure 7.4: With SSO

```
                    <tbody>
                        <tr>
                            <th>Employee Id</th>
                            <td>{{employeeId}}</td>
                        </tr>
                        <tr>
                            <th>Name</th>
                            <td>{{name}}</td>
                        </tr>
                        <tr>
                            <th>Email Id</th>
                            <td>{{emailId}}</td>
                        </tr>
                        <tr>
                            <th>Mobile Number</th>
                            <td>{{mobileNo}}</td>
                        </tr>
                        <tr>
                            <th>Location</th>
                            <td>{{location}}</td>
                        </tr>
                        <tr>
                            <th>Department</th>
                            <td>{{department}}</td>
                        </tr>
                    </tbody>
                </table>
            </div>
        </div>
    </div>
  </div>
</div>
<ng-template #nodata>
    <div class="text text-center mt-5">
        <h2>Please wait while we prepare your profile</h2>
        <div class="spinner-border"></div>
    </div>
</ng-template>
```

Listing 7.2: app.component.html

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
const httpOptions = {
  headers: new HttpHeaders({ 'Content-Type': 'application/json' })
};
@Injectable({
  providedIn: 'root'
})
export class AppService {
  constructor(private http: HttpClient) { }
  getData() {
    return this.http.get('http://localhost:3000/', { withCredentials: true })
  }
}
```

Listing 7.3: app.service.ts

```
import { Component } from '@angular/core';
import { AppService } from './app.service';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'sso-demo-ui';
  successmessage: any;
  errorMessage: any;
  name: any;
  emailId: any;
  department: any;
  employeeId:any;
  location: any;
  mobileNo: any;
  constructor(private appservice: AppService) {
  }
  ngOnInit() {
    this.appservice.getData().subscribe(
      (success) => {
        console.log(success, typeof success);
        let data : any = success;
        this.successmessage= data.sso.user.name;
        this.name = data['sso']['user']['displayName'];
        this.emailId=data['sso']['user']['domain'];
         this.department=data['sso']['user']['adUser']['department'];
        this.employeeId=data['sso']['user']['adUser']['company'];
        this.location=data['sso']['user']['adUser']['l'];
        this.mobileNo= data['sso']['user']['adUser']['mobile'];

        console.log(this.successmessage, typeof this.successmessage);
      },
      (err) => {
        console.log(err);
        this.errorMessage = err.error.message;
      }
    )
  }
}
```

Listing 7.4: app.component.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { HttpClientModule } from '@angular/common/http';
import { AppComponent } from './app.component';
@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
    BrowserModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Listing 7.5: app.module.ts

