

How to use Standalone Components with Angular Router

[Leave a Comment](#) / [10 minutes of reading](#) / [March 8, 2024](#)

[← Angular Tutorial](#)

[Angular Tutorial](#)

[Angular Tutorial →](#)

In this step-by-step tutorial, we learn how to use Standalone Components with Angular Router. Angular introduced a standalone Component API in version 14. It became a stable API in the Angular 15. One of the biggest benefits of the Standalone Component is that it offers better Lazy loading than the Module-based API. In this guide, we will show you how to set up and configure the Angular routes to Standalone Components. We will show you this by creating an Angular Routing Example application with Standalone Components and creating menu navigation system using the Angular Router.

Table of Contents

[What is Angular Routing](#)

[How to configure Angular Router](#)

[Set the <base href>](#)

[Define the routes](#)

[Register the Routes](#)

[Map Action to Routes](#)

[Choose where you want to display](#)

[Standalone Components with Angular Router: Example application](#)

[Create a new Angular Standalone project](#)

[Create Routes to Component](#)[Register the Routes](#)[Design the Navigation system](#)[Display the component using Router-outlet](#)[Running the Application](#)[References](#)

What is Angular Routing

Routing allows you to move from one part of the application to another part or one View to another View. The Angular Router Module provides the necessary services to handle the Routing. The Router is a separate module and it is present in the library package, [@angular/router](#).

The Angular Router provides the necessary service providers and directives for navigating through application views.

How to configure Angular Router

To Configure the Router in Angular in standalone components, you need to follow these steps

- Set the <base href>
- Define routes for the view
- Register the Router Service with Routes at application startup
- Map HTML Element actions to Route
- Choose where you want to display the view

Set the <base href>

The HTML `<base>` element specifies the base URL to use for all relative URLs contained within a document.

The Angular Router uses the HTML5 style of Routing (or [PathLocationStrategy](#)) as the default option. The router makes use of the browser's history API for navigation and URL interaction.

```
1  
2 <base href="/">  
3
```

To make HTML5 routing work, we need to set up the “**base href**” in the DOM. This is done in the *index.html* file immediately after the head tag.

You can read more about the [PathLocationStrategy & HashLocationStrategy](#)

Define the routes

Next, create an array of route objects. Each route maps the path (URL Segment) to the component. You can do this in `app.routes.ts`.

```
1  
2 import { Routes } from '@angular/router';  
3 import { HomeComponent } from './home/home.component';  
4  
5 export const appRoutes: Routes = [  
6   { path: 'home', component: HomeComponent },  
7 ];  
8
```

Where

path: The URL path segment of the route. We will use this value to refer to this route elsewhere in the app

component: The component to display.

This route tells angular to render HomeComponent when the user navigates to the URL `"/home"`

Register the Routes

We need to register the routes at the time of application startup. The application startup code is in the `main.ts` file. If you open the file, this is what you will find.

```
1
2 import { bootstrapApplication } from '@angular/platform-browser';
3 import { AppComponent } from './app/app.component';
4 import { appConfig } from './app/app.config';
5
6 bootstrapApplication(AppComponent, appConfig)
7   .catch((err) => console.error(err));
8
```

`bootstrapApplication` method is responsible for rendering the root component. It expects two arguments.

1. The component to render when the application starts.
2. `ApplicationConfig` object, which provides the set of configurations for the `bootstrapApplication` method.

The `ApplicationConfig` contains only one member: the `Providers` array. In the `providers` array, we register every service that needs to be made available to the root component and all of its children. Hence we need to add the Angular router related services in the `providers` array.

The `ApplicationConfig` is configured in the file `app.config.ts`.

```
1
2 import { ApplicationConfig } from '@angular/core';
3 import { provideRouter } from '@angular/router';
4 import { appRoutes } from './app.routes';
5
6 export const appConfig: ApplicationConfig = {
7   providers: [provideRouter(appRoutes)]
8 };
9
```

The `provideRouter` is a helper method that registers the services related to Angular Router and configures the routing. This is where we register the routes we created.

`provideRouter(appRoutes)`

Map Action to Routes

Next, we need to bind the click event of the link, image, or button to a route. This is done using the [routerLink](#) directive.

```
1
2 <a class="nav-link" [routerLink]="['contact']">Contact us</a>
3
```

The [routerLink](#) directive accepts an array of route names along with parameters. This array is called a [Link Parameters array](#).

When the application requests navigation to the route “**contact**”, the router looks in the routes array and activates the instance of the component associated with the route “**contact**”, which is `ContactComponent`. The browser address location & history are updated to `/contact`

Choose where you want to display

Finally, we need to tell the angular where to display the view. This is done using the `RouterOutlet` directive as shown. We will add the following directive to the root component

```
1  
2 <router-outlet></router-outlet>  
3
```

Standalone Components with Angular Router: Example application

Let's build an Angular Router example application with two components and build a navigation system. You can download the [source code from here](#)

Create a new Angular Standalone project

Go ahead and create a new Angular project `routingdemo` using the standalone API.

On Angular 17 or higher

Angular 17 and above creates the Angular projects using Standalone Components as the default option.

```
1  
2 ng new routingdemo  
3
```

Create the following components. Again no need to specify the `standalone` attribute as they are defaults

```
1  
2 ng g c home/HomeComponent --inline-style --inline-template --skip-tests --flat  
3 ng g c home/ContactComponent --inline-style --inline-template --skip-tests --flat  
4 ng g c ErrorComponent --inline-style --inline-template --skip-tests --flat  
5
```

On Angular 16

In Angular 16, you need to include the `--standalone` flag to create the standalone project.

```
1  
2 ng new routingdemo --standalone  
3
```

Similarly, you need to include the `standalone` attribute while creating the components.

```
1  
2 ng g c home/HomeComponent --inline-style --inline-template --standalone --skip-tests --flat  
3 ng g c home/ContactComponent --inline-style --inline-template --standalone --skip-tests --fl  
4 ng g c ErrorComponent --inline-style --inline-template --standalone --skip-tests --flat  
5
```

Create Routes to Component

We created `HomeComponent` & `ContactComponent` in the previous section. Now let us create routes to them.

It is good practice to create all our route configurations in a separate file. The [ng new](#) automatically creates **app.routes.ts** under the app folder. Open it and add the following routes.

```
1
2 import { Routes } from '@angular/router';
3 import { HomeComponent } from './home/home.component';
4 import { ContactComponent } from './home/contact.component';
5 import { ErrorComponent } from './error.component';
6
7 export const appRoutes: Routes = [
8   { path: 'home', component: HomeComponent },
9   { path: 'contact', component: ContactComponent },
10  { path: '', redirectTo: 'home', pathMatch: 'full' },
11  { path: '**', component: ErrorComponent }
12 ];
13
```

First, we import Routes from the router module

```
1
2 import { Routes } from '@angular/router';
3
```

Next, we need to import all the components, that require routing. We have imported Home, Contact and Error Components

Finally, we define a constant (`appRoutes`) that contains the **Routes** that we wish to create. The **Routes** is an array of **route configuration objects** (or route objects).

Each route has several configurable properties.

Our First route is

```
1  
2 { path: 'home', component: HomeComponent },  
3
```

The first parameter is the **path**, which represents the URL path segment. The second parameter is the **component** to display. The above route configuration means, when you navigate to `/home` (URL path segment), then the `HomeComponent` gets displayed.

Note that the path does not contain the leading slash

The next route is similar to the home route

```
1  
2 { path: 'contact', component: ContactComponent },  
3  
4
```

Default Route

The third route is **default route**

```
1  
2  
3 { path: '', redirectTo: 'home', pathMatch: 'full' },  
4
```

When the path is empty, it indicates a **default route**. The default route is redirected to the home path using the `RedirectTo` argument. This route means that, when you navigate to the root of your application `/`, you are redirected to the home path (`/home`), which in turn displays the `HomeComponent`.

Note, that we have `pathMatch` argument set to `'full'`. The `pathMatch` tells the Router how to match the URL. When it is set to `full`, the path is matched to the entire URL.

Every route ends in an empty space for ex: `/contact/`. If `pathMatch` is not set to `full` then the router will apply the redirect to every route, which results in the error.

Wild Card Route

The next route is the **wildcard route**

The `“***”` matches every URL. The Router will display the `ErrorComponent`.

Order matters: First one wins

Note that the order of the route is important. The Routes are matched in the order they are defined. The Router always returns the first matching route (first-match-wins strategy)

Since the wildcard route (**) matches every URL. we should be place them last.

Now, we have set up our routes. Now we will add these routes to our application.

Register the Routes

We register the routes with the Angular Router Service, when we configure it during the application startup.

To do that open the `app.config.ts`. This file created automatically by the `ng new`.

```
1
2 import { ApplicationConfig } from '@angular/core';
3 import { provideRouter } from '@angular/router';
4 import { appRoutes } from './app.routes';
5
6 export const appConfig: ApplicationConfig = {
7   providers: [provideRouter(appRoutes)]
8 };
9
```

import the `appRoutes` that we created in the previous step.

```
1
2 import { appRoutes } from './app.routes';
3
```

Pass the routes as the argument to the `provideRouter` method. `provideRouter` method registers the routes in the Angular Router service and adds all the router related services into the `Providers` array of the `ApplicationConfig`.

```
1
2 export const appConfig: ApplicationConfig = {
3   providers: [provideRouter(appRoutes)]
4 };
```

5
6

The `ApplicationConfig` is passed to the `bootstrapApplication` at the time of application startup.

```
1
2 import { bootstrapApplication } from '@angular/platform-browser';
3 import { AppComponent } from './app/app.component';
4 import { appConfig } from './app/app.config';
5
6 bootstrapApplication(AppComponent, appConfig)
7   .catch((err) => console.error(err));
8
```

Design the Navigation system

The next step is to design the navigation. We use `AppComponent` to handle the navigation. It will display the menu option, which the user can click to navigate to a view.

To do that we need to use the `RouterLink` & `RouterOutlet` directives from the Angular Router Module. Hence we need to import them to `AppComponent`.

In the older module based API, we used to import the Router Module in the `NgModule`. All the Components, Directives and Pipes that are part of the `NgModule` would automatically get to use them.

But, in Standalone API, there is no `NgModule`. Hence whatever the component needs, we need to import it in the component itself.

Open the `app.component.ts`.

```
1
2 import { Component } from '@angular/core';
3 import { RouterModule } from '@angular/router';
4
5 @Component({
6   selector: 'app-root',
7   standalone: true,
8   imports: [RouterModule],
9   templateUrl: './app.component.html',
10  styleUrls: ['./app.component.css']
11 })
12 export class AppComponent {
13   title = 'routingdemo';
14 }
15
```

Add the RouterModule to the imports array.

```
1
2 imports: [RouterModule],
3
```

Since, we are using RouterLink & RouterOutlet from the Router Module, we can import them, instead of importing the entire Router Module.

```
1
2 imports: [RouterLink, RouterOutlet],
3
```

Now, Open the app.component.html.

```
1
2 <div class="container">
3   <nav class="navbar navbar-expand-lg navbar-light bg-light">
4     <div class="container-fluid">
5       <a class="navbar-brand" href="#">{{ title }}</a>
6       <button
7         class="navbar-toggler"
8         type="button"
9         data-bs-toggle="collapse"
10        data-bs-target="#navbarSupportedContent"
11        aria-controls="navbarSupportedContent"
12        aria-expanded="false"
13        aria-label="Toggle navigation"
14      >
15        <span class="navbar-toggler-icon"></span>
16      </button>
17      <div class="collapse navbar-collapse" id="navbarSupportedContent">
18        <ul class="navbar-nav me-auto mb-2 mb-lg-0">
19          <li class="nav-item">
20            <a class="nav-link" [routerLink]="['home']">Home</a>
21          </li>
22          <li class="nav-item">
23            <a class="nav-link" [routerLink]="['contact']">Contact us</a>
24          </li>
25        </ul>
26      </div>
27    </div>
28  </nav>
29
30  <router-outlet></router-outlet>
31 </div>
32
33
```

We are using [bootstrap](#) to style our components. You will find the link to bootstrap libraries in the index.html

We use the [routerLink](#) directive to bind anchor tag elements to the route

```
2
3     <li class="nav-item">
4       <a class="nav-link" [routerLink]="['home']">Home</a>
5     </li>
6     <li class="nav-item">
7       <a class="nav-link" [routerLink]="['contact']">Contact us</a>
8     </li>
9
```

[RouterLink](#) is an [attribute directive](#). We enclose it in a square bracket. The [routerLink](#) is then bound to the template expression, which returns a **link parameters array**.

The **Link Parameters array** is the Parameters or arguments to the Route. The Angular Router module constructs the URL using the link parameters array

In the example above, ([routerLink]="['home']") the string “home” is passed to the routerLink directive. The Angular router constructs the url as /home for the anchor link.

When the user clicks on the link, the Angular Router uses the path to locate the route associated with the path and activates the component.

Display the component using Router-outlet

Finally, we need to tell Angular where to display the Component. To do that, we use the **Router-outlet directive**

The RouterOutlet is a directive that tells Angular where on our page we want to display the view.

```
1
2 <router-outlet></router-outlet>
3
```

That's it. Download the complete [source code](#)

Running the Application

Go to `http://localhost:4200`, you should see the HomeComponent is rendered, which is the default root.

Type the invalid URL and you should see the ErrorComponent rendered.

Click on the menu options or Type the Back & forward button in the browser. Everything should work as intended.



References

Related Posts