# Routing and Navigation

Building Efficient and Navigable Applications

**Scott Tremaine**
*Software Developer and Educator*

Breakpoint Coding Tutorials

# Contents

# Introduction to Routing

## Overview of Routing in Web Applications

Routing is a mechanism in web development that allows users to navigate between different parts of an application when they interact with elements, typically links. This mechanism is often found in single-page applications (SPAs) like those typically built with Angular. Unlike traditional multi-page websites where each click results in a new page load, SPAs handle these navigations client-side. The router intercepts URL changes and fetches only the necessary content or component, updating the view dynamically without reloading the page.

## Definition and Importance of Routing

Routing is defined as the process of selecting a path in an application's component tree based on a browser URL or an action. It is used for:

- Maintaining bookmarkable URLs for specific application states.
- Enabling browser history navigation, like forward and back actions.
- Handling deep linking, where users can be taken directly to a specific part of an application through a URL.

In Angular, routing is handled by the Angular Router, which is a powerful and configurable tool that maps URLs to components.

## How Routing Enhances User Experience

The use of routing significantly enhances the user experience in several ways:

- Since SPAs load all necessary resources (JavaScript, CSS, etc.) in the first load, subsequent navigations require less data fetch, making the application faster and more responsive.
- Routing allows developers to define paths that make sense from the user's perspective, often mimicking the folder paths of a file system.
- By dividing the application into different components that can be loaded as needed, routing enables a modular application structure. This not only improves performance but also organizes code in a maintainable way.
- Modern applications often need to maintain state across various views. Routing facilitates state management by allowing states to be tied to routes, either in the URL itself or behind the scenes.

# Step-by-Step Guide to Route Configuration

## Create Standalone Components

First, we need to create the components that will be routed. Each component is standalone, meaning it declares its own dependencies. For this example, we will create two simple components: `HomeComponent` and `AboutComponent`.

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

@Component({
  selector: 'app-home',
  standalone: true,
  imports: [],
  templateUrl: './home.component.html',
  styleUrl: './home.component.css'
})
export class HomeComponent {

}

import { Component } from '@angular/core';

@Component({
  selector: 'app-about',
  standalone: true,
  imports: [],
  templateUrl: './about.component.html',
  styleUrl: './about.component.css'
})
export class AboutComponent {

}
```

## Setup Routing Configuration

### Update app.routes.ts

```
import { HomeComponent } from './home/home.component';
import { AboutComponent } from './about/about.component';

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

### Imports

```
import { HomeComponent } from './home/home.component';
import { AboutComponent } from './about/about.component';
```

These lines import `HomeComponent` and `AboutComponent`. These components will be used to render views when specific routes are navigated to.

### Routes Configuration

```
export const routes = [ ... ];
```

This line defines and exports an array named `routes`, which Angular uses to handle navigation within the application based on the browser's URL path. Each object in the `routes` array represents a route definition with two properties:

- **path:** A string that specifies the URL path. For example, `path: ''` represents the root of the application (often the "homepage"), and `path: 'about'` represents a specific segment (i.e., `/about`).
- **component:** Specifies which Angular component should be rendered when the route is activated. In this case, navigating to the root URL (`/`) will render `HomeComponent`, and navigating to `/about` will render `AboutComponent`.

**Ensure that `app.config.ts` is handling the routes properly**

```
import { ApplicationConfig } from '@angular/core';
import { provideRouter } from '@angular/router';
import { routes } from './app.routes';

export const appConfig: ApplicationConfig = {
  providers: [provideRouter(routes)]
};
```

## Updating the Root Component and Template

`app.component.ts`

```
import { Component } from '@angular/core';
import { RouterLink, RouterLinkActive, RouterOutlet } from '@angular/
    router';
import { HomeComponent } from './home/home.component';
import { AboutComponent } from './about/about.component';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, RouterLink, RouterLinkActive, HomeComponent,
    AboutComponent],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'todo-app';
}
```

The main change here is to import the libraries needed to implement router linking (`RouterOutlet`, `RouterLink`, `RouterLinkActive`).

`app.component.html`

```
<nav>
  <ul>
    <li><a routerLink="/">Home</a></li>
    <li><a routerLink="/about">About</a></li>
  </ul>
</nav>
<router-outlet></router-outlet>
```

**routerLink** The `routerLink` directive is used within the `<a>` anchor tags in your navigation menu. It instructs Angular's Router to handle link clicks that change the view without reloading the page, typical of SPA behavior. When you use `routerLink`, Angular intercepts the click events on these links. Instead of the browser navigating to a new URL with a page reload, Angular's Router changes the browser's URL and displays the corresponding component for that route without reloading the page.

**router-outlet** The `router-outlet` directive acts as a placeholder that Angular dynamically fills based on the current router state. It tells Angular where to render the component associated with the current URL in the browser's address bar. Whenever a route is activated via `routerLink` or directly through the URL bar, Angular Router looks up the route configuration to find the appropriate component to display. It then renders this component within the `<router-outlet>` tag in the HTML template.