```
ng serve --open
```

From this demo, we have learnt how to fetch the various details of a loggin in user. Similarly, other frontend applications can get the user logged in without asking for credentials by using the node-expose-sspi.

7.3 Need for OAuth

In most websites, we find an option to login using social networking accounts like Facebook, Twitter, Google, LinkedIn, etc. This makes login easy as we dont have to create multiple accounts in multiple web pages and facing the problem of remembering all the usernames and passwords. So how does this work? Consider the below illustration:

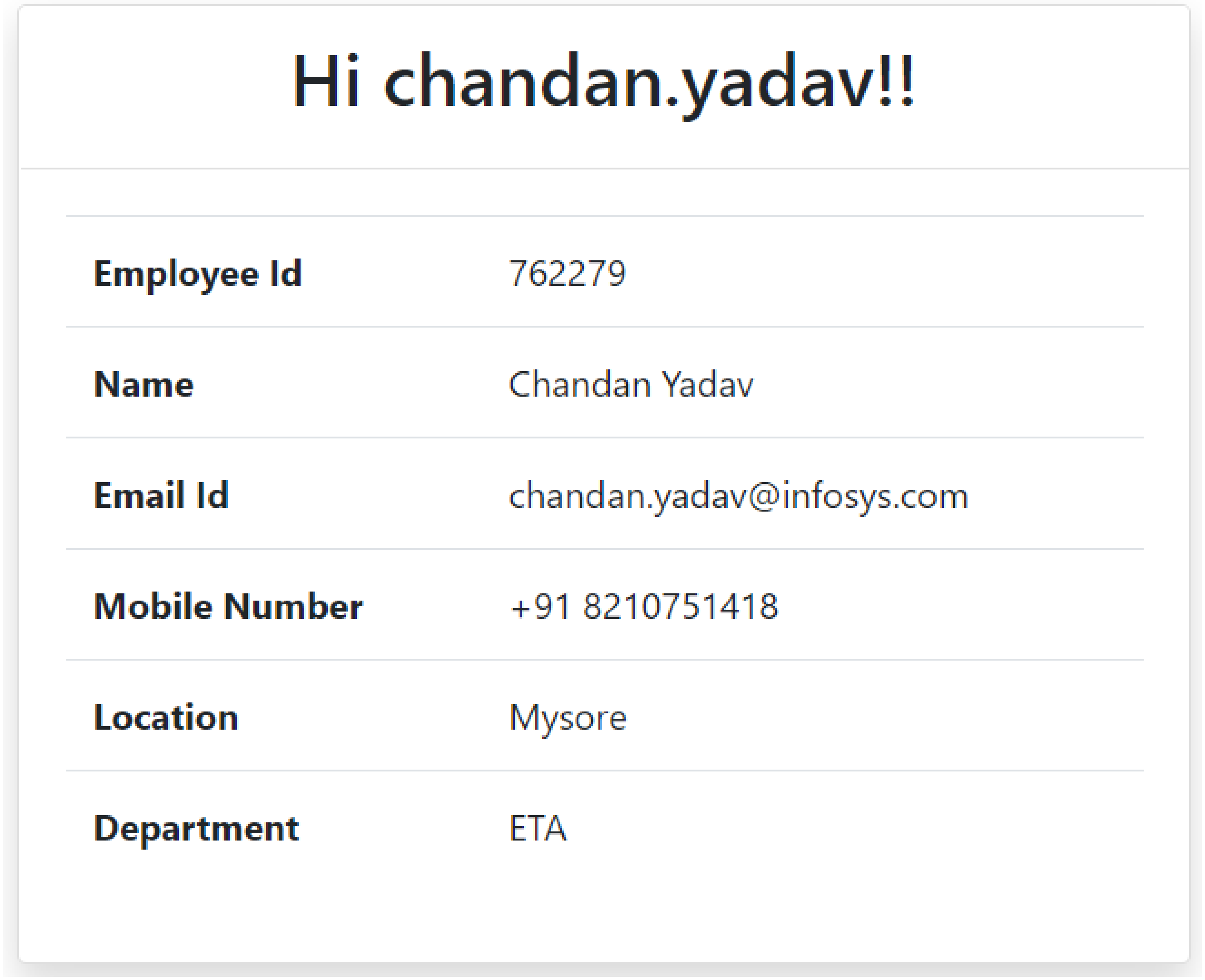
- The User clicks on a button in our Angular App.
- The Angular app will now redirect the user to the Google sign in page.
- The user provides his credentials in the google sign-in page (not in our angular app).
- By signing in, he gives permissions to our Angular app to access the users public profile data.
- When the user has thus authorized Google to allow our app to access his details.

OAuth (Open Authorization) is a standard which can allow one app to authorize another app to use its details. It is basically an Authorization standard and not an authentication standard. In our example, we use Google to both authenticate and authorize the user. There are several steps in involved in creating OAuth capabilities in our Angular app. Let us look at them in the coming pages.

7.3.1 Steps in OAuth

The various steps in enabling our app with OAuth capabilities are:

1. Create a Google account
2. Register our App in Google and generate Client token.
3. Install an OAuth library in Angular app
4. Use the client token in the Angular app
5. Invoke the libraries API to contact Google and handle the response.


Hi chandan.yadav!!

Employee Id	762279
Name	Chandan Yadav
Email Id	chandan.yadav@infosys.com
Mobile Number	+91 8210751418
Location	Mysore
Department	ETA

Figure 7.5: Demo Image

Assuming you already know how to create a Google account, let us take a look at how to register our app in Google Cloud Platform. Once we have registered an app in Google, we have to use the generated tokens in the angular app. For this discussion, we will be using the *angularx-social-login* node module. The steps are:

- `npm install angularx-social-login --save`
- Modify specific parts of the *app.module.ts* as shown below:

```
    import { SocialLoginModule } from 'angularx-social-login';
import { AuthServiceConfig, GoogleLoginProvider } from 'angularx-social-login';
const config = new AuthServiceConfig([
  {
    id: GoogleLoginProvider.PROVIDER_ID,
    provider: new GoogleLoginProvider('<Your client ID generated by Google>')
  },
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    SocialLoginModule
```

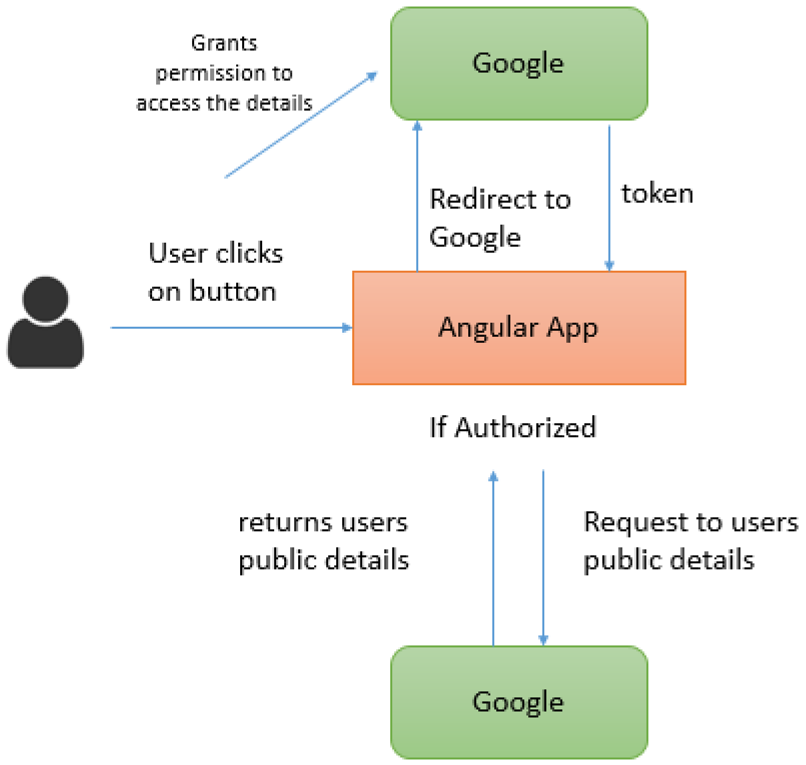

Grants permission to access the details
Google
Redirect to Google
token
User clicks on button
Angular App
If Authorized
returns users public details
Request to users public details
Google

Figure 7.6: OAuth Example Flow

