Modules, Signals and Routing

Creating Single-Page Applications

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Have a Question?

Software University

sli.do

#angular

Table of Contents

Standalone Components

Building Blocks of the Application

What is Standalone in Angular?

- Components **import** other standalone components, directives, and pipes directly within their metadata (**imports array**), removing the need for a shared or parent module to connect them
- These components can function **independently** and be used directly in Angular applications
- This feature is **backward-compatible**, meaning you can still use traditional modules in combination with standalone components

For What Standalone is Used in Angular?

- It is used to simplify the structure of Angular applications
- Standalone components help optimize **dependency management** since they declare their own dependencies instead of relying on a **module**
- Standalone components are ideal for building self-contained, reusable **UI** elements that don't require extra configurations

Standalone Component Example

```
@Component({
  selector: 'app-example',
  templateUrl: './example.component.html',
  standalone: true,
  imports: [CommonModule, OtherComponent,
     SomeDirective]
})
export class ExampleComponent {}
```

Property in the component decorator when defining the component

Signals

Optimize Rendering Updates

What Are Signals

- **Signals** are a new reactivity model introduced in Angular to manage **state** and **reactivity** more predictably and efficiently
- They offer a simpler alternative to **observables** for local state management in components
- A **signal** is a wrapper around a value that notifies interested consumers when that value changes
- Signals can contain **any** value, from primitives to complex data structures

Signal Key Features

- A signal holds a value, and this value is **immutable**, meaning it can only be changed explicitly (using methods like **set**, **update**)
- Signals automatically trigger view updates when their value changes, making them reactive and removing the need for manual subscriptions or **`ChangeDetectorRef`**
- Signals provide a clear, **declarative** syntax to manage state and react to state changes

Signal Core Concepts

- Use the `signal()` function to create a signal and initialize it with a value
- Signals have methods to **modify** their value
  - `set`(value): Replaces the current value
  - `update`(fn): Updates the value based on the previous one
  - `mutate`(fn): Mutates the value if it's a mutable object (e.g., arrays, objects)

Signal Example

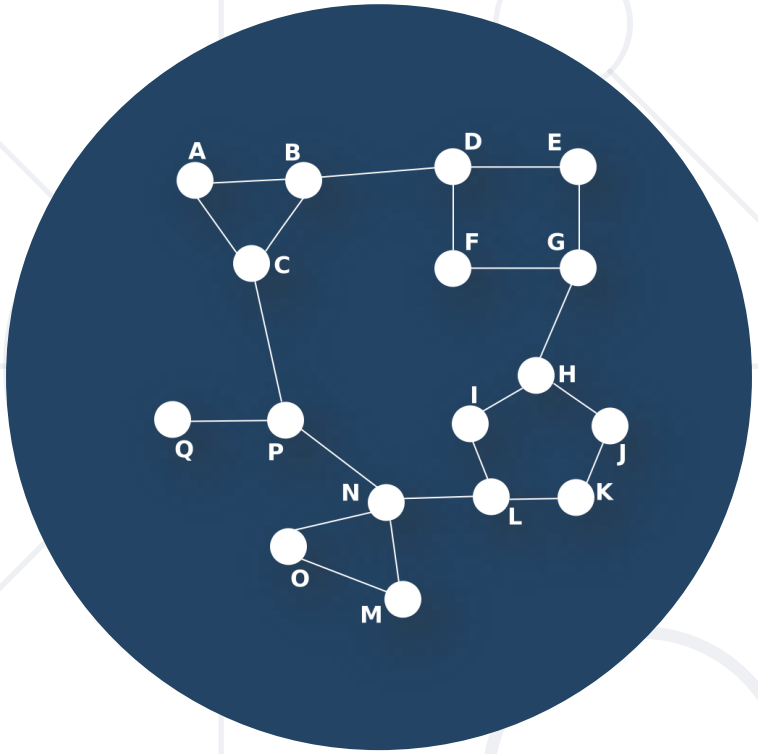
```
import { signal } from '@angular/core’;

const count: WritableSignal<number> = signal(0);
console.log('The count is: ' + count());
count.set(3);
count.update(value => value + 1); // Increment the
count by 1
```

