In-app navigation: routing to views

Contents ›

In a single-page app, you change what the user sees by showing or hiding portions of the display that correspond to particular components, rather than going out to the server to get a new page. As users perform application tasks, they need to move between the different views that you have defined.

To handle the navigation from one view to the next, you use the Angular `Router`. The `Router` enables navigation by interpreting a browser URL as an instruction to change the view.

To explore a sample app featuring the router's primary features, see the live example / download example.

Prerequisites

Before creating a route, you should be familiar with the following:

- Basics of components
- Basics of templates
- An Angular app—you can generate a basic Angular app using the Angular CLI.

For an introduction to Angular with a ready-made app, see Getting Started. For a more in-depth experience of building an Angular app, see the Tour of Heroes tutorial. Both guide you through using component classes and templates.

Generate an app with routing enabled

The following command uses the Angular CLI to generate a basic Angular app with an app routing module, called `AppRoutingModule`, which is an NgModule where you can configure your routes. The app name in the following example is `routing-app`.

```
ng new routing-app --routing
```

When generating a new app, the CLI prompts you to select CSS or a CSS preprocessor. For this example, accept the default of `CSS`.

Adding components for routing

To use the Angular router, an app needs to have at least two components so that it can navigate from one to the other. To create a component using the CLI, enter the following at the command line where `first` is the name of your component:

```
ng generate component first
```

Repeat this step for a second component but give it a different name. Here, the new name is `second`.

```
ng generate component second
```

The CLI automatically appends `Component`, so if you were to write `first-component`, your component would be `FirstComponentComponent`.

> `<base href>`
>
> This guide works with a CLI-generated Angular app. If you are working manually, make sure that you have `<base href="/">` in the `<head>` of your index.html file. This assumes that the `app` folder is the application root, and uses `"/"`.

Importing your new components

To use your new components, import them into `AppRoutingModule` at the top of the file, as follows:

AppRoutingModule (excerpt)

```
import { FirstComponent } from './first/first.component';
import { SecondComponent } from './second/second.component';
```

Defining a basic route

There are three fundamental building blocks to creating a route.

Import the `AppRoutingModule` into `AppModule` and add it to the `imports` array.

The Angular CLI performs this step for you. However, if you are creating an app manually or working with an existing, non-CLI app, verify that the imports and configuration are correct. The following is the default `AppModule` using the CLI with the `--routing` flag.

Default CLI AppModule with routing

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module'; // CLI imports
AppRoutingModule
```

```
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule // CLI adds AppRoutingModule to the AppModule's imports array
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

1. Import `RouterModule` and `Routes` into your routing module.
   The Angular CLI performs this step automatically. The CLI also sets up a `Routes` array for your routes and configures the `imports` and `exports` arrays for `@NgModule()`.

   CLI app routing module

   ```
   import { NgModule } from '@angular/core';
   import { Routes, RouterModule } from '@angular/router'; // CLI imports router

   const routes: Routes = []; // sets up routes constant where you define your
   routes

   // configures NgModule imports and exports
   @NgModule({
     imports: [RouterModule.forRoot(routes)],
     exports: [RouterModule]
   })
   export class AppRoutingModule { }
   ```

2. Define your routes in your `Routes` array.
   Each route in this array is a JavaScript object that contains two properties. The first property, `path`, defines the URL path for the route. The second property, `component`, defines the component Angular should use for the corresponding path.

   AppRoutingModule (excerpt)

   ```
   const routes: Routes = [
     { path: 'first-component', component: FirstComponent },
     { path: 'second-component', component: SecondComponent },
   ];
   ```

3. Add your routes to your application.

   Now that you have defined your routes, you can add them to your application. First, add links to the two components. Assign the anchor tag that you want to add the route to the `routerLink` attribute. Set the value of the attribute to the component to show when a user clicks on each link. Next, update your component template to include `<router-outlet>`. This element informs Angular to update the application view with the component for the selected route.

   Template with routerLink and router-outlet

```
<h1>Angular Router App</h1>
<!-- This nav gives you links to click, which tells the router which route to
use (defined in the routes constant in  AppRoutingModule) -->
<nav>
  <ul>
    <li><a routerLink="/first-component" routerLinkActive="active">First
Component</a></li>
    <li><a routerLink="/second-component" routerLinkActive="active">Second
Component</a></li>
  </ul>
</nav>
<!-- The routed views render in the <router-outlet>-->
<router-outlet></router-outlet>
```

Route order

The order of routes is important because the `Router` uses a first-match wins strategy when matching routes, so more specific routes should be placed above less specific routes. List routes with a static path first, followed by an empty path route, which matches the default route. The wildcard route comes last because it matches every URL and the `Router` selects it only if no other routes match first.

Getting route information

Often, as a user navigates your application, you want to pass information from one component to another. For example, consider an application that displays a shopping list of grocery items. Each item in the list has a unique `id`. To edit an item, users click an Edit button, which opens an `EditGroceryItem` component. You want that component to retrieve the `id` for the grocery item so it can display the right information to the user.

You can use a route to pass this type of information to your application components. To do so, you use the ActivatedRoute interface.

To get information from a route:

1. Import `ActivatedRoute` and `ParamMap` to your component.

   In the component class (excerpt)

```
import { Router, ActivatedRoute, ParamMap } from '@angular/router';
```

These `import` statements add several important elements that your component needs. To learn more about each, see the following API pages:

- `Router`
- `ActivatedRoute`
- `ParamMap`

2. Inject an instance of `ActivatedRoute` by adding it to your application's constructor:

In the component class (excerpt)

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

3. Update the `ngOnInit()` method to access the `ActivatedRoute` and track the `id` parameter:

In the component (excerpt)

```
ngOnInit() {
  this.route.queryParams.subscribe(params => {
    this.name = params['name'];
  });
}
```

Note: The preceding example uses a variable, `name`, and assigns it the value based on the `name` parameter.

Setting up wildcard routes

A well-functioning application should gracefully handle when users attempt to navigate to a part of your application that does not exist. To add this functionality to your application, you set up a wildcard route. The Angular router selects this route any time the requested URL doesn't match any router paths.

To set up a wildcard route, add the following code to your `routes` definition.

AppRoutingModule (excerpt)

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

The two asterisks, `**`, indicate to Angular that this `routes` definition is a wildcard route. For the component property, you can define any component in your application. Common choices include an application-specific `PageNotFoundComponent`, which you can define to display a 404 page to your users; or a redirect to your

application's main component. A wildcard route is the last route because it matches any URL. For more detail on why order matters for routes, see Route order.

Displaying a 404 page

To display a 404 page, set up a wildcard route with the `component` property set to the component you'd like to use for your 404 page as follows:

AppRoutingModule (excerpt)

```
const routes: Routes = [
  { path: 'first-component', component: FirstComponent },
  { path: 'second-component', component: SecondComponent },
  { path: '**', component: PageNotFoundComponent },  // Wildcard route for a 404 page
];
```

The last route with the `path` of `**` is a wildcard route. The router selects this route if the requested URL doesn't match any of the paths earlier in the list and sends the user to the `PageNotFoundComponent`.

Setting up redirects

To set up a redirect, configure a route with the `path` you want to redirect from, the `component` you want to redirect to, and a `pathMatch` value that tells the router how to match the URL.

AppRoutingModule (excerpt)

```
const routes: Routes = [
  { path: 'first-component', component: FirstComponent },
  { path: 'second-component', component: SecondComponent },
  { path: '',   redirectTo: '/first-component', pathMatch: 'full' }, // redirect to
`first-component`
  { path: '**', component: PageNotFoundComponent },  // Wildcard route for a 404 page
];
```

In this example, the third route is a redirect so that the router defaults to the `first-component` route. Notice that this redirect precedes the wildcard route. Here, `path: ''` means to use the initial relative URL (`''`).

For more details on `pathMatch` see Spotlight on `pathMatch`.

Nesting routes

As your application grows more complex, you may want to create routes that are relative to a component other than your root component. These types of nested routes are called child routes. This means you're adding a second `<router-outlet>` to your app, because it is in addition to the `<router-outlet>` in `AppComponent`.

In this example, there are two additional child components, `child-a`, and `child-b`. Here, `FirstComponent` has its own `<nav>` and a second `<router-outlet>` in addition to the one in `AppComponent`.

In the template

```
<h2>First Component</h2>

<nav>
  <ul>
    <li><a routerLink="child-a">Child A</a></li>
    <li><a routerLink="child-b">Child B</a></li>
  </ul>
</nav>

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

A child route is like any other route, in that it needs both a `path` and a `component`. The one difference is that you place child routes in a `children` array within the parent route.

AppRoutingModule (excerpt)

```
const routes: Routes = [
  {
    path: 'first-component',
    component: FirstComponent, // this is the component with the <router-outlet> in the
template
    children: [
      {
        path: 'child-a', // child route path
        component: ChildAComponent, // child route component that the router renders
      },
      {
        path: 'child-b',
        component: ChildBComponent, // another child route component that the router
renders
      },
    ],
  },
];
```

Using relative paths

Relative paths allow you to define paths that are relative to the current URL segment. The following example shows a relative route to another component, `second-component`. `FirstComponent` and `SecondComponent` are at the same level in the tree, however, the link to `SecondComponent` is situated within the `FirstComponent`, meaning that the router has to go up a level and then into the second directory to find the `SecondComponent`. Rather than writing out the whole path to get to `SecondComponent`, you can use the `../` notation to go up a level.

In the template

```
<h2>First Component</h2>
```

```
<nav>
  <ul>
    <li><a routerLink="../second-component">Relative Route to second component</a></li>
  </ul>
</nav>
<router-outlet></router-outlet>
```

In addition to `../`, you can use `./` or no leading slash to specify the current level.

Specifying a relative route

To specify a relative route, use the `NavigationExtras` `relativeTo` property. In the component class, import `NavigationExtras` from the `@angular/router`.

Then use `relativeTo` in your navigation method. After the link parameters array, which here contains `items`, add an object with the `relativeTo` property set to the `ActivatedRoute`, which is `this.route`.

RelativeTo

```
goToItems() {
  this.router.navigate(['items'], { relativeTo: this.route });
}
```

The `goToItems()` method interprets the destination URI as relative to the activated route and navigates to the `items` route.

Accessing query parameters and fragments

Sometimes, a feature of your application requires accessing a part of a route, such as a query parameter or a fragment. The Tour of Heroes app at this stage in the tutorial uses a list view in which you can click on a hero to see details. The router uses an `id` to show the correct hero's details.

First, import the following members in the component you want to navigate from.

Component import statements (excerpt)

```
import { ActivatedRoute } from '@angular/router';
import { Observable } from 'rxjs';
import { switchMap } from 'rxjs/operators';
```

Next inject the activated route service:

Component (excerpt)

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

Configure the class so that you have an observable, `heroes$`, a `selectedId` to hold the `id` number of the hero, and the heroes in the `ngOnInit()`, add the following code to get the `id` of the selected hero. This code snippet assumes that you have a heroes list, a hero service, a function to get your heroes, and the HTML to render your list and details, just as in the Tour of Heroes example.

Component 1 (excerpt)

```
heroes$: Observable;
selectedId: number;
heroes = HEROES;

ngOnInit() {
  this.heroes$ = this.route.paramMap.pipe(
    switchMap(params => {
      this.selectedId = Number(params.get('id'));
      return this.service.getHeroes();
    })
  );
}
```

Next, in the component that you want to navigate to, import