```
  ],
  providers: [
    {
      provide: AuthServiceConfig,
      useFactory: provideConfig
    }
  ]
```

Listing 7.6: app.module.ts

- Create a new component to display the login button and user details as shown below:

```
    <div *ngIf="user" class="card text-center">
    <h6 class="card-header">
       Logged In with Google
    </h6>
    <div class="card-block"></div>
    <img class="card-img-top img-responsive photo" src="{{ user.photoUrl }}">
    <div class="card-block">
      <h4 class="card-title">{{ user.name }}</h4>
      <p class="card-text">{{ user.email }}</p>
    </div>
    <div class="card-block">
      <button class="btn btn-danger" (click)="signOut()">Sign out</button>
    </div>
  </div>
```

Listing 7.7: HTML

```
   user: SocialUser;
 constructor(private authService: AuthService) { }
 ngOnInit() {
   this.authService.authState.subscribe((user) => {
     this.user = user;
     console.log(user);
   });
 }
 signInWithGoogle(): void {
   this.authService.signIn(GoogleLoginProvider.PROVIDER_ID).then(x => console.log(x));
Click here to download zip folder :Steps in Angular App-Oauth.zip
```

Listing 7.8: TS

Chapter 8

Angular Security

Angular provides built-in security protection for common web application vulnerabilities like XSS, CSRF, etc. Let us see how Angular handles them.

Cross-site scripting (XSS):

The cross-site scripting attack happens when malicious code is injected into the webpage. This malicious code can steal or modify the user's data and impersonate the user. In the OWASP Top 10 2021, Cross-Site-Scripting is also included under the Injection category and is one of the most common attacks on web applications.

To prevent XSS attacks, malicious code should be sanitized before entering the DOM. Sanitization is all about converting untrusted values to a safe format that can be inserted into DOM. Angular handles this by treating all values as untrusted by default. Angular escapes the untrusted values whenever malicious data is entered into DOM using interpolation or binding.

DomSanitizer class of Angular helps to sanitize the values and helps to prevent XSS attacks.

Server-side XSS:

XSS attacks can also happen to server-side code, which enables attackers to exploit the complete application. HTML rendered from the server is vulnerable to injection attacks which give attackers complete control of your Angular application. To mitigate this, use a templating language that automatically escapes the untrusted values and helps to block the attacks. Angular templates generated by templating language on the server side create template injection vulnerabilities and pose a higher risk.

CSRF and XSSI:

Angular has built-in support, which helps in preventing vulnerabilities like cross-site request forgery (CSRF or XSRF) and cross-site script inclusion (XSSI). Although both are mitigated primarily on the server side, Angular provides the HttpClient class, making the client-side integration easier and simpler. You can add another layer of protection to the Angular application by setting Security Policy (CSP) header and AOT template compiler.

Content Security Policy

It is an HTTP response header that helps mitigate numerous content injection attacks. It provides different directives that can be used to restrict the content loaded into a web application. It provides directives like style-src, script-src, etc. The minimal policy required for the Angular project is: default-src 'self'; style-src 'self' 'unsafe-inline'; Angular itself requires these settings to function correctly.

- default-src is a fallback for other resource types when they are not configured
- style-src directive restricts loading styles from invalid resources

- script-src directive restricts loading scripts from invalid resources.

Consider the below policy: Content-Security-Policy: default-src 'self' Setting the value of the "default-src" directive to "self" indicates that only the sources from the original domain are allowed. Content-Security-Policy: style-src 'self' 'unsafe-inline' Setting the value of the "style-src" directive to "self" indicates that only the styles that come from the original domain are allowed. Disallowing inline styles and inline scripts is one of the advantages CSP provides. However, if you need to use it, you can use 'unsafe-inline.' As your project grows, however, you may need to expand your CSP settings beyond this minimum to accommodate additional features specific to your application.

Using AOT template compiler

The AOT template compiler prevents template injection and dramatically improves application performance. It is the default compiler used by Angular CLI applications, and in all production deployments, it is recommended.

Angular XSRF

The below things need to be performed for enabling CSRF protection in an Angular application:

1. Including a custom HTTP header which has a definite name and value. This can be coupled with the session id of the user token. This is called "double submit cookie method". This implementation however should be used with caution as it is known for having vulnerabilities when all the sub-domains of the origin are not controlled.

2. The server will check if the header is present and also checks if the header value present in the request matches with what was sent on the client-side. If not matching, it indicates a malicious request.

3. All browsers can implement same origin policy. This way it possible to ensure that only code which initiates from a website from where cookies have been set, is able to read the cookie named xsrf token.

4. Care can be taken to store the API key in a server-side environment variable and access it from the backend so that XSRF attacks can be prevented.

For setting the XSRF token cookie or header in the Angular application, the below can be used:

```
imports: [
 HttpClientModule,
 HttpClientXsrfModule.withOptions({
 cookieName: 'app-Xsrf-Cookie',
 headerName: 'app-Xsrf-Header',
 }),
],
```

Listing 8.1: app.module.ts

Chapter 9

Internationalization in Angular

9.1 Introduction to Internationalization (xi18n)

Consumption of modern web and mobile experiences is worldwide and access to the web has been fundamental to all the users and goals of the W3C since the beginning. Therefore, making an application available in multiple languages and user-friendly to a worldwide audience is important. And this can be achieved by Angular's Internalization(i18n) tools, which will help in making applications available in multiple languages.

Internationalization (i18n):

Internationalization is the process of making an application to be usable in various languages. For internationalization, Angular pipes can be used. Template translation using i18n has 4 phases:

- Mark the text content for translation in the template
- Create a translation file i.e. *messages.xlf* using the command `xi18n`, which extracts the marked text content into an industry-standard translation source file
- Edit the translation file by translating the extracted text content into the required language
- Merge the translation file into the application by editing the configuration in *angular.json*

Let us understand the steps to be followed for language translation:

1. Mark the text content for translation in the template using Angular i18n attribute
   (a) Consider the *app.component.html* with the below-given content in it, which we will translate to a different language

```
        <h1 i18n>Hello, Welcome to angular application</h1>
```

Listing 9.1: app.component.html

   (b) To translate the text content, a translator may need additional information such as the meaning of the text content and description about it as shown below:

```
        <h1 i18n="Welcome | To greet the user">Hello, Welcome to angular application</h1>
```

Listing 9.2: app.component.html

   Meaning and description can be separated by — pipe

   (c) Provide the custom id to the text content by adding the prefix @@ as shown below:

```
        <h1 i18n="Welcome| To greet the user @@welcome">Hello, Welcome to angular application</h1>
```

Listing 9.3: app.component.html

2. Create a translation source file

   (a) Text content which has to be translated will be extracted to translation source file by running the following command in the project folder:

```
    D:\>my-app> ng xi18n
```

   (b) This command will generate the file named *messages.xlf* in the project folder with the below content:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
  <file source-language="en" datatype="plaintext" original="ng2.template">
    <body>
      <trans-unit id="welcome" datatype="html">
        <source>Hello, Welcome to angular application </source>
        <context-group purpose="location">
          <context context-type="sourcefile">src/app/app.component.html</context>
          <context context-type="linenumber">1</context>
        </context-group>
        <note priority="1" from="description"> To greet the user </note>
        <note priority="1" from="meaning">Wlecome</note>
      </trans-unit>
    </body>
  </file>
</xliff>
```

Listing 9.4: messages.xlf

3. Translate the extracted text content to the required language

   (a) In the src folder of the project folder, create a folder as locale and move the messages.xlf file into this folder

   (b) Create a copy of the *messages.xlf* file for the required language. For example, if it is for French-language rename it as *messages.fr.xlf*

   (c) *messages.fr.xlf* file will have ¡trans-unit¿ section as shown below, which represents the text which has to be translated

```
  <trans-unit id="welcome" datatype="html">
        <source>Hello, Welcome to angular application </source>
        <context-group purpose="location">
          <context context-type="sourcefile">src/app/app.component.html</context>
          <context context-type="linenumber">1</context>
        </context-group>
        <note priority="1" from="description"> To greet the user </note>
        <note priority="1" from="meaning">Wlecome</note>
</trans-unit>
```

   (d) Create a copy of ¡source¿ tag, rename it as ¡target¿ which is given in line no. 3 and replace its text content to the French language as shown below:

```
1    <trans-unit id="welcome" datatype="html">
2        <source>Hello, Welcome to angular application </source>
3        <target>Bonjour, Bienvenue dans l'application angulaire</target>
4        <context-group purpose="location">
5          <context context-type="sourcefile">src/app/app.component.html</context>
```

```
          <context context-type="linenumber">1</context>
        </context-group>
        <note priority="1" from="description"> To greet the user </note>
        <note priority="1" from="meaning">Wlecome</note>
</trans-unit>
```

4. Merge the translation file into the application We need to compile the app with the translation file to merge the translated text into the component template. In order to do this, the below information should be provided to the Angular compiler.

   - Translation file
   - Translation file format
   - locale of the file

   Angular provides 2 ways to compile the application i.e. JIT and AOT. Here, we will use AOT as this is the recommended type of compilation for production since it provides great performance and loading time.