Computed Signals

- **Computed** signal are **read-only** signals that derive their value from other signals
- Computed signals are defined using the **computed** function and specifying a derivation
- Computed signals are both **lazily** evaluated and **memorized**
- As a result, you can safely perform **computationally expensive** derivations in computed signals, such as filtering arrays

```
const count: WritableSignal<number> = signal(0);
const doubleCount: Signal<number> = computed(() => count() * 2);
```

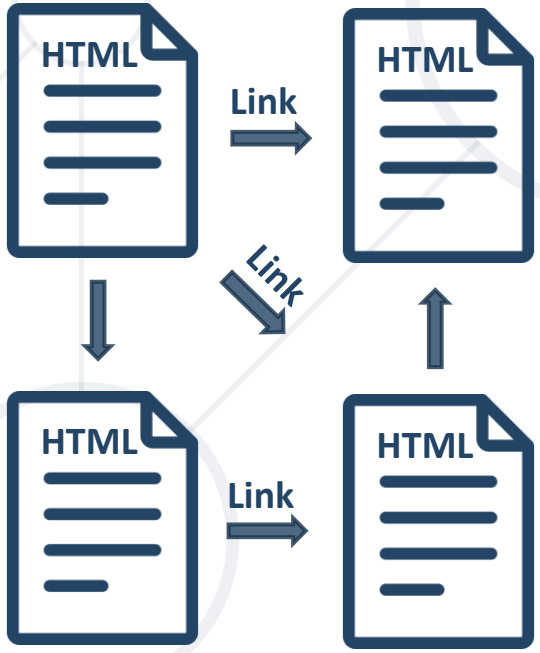


Routing Concepts

Navigation for Single Page Applications

What is Routing?

- Allows navigation, **without reloading** the page
- Pivotal element of writing **Single Page Applications**



Standard Navigation

Navigation using Routing

Single Page Applications

- A **Router** loads the appropriate content when the **location changes**
  - E.g. when the user manually enters an address
- Conversely, a change in content is **reflected** in the **address bar**
  - e.g., when the user clicks on a link
- Benefits
  - Load all scripts **only once**
  - Maintain state across multiple pages
  - Browser history can be used
  - Build User Interfaces that react quickly

Router Module

Setup, Links, Redirects, Parameters

Define the Template

- First add the **base** meta tag into the **index.html** file

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

Usually **added** by the **CLI**

- Add a **nav** tag so the **user** can navigate through the app

```
<nav>
 <a routerLink="/home">Home</a>
 <a routerLink="/about">About</a>
</nav>
```

Use **routerLink** instead of **href**

- Define the **router outlet** where the **content** will be **rendered**

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

Create Routes

- Import **Routes** and **Components**

```
import { Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { AboutComponent } from './about/about.component';
```

- Define the needed **routes** as an **array** of **objects**

```
export const routes: Routes = [
    {path: '', component: HomeComponent},
    {path: 'catalog', component: CatalogComponent},
    {path: 'about', component: AboutComponent}
];
```

'/' is omitted

Create Routes

- Add **Components** to the **imports** array

```
import { Component } from '@angular/core';
import {  RouterOutlet } from '@angular/router';
import { NavbarComponent } from "./navbar/navbar.component";
import { CatalogComponent } from './catalog/catalog.component';
import { AboutComponent } from './about/about.component';
```

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, CatalogComponent, AboutComponent,
      NavbarComponent]
})
```

The RouterLink Directive

- A basic usage of the **RouterLink** directive

```
<a routerLink="/user/profile">Profile Page</a>
```

- Bind to the directive a pass an **array** of **parameters**

```
<a
 [routerLink]="[ '/user', 1, 'profile' ]">
   Profile Page
</a>
```

Navigate Programmatically

- Inject the Angular Router in components

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

From "@angular/router"

- Use it to **navigate** from one component to another