# Basic Route Parameters and Linking Techniques

When building Angular applications, defining routes with parameters is essential for creating dynamic content that adapts based on user input or other criteria. In addition to basic routing, Angular provides powerful techniques for linking within applications. Here's a detailed explanation of how to implement basic route parameters and effective linking techniques.

## Create the Service

First, we need to define a service that handles fetching user data. Since the data is static in this case, we'll simulate an asynchronous operation using Observable to mimic real-world scenarios where data might come from an external API.

```
ng generate service services/user
```

```
import { Injectable } from '@angular/core';
import { Observable, of } from 'rxjs';

export interface User {
  id: number;
  name: string;
}

@Injectable({
  providedIn: 'root'  // This ensures our service is available
   application-wide
})
export class UserService {
  private users: User[] = [
    { id: 1, name: 'Alice' },
    { id: 2, name: 'Bob' },
    { id: 3, name: 'Charlie' }
  ];

  constructor() {}

  getUsers(): Observable<User[]> {
```

```
    return of(this.users);
  }

  getUserById(id: number): Observable<User | undefined> {
    // Return Observable of User or undefined
    const user = this.users.find(user => user.id === id);
    return of(user);
  }
}
```

### Purpose and Usage of `of`

The `of` function from RxJS is a utility that you can use to create a new Observable instance that emits the values you provide as arguments. When you use `of`, it immediately emits the values you pass to it and then completes. It's a simple and useful way to create a quick observable stream from a value or set of values.

`OF` is particularly useful in scenarios where you need to provide a simple observable for testing purposes or when mocking services in Angular applications. It helps simulate real-world observable behavior without needing to set up complex observables.

In Angular services or components, you can use `of` to return static or mock data, simulating the behavior of asynchronous operations like HTTP requests. This can be particularly useful during initial development phases or when writing unit tests.

If your function or service method must return an Observable, but the data you want to return is statically available or already computed, `of` provides a straightforward way to meet this requirement.

## Create the Components

Next, we'll create two components: `UserList` and `UserDetails`.

### user-list.component.ts

This component will display a list of users. Each user in the list will have a link that navigates to the `UserDetails` component, which displays details for that user.

```
import { Component, OnInit } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterLink, RouterOutlet } from '@angular/router';
import { UserService } from '../user.service';
import { User } from '../user.interface';

@Component({
  selector: 'app-user-list',
  templateUrl: './user-list.component.html',
  standalone: true,
  imports: [CommonModule, RouterLink, RouterOutlet]
})
export class UserListComponent implements OnInit {
  users: User[] = [];
```

```
  constructor(private userService: UserService) {} // Inject the
   UserService

  ngOnInit(): void {
    this.userService.getUsers().subscribe(users => this.users = users);
  }
}
```

### user-list.component.html

```
<ul>
    <li *ngFor="let user of users">
      <a [routerLink]="['/user', user.id]">{{ user.name }}</a>
    </li>
  </ul>
```

### user-details.component.ts

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { CommonModule } from '@angular/common';
import { UserService } from '../user.service'; // Import the service
   and User interface
import { User } from '../user.interface';

@Component({
  selector: 'app-user-details',
  templateUrl: './user-details.component.html',
  standalone: true,
  imports: [CommonModule]
})
export class UserDetailsComponent implements OnInit {
  user: User | undefined;

  constructor(private route: ActivatedRoute, private userService:
   UserService) {}

  ngOnInit(): void {
    this.route.params.subscribe(params => {
      const id = +params['id'];
      this.userService.getUserById(id).subscribe(user => {
        this.user = user ?? { id: -1, name: 'No User' }; // Default
   user
      });
    });
  }
}
```

### user-details.component.html

```
<div *ngIf="user">
  <h2>{{ user.name }}</h2>
  <p>ID: {{ user.id }}</p>
```

```
</div>
<div *ngIf="!user">
  <p>User not found.</p>
</div>
```

**app.routes.ts**

```
import { Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { AboutComponent } from './about/about.component';
import { NotFoundComponent } from './not-found/not-found.component';
import { UserListComponent } from './user-list/user-list.component';
import { UserDetailsComponent } from './user-details/user-details.
   component';

export const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'user', component: UserListComponent },
  { path: 'user/:id', component: UserDetailsComponent },
  { path: '**', component: NotFoundComponent }
];
```

# Wildcard and Redirect Routes

Implementing wildcard and redirect routes in Angular is essential for handling navigation errors such as accessing undefined paths (404 errors) and managing default paths effectively.

## Using Redirect Routes for Default Navigation Paths

Redirect routes are useful for guiding users to a default page, such as redirecting from an empty path to a homepage or a dashboard as the default view.

### Determine which path you want as your default or home page

Suppose you want `/home` to be the default:

### Create a Home Component

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

@Component({
  selector: 'app-home',
  standalone: true,
  imports: [],
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.css']
})
export class HomeComponent {

}
```

### Add a route for the Home Component

```
import { HomeComponent } from './home/home.component';

export const routes = [
  { path: 'home', component: HomeComponent, standalone: true }
];
```

### Set the home component to be the default page

```
import { HomeComponent } from './home/home.component';

export const routes = [
  { path: '', redirectTo: '/home', pathMatch: 'full' as 'full' },
  { path: 'home', component: HomeComponent, standalone: true }
];
```

- **path: ''**: This property specifies the path that triggers the route action. An empty string ('') indicates that this route should be activated when there is no additional path beyond the base URL of the application.
- **redirectTo: '/home'**: This property instructs Angular's router what URL to redirect to when the route is activated. Here, it redirects to the `/home` path.
- **pathMatch: 'full' as 'full'**: The `pathMatch` property tells the router how to match the URL path to the route's path. 'full' means that the entire URL path must exactly match the path for this route configuration to take effect.