   - To instruct the AOT compiler on how to use the translation configuration, add the below build configurations options in angular.json file:

```
1        "build": {
2  ...
3  "configurations": {
4        ...
5         "fr": {
6                "aot": true,
7                "outputPath": "dist/my-app/",
8                "i18nFile": "src/locale/messages.fr.xlf",
9                "i18nFormat": "xlf",
10               "i18nLocale": "fr"
11              }
12       }
13 }
```

Listing 9.5: angular.json

   Line no. 7: ”outputPath”: ”dist/my-app/” : output folder path
   Line no. 8: ”i18nFile”: ”src/locale/messages.fr.xlf”: Translation file path
   Line no. 9: ”i18nFormat”: ”xlf” : Translation file format
   Line no. 10: ”i18nLocale”: ”fr”: locale id

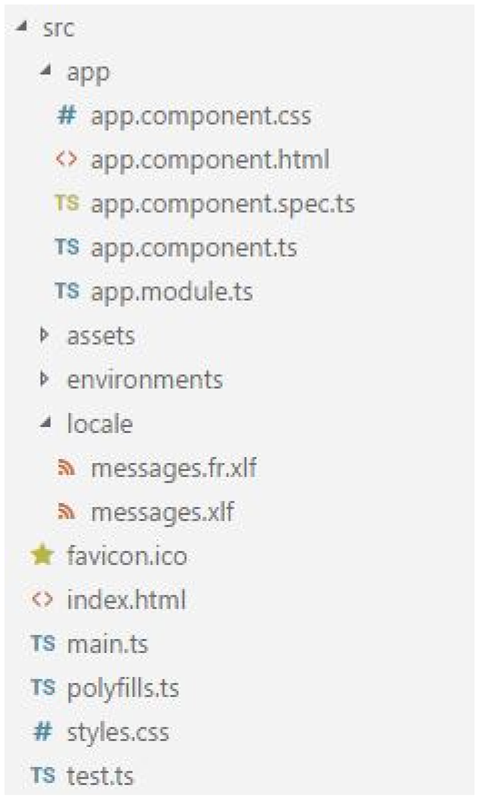
   - Update the configurations inside of serve for fr as shown below:
   - To run the application in one of the configured languages, run the below command:

```
      D:>my-app> ng serve --configuration=fr
```

Note: update the package.json’s script with below lines to run the application

```
{
"start:fr": "ng serve --configuration=fr",
"build:fr": "ng build --configuration=fr",
"int:extract": "ng xi18n --output-path src/locale"
}
```

Listing 9.6: package.json



```
src
  app
    app.component.css
    app.component.html
    app.component.spec.ts
    app.component.ts
    app.module.ts
  assets
  environments
  locale
    messages.fr.xlf
    messages.xlf
  favicon.ico
  index.html
  main.ts
  polyfills.ts
  styles.css
  test.ts
```

Figure 9.1: project structure

Chapter 10

Debugging

10.1 Debugging Extensions

10.1.1 Extensions

- Augury Extension
- Angular Devtools

10.1.2 Debugging Angular Application in Visual Studio Code IDE

1. In the VSC IDE, click on the left side of the editor and go to the Debug View menu. The launch.json file will store the debugging configuration. When debugging is being attempted fom here, for the first time, launch.json is not configured yet. VSC IDE will ask to create one. Click on Run and Debug option.
2. Once Run and Debug is clicked, Select Environment dialog box will open. Type "Chrome" there and select.
3. VSC IDE creates the *launch.json* file with necessary configurations which can help in launching the application.
4. Update port number to 4200 as the application would be running at 4200.
5. If the *launch.json* is already available in the VSC IDE, then choose the Add configuration option present within the Run and Debug dropdown.
6. Run the application using `ng serve --open` command
7. Click of the left side of line number and add a breakpoint. Add as many breakpoints as needed. Once done, click on the green arrow to start debugging the application. The application launch happens in Google Chrome and the application's execution will stop at the breakpoint number 1.

Note: Its key to execute and launch the application before the debugger is launched. This is because the debugger has to connect to a running instance of the application so that it can interact with it.

Chapter 11

Zone pollution

11.1 Intorduction

Angular uses Zone.js as its signaling mechanism in order to detect when the application state changes. Zone.js can capture asynchronous operations like setTimeout, network requests, etc and Angular performs change detection based on signals from Zone.js. Consider the below tasks or micro tasks:

- setTimeOut, setInterval
- Task or micro task scheduled by third party library

These do not make any changes to data. Running change detection becomes unnecessary in such contexts. Hence in order to improve performance, its critical to recognise conditions when change detection is essentially not needed and then execute the code outside the Angular zone which can in turn avoid triggering redundant change detection calls.

Note: A zone in Angular is necessaily an execution context.

Zone pollution occurs when multiple zones are in use within the same Angular application. These can happen because of:

- Third party libraries: Some third party libraries may create their own zones which may conflict with Angular zone.
- Manual Zone creation: Developers may unknowingly create new zones if they are unware about zone handling by Angular.
- Integration with Non-Angular code: When integrating with non-Angular code, its important for developers to stay knowledgeful about how zones are managed to prevent pollution.

```
import { Component, NgZone } from '@angular/core';
@Component({
  selector: 'app-zone-pollution',
  templateUrl: './zone-pollution.component.html',
  styleUrls: ['./zone-pollution.component.css']
})
export class ZonePollutionComponent {
  constructor(private ngZone: NgZone){}
  handleClick(){
    this.ngZone.run(()=>{
      // code here will run in Angular zone
      console.log('Button has been clicked');

    })
    setTimeout(()=>{
      // code here will in a different zone
      console.log('setTimeout has been called');
    },1000)
  }
}
```

Listing 11.1: zone-pollution.component.ts

In the above example,

- Inside handleClick(), ngZone.run() has been used to run code explicitly in Angular zone. This ensure Angular is aware of changes and change detection occurs.
- In the setTimeOut, however, code executes in a different separate zone and not Angular zone.

This can lead to zone pollution. The outcome of the asynchronous tasks needs not be reflected in the Angular application state, they may not be detected or synchronized properly due to multiple zones. This is because of JavaScript's event loop nature and this makes the code inside setTimeOut executes in a different zone.

Zone pollution can lead to unexpected behavior and make it quite difficult to track and debug issues. As an example, an asynchronous task initiated in different zone might not trigger Angular's change detection which might result in UI inconsistencies. AngularDevTools can be used to detect unnecessary change detection. Angular provides NgZone to explicitly run code in a specific zone. This comes in handy while integrating non-Angular code, particularly while using third-party libraries which are not designed keeping Zone.js in mind. Consider the below code:

```
import { Component, NgZone, OnInit } from '@angular/core';
@Component(...)
class AppComponent implements OnInit {
  constructor(private ngZone: NgZone) {}
  ngOnInit() {
    this.ngZone.runOutsideAngular(() => setInterval(pollForUpdates), 500);
  }
}
```

Listing 11.2: app.component.ts

Here, pollForUpdates is called in a zone outside Angular. Here, post the execution of the initialization logic, change detection will not get invoked inside the Angular zone, but will be invoked outside the Angular zone and hence change detection will get skipped.

Chapter 12

Slow Computations

When change detection cycle runs, Angular will synchronously,

- Assess all the template expressions for all the components, unless stated explicitly. This is performed on the basis of the change detection strategy that has been configured for that component
- Assess the life cycle hooks like ngDoCheck, ngAfterViewChecked, ngAfterContentChecked and ngOnChanges. Since Angular runs computations in a sequential manner, with just a single slow computation within a template or a lifecycle hook, it can completely slow down the complete change detection process.

Angular DevTool's profiler can be used to indetify heavy computations, so that needful actions can be taken. Slow computations can be avoided by following the below strategies:

- Optimization of the underlying algorithm: Effort can be put in to speed up the algorithm that seems to be causing slowness. This is suggested approach.
- Using pure pipes to cache: Heavy computations can be moved a pure pipe. This can surely prove to be helpfu as Angular performs reevaluation of a pure pipe only when changes to input to the pipe gets detected, that too, by comparing it to the previous invocation by Angular.
- Usage of memoization: Memoization technique is a similar to pure pipes. The difference is that pure pipes keeps only the latest result from the computation whereas memoization can store multiple results.
- Avoiding repaints/reflows in lifecycle hooks: Reflows and repaints are generally very slow. So its advisable to avoid doing these in every single change detection cycle.

Chapter 13

Skipping Component subtrees

13.1 Understanding skipping component subtrees

Why there is need to skip component subtrees?

Change detection happens necessarily very fast for most of the applications. However, there can be scenarios when an application has a remarkably large component tree. It can so happen that a certain part of the application is actually not much affected by a state change. However, as per norms, Angular will make change detection to run across the entire application. This can result in performance issues.

What does skipping component subtrees mean?

If a certain part of the application doesn't seem to be affected by a state change, then the change detection associated with the entire component subtree can be skipped by making use of OnPush.

With OnPush change detection, Angular is instructed to perform change detection for a component subtree only when:

- A new input is received by the root component as a result of a template binding. The current and past value of the input is compored using == by Angular.
- An event is handled by Angular (for example usage of event binding, output binding, or @HostListener) in the subtree's root component or any of its children if they have used OnPush change detection or not.

Using OnPush

Change detection strategy of a component can be set to OnPush within the @Component decorator.

```
import { ChangeDetectionStrategy, Component } from '@angular/core';
@Component({
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class MyCustomComponent {}
```

Let us look at how Change Detection happens in specific scenarios.

- Scenario1: For a component tree where OnPush is not mentioned, Angular performs change detection for the entire component tree as usual. For components having OnPush configured, change detection is skipped until the component gets new inputs.

  Green ones will be checked by Angular.

- Scenario2: When Angular has to handle an event within a component for which OnPush strategy is configured, Angular will execute change detection inside the entire component tree. Component subtrees having OnPush will be ignored, until new inputs are received.

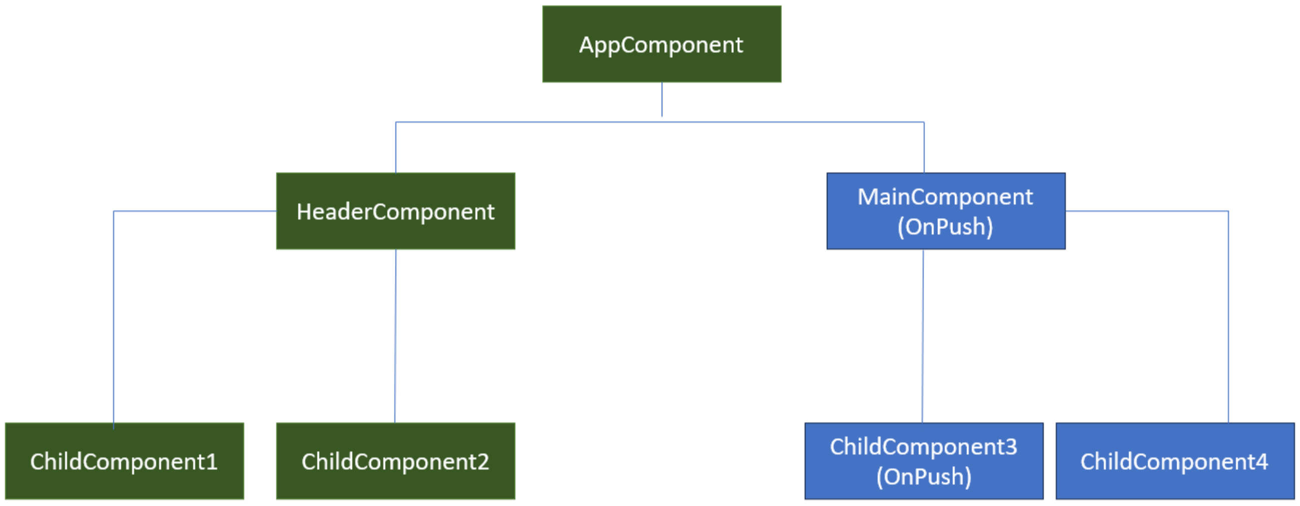


Figure 13.1: Scenario 1

- Scenario3:

  When Angular has to handle an event withing a component for which OnPush has been configured, Angular will execute change detection inside the entire component tree, including the ancestors of the component.

- Scenario4: When parent passes new input to child component having OnPush, Angular will run change detection in child component as it received new inputs. But for further grand children where OnPush is configured, Angular will not run change detection unless it receives new inputs as well.

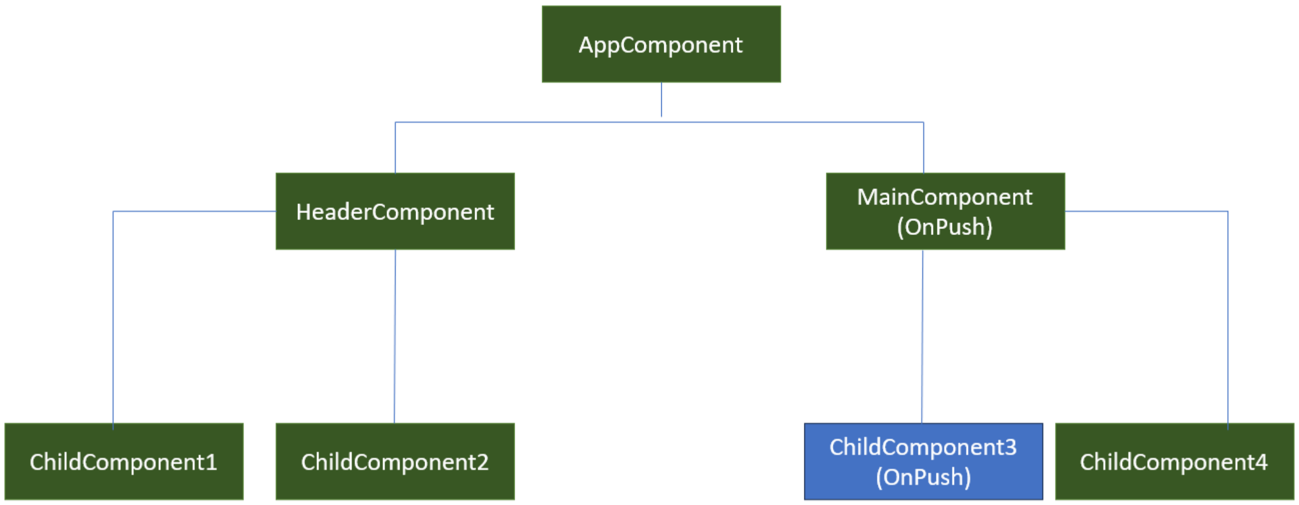


Figure 13.2: Scenario 2

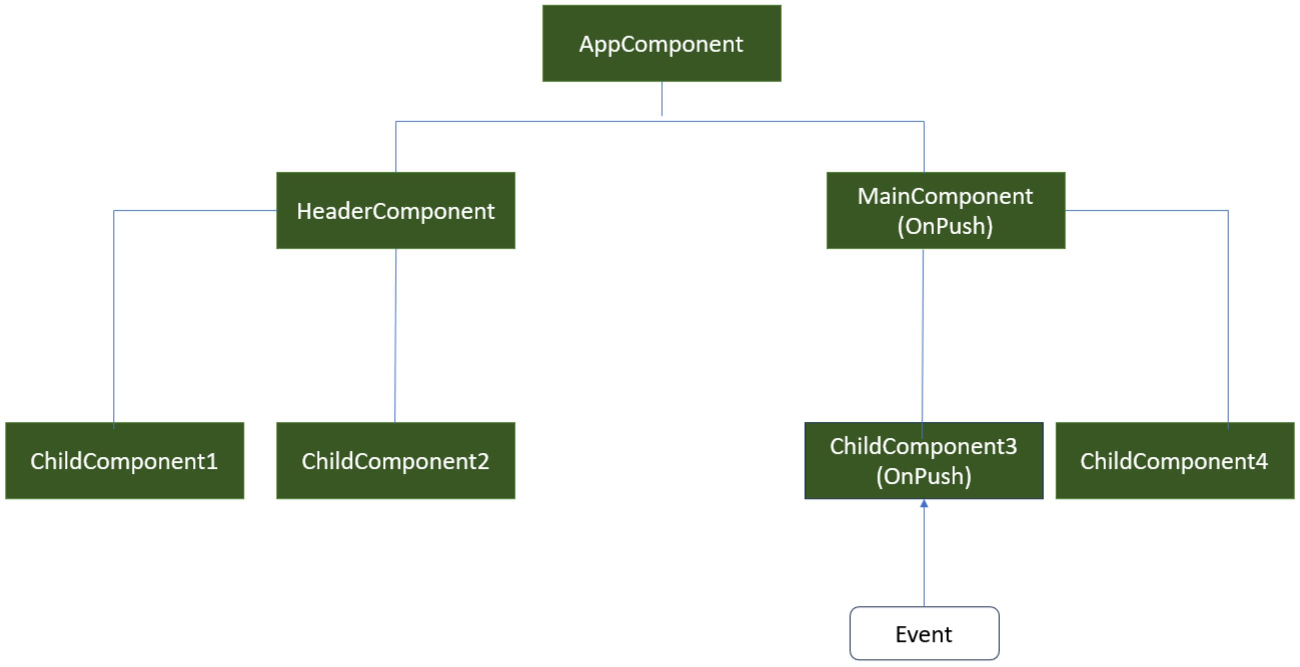


Figure 13.3: Scenario 3

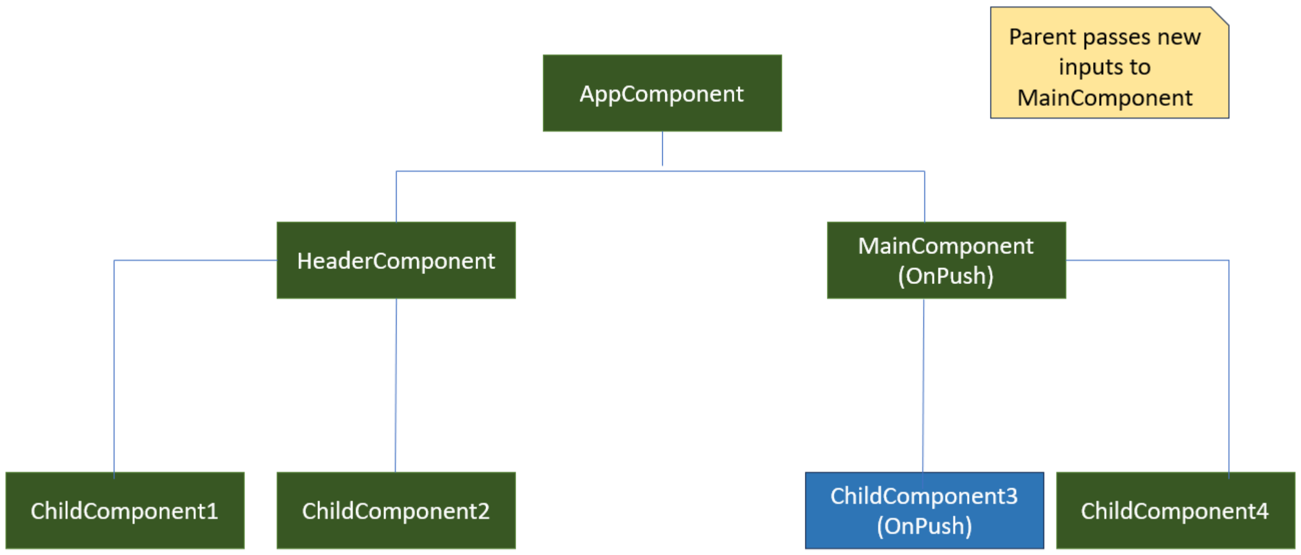


Figure 13.4: Scenario 4