```
loadData() {
 // Service call goes here
 this.router.navigate([ '/home' ])
}
```

Passing Parameters to Routes

- Define routes with parameters the following way

```
{ path: 'user/:id', component: UserDetailsComponent }
```

- Nested parameters

```
{
 path: 'user/:id/:username',
 component: UserProfileComponent
}
```

Fetching Parameters

- Inject **ActivatedRoute** in components

```
constructor(
 private route: ActivatedRoute
) { }
```

- Retrieve parameters directly from the snapshot

```
ngOnInit() {
  const id = this.route.snapshot.params['id']
}
```

Only runs **one time** when the component is **initiated**

Fetching Parameters Reactively

- To change the content of a component **inside the same one** use an **Observable** instead

```
ngOnInit() {
  this.route.params
    .subscribe((params: Params) => {
        const id = params['id']
      }
    )
}
```

Query Strings and Fragments

- To pass query parameters / fragments attach directives

```
<a
   [routerLink]="[ '/users', user.id, user.name ]"
   [queryParams]="{ search: 'Peter' }"
   fragment="loading"
</a>
```

- Retrieve them from the **snapshot**

```
this.route.snapshot.queryParams
this.route.snapshot.fragment
```

Setting Up Child (Nested) Routes

- Create nested routing by defining **child routes** using the **children property** of a route

```
{
  path: 'users', component: UsersComponent, children: [
    { path: ':id', component: UserComponent },
    { path: ':id/details', component: UserDetailsComponent }
  ]
}
```

- New router outlet needed at **UsersComponent**

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

Using Wildcards and Redirects

- If the requested **URL** doesn't **match** any paths for routes, **show** a **404** Not Found Page
  - This is done by using a **wildcard** '**'

```
{ path: '**', component: PageNotFoundComponent }
```

  - To redirect from one path to another

```
{ path: '', redirectTo: 'home', pathMatch: 'full' }
```

Telling the router how to match a URL to the path of the route

Router Guards

Protecting Routes

Guards Overview

- Limiting access to a route is **needed** in every application
- In Angular there are route **guards**
  - Build a guard **service**
  - Register the **service** in an Angular
  - **Add** the guard to a desired **route**

CanActivate Guard

- The **CanActivate** guard **checks** criteria before **activating** a route
- It **limits** route access to **specific** users (register users, admins..)
- Called when the url **changes**

```
import { Injectable, inject } from
"@angular/core";
import {
   Router, CanActivateFn,
   ActivatedRouteSnapshot,
   RouterStateSnapshot
} from "@angular/router";
```

Guard Example

- Create a **guard** that restricts **non-authenticated** users

```
export const AuthGuard: CanActivateFn = (route, state) => {
const authService = inject(AuthService);
const router = inject(Router);
const isLoggedIn = authService.checkIfLogged(state.url);  if
(!isLoggedIn) {
    router.navigate(['/login']); // Redirect to login if not
logged in  }
return isLoggedIn;
};
```

Angular Router Resolver

- The Angular Router provides a **resolve** property
- It takes a route resolver and allows your application to fetch data **before** navigating to the route

```
path: 'users', component: ServersComponent, children: [
  {
    path: ':id',
    component: UserDetailsComponent,
    resolve: { user: UsersResolver }
  }
]
```

Implement the Resolver

- Create the Resolver Guard

```
import { Injectable } from '@angular/core’;
import { ResolveFn } from '@angular/router’;
import { inject } from '@angular/core’;
import { UsersService } from ’./users.service’;
import { User } from ’./user.model’;

export const userResolver: ResolveFn<User> = (route) => {
const usersService = inject(UsersService);
const userId = route.paramMap.get('id')!;
return usersService.getUserById(userId);
};
```

Inject the service inside the guard

Use It Inside a Component

- Inside a Component fetch the data from the **data property** of the **snapshot**

```
export class UserComponent {
route = inject(ActivatedRoute);
user!: User;

ngOnInit() {
this.user = this.route.snapshot.data['user'];
 }
}
```

The name bound **inside** the route resolver

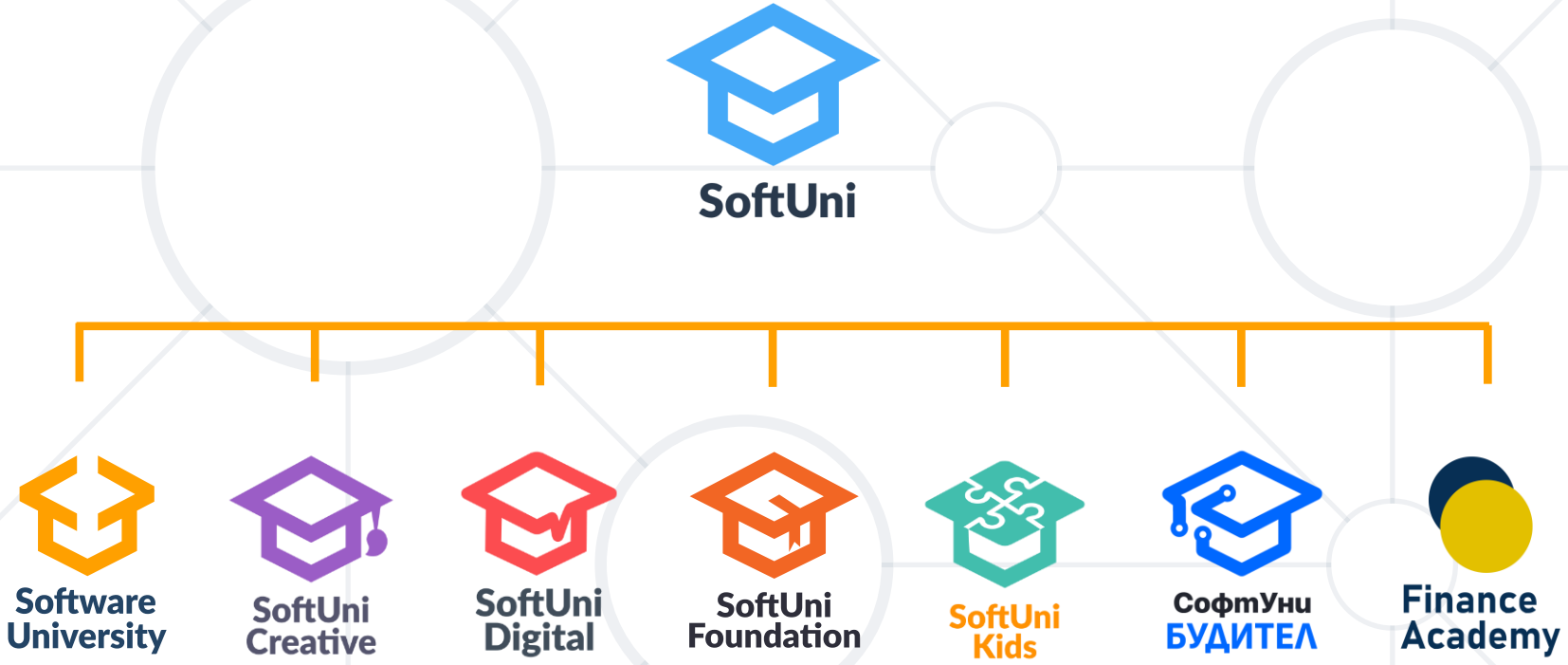
Summary

- **Standalone** components declare their own dependencies

```
standalone: true
```

- Routing allows **navigation** without **reloading** the page
- The **Router Module** in Angular is a **powerful** tool
  - It supports routing with **params**, **child** routes, route **guards**, **resolvers** and more

Questions?



SoftUni Diamond Partners

Software University

SmartIT

Coca-Cola HBC Bulgaria

SUPER HOSTING .BG

createX

Postbank

Решения за твоето утре

DRAFT KINGS

THE CROWN IS YOURS

VIVACOM

AMBITIONED

INDEAVR

Serving the high achievers

PHAR VISION

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, softuni.org
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – https://softuni.org
- © Software University – https://softuni.bg