## Implementing Wildcard Routes for Handling 404 Pages

Wildcard routes in Angular are used to catch undefined paths and usually redirect users to a custom 404 Not Found page.

### Create a 404 Component

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

@Component({
  selector: 'app-not-found',
  standalone: true,
  imports: [],
  templateUrl: './not-found.component.html',
  styleUrls: ['./not-found.component.css']
})
export class NotFoundComponent {

}
```

```
<h1>404 Not Found</h1>
<p>The page you are looking for does not exist.</p>
```

### Add a Wildcard Route

```
import { HomeComponent } from './home/home.component';
import { NotFoundComponent } from './not-found/not-found.component';

export const routes = [
  { path: '', redirectTo: '/home', pathMatch: 'full' as 'full' },
  { path: 'home', component: HomeComponent, standalone: true },
  { path: '**', component: NotFoundComponent }
];
```

This setup ensures that navigation to any undefined path will lead to the `NotFoundComponent`, effectively managing users' navigation errors by providing a clear 404 error page.

## Order of Route Definitions

The order in which routes are defined in Angular matters because Angular uses a first-match-wins strategy. If a general path (like a wildcard path '**') is placed before more specific paths, it might preemptively match and render the wrong component.

# Nested Routes and Complex Routing Structures

To demonstrate nested routes and complex routing structures, we'll create an example application structure that includes parent and child routes, nested `<router-outlet>` elements, and managing hierarchical navigation.

## Define Components and Their Routes

### Dashboard Component (Parent):

This component acts as a parent route that contains additional nested routes.

```
import { Component } from '@angular/core';
import { RouterOutlet, RouterLink, RouterLinkActive } from '@angular/
    router';
import { SettingsComponent } from '../settings/settings.component';

@Component({
  selector: 'app-dashboard',
  templateUrl: './dashboard.component.html',
  standalone: true,
  imports: [RouterOutlet, RouterLink, RouterLinkActive]
})
export class DashboardComponent {}
```

```
<h1>Dashboard</h1>
<nav>
  <li><a routerLink="profile">Profile</a></li>
  <li><a routerLink="settings">Settings</a></li>
</nav>
<router-outlet></router-outlet>
```

### Profile Component (Child of Dashboard)

A child component that will be rendered within the DashboardComponent.

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

@Component({
  selector: 'app-profile',
  templateUrl: './profile.component.html',
  standalone: true
})
export class ProfileComponent {}
```

```
<p>Profile Component</p>
```

### Settings Component (Child of Dashboard):

Another child component under DashboardComponent.

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

@Component({
  selector: 'app-settings',
  templateUrl: './settings.component.html',
  standalone: true
})
export class SettingsComponent {}
```

```
<p>Settings Component</p>
```

### Define Nested Routes

Update `app.routes.ts` to include nested routes for the DashboardComponent.

```
import { HomeComponent } from './home/home.component';
import { NotFoundComponent } from './not-found/not-found.component';
import { DashboardComponent } from './dashboard/dashboard.component';
import { ProfileComponent } from './profile/profile.component';
import { SettingsComponent } from './settings/settings.component';

export const routes = [
  { path: '', redirectTo: '/home', pathMatch: 'full' as 'full' },
  { path: 'home', component: HomeComponent, standalone: true },
  {
    path: 'dashboard',
    component: DashboardComponent,
    standalone: true,
    children: [
      { path: 'profile', component: ProfileComponent, standalone: true
   },
      { path: 'settings', component: SettingsComponent, standalone:
   true }
    ]
  },
  { path: '**', component: NotFoundComponent }
];
```

### Route Configuration Breakdown

**Main Route Configuration:**

- **path: 'dashboard'**: Specifies the URL path that will activate this route. When a user navigates to `/dashboard`, the DashboardComponent is loaded.
- **component: DashboardComponent**: Defines which component Angular should instantiate and display when the route is activated.
- **standalone: true**: Indicates that the DashboardComponent is a standalone component, allowing it to be declared without being associated with an Angular module.

**Nested/Child Routes Configuration:**

- **children:** [...]: Defines an array of route configurations, each of which is a child route of the parent `/dashboard` route.
- **path: 'profile', component: ProfileComponent**: Specifies that the ProfileComponent should be rendered when this route is activated.
- **path: 'settings', component: SettingsComponent**: Similar to the profile route, this is nested under the dashboard route.

### Add route links to AppComponent

```
<nav>
  <li><a routerLink="home">home</a></li>
  <li><a routerLink="dashboard">Dashboard</a></li>
</nav>
<router-outlet></router-outlet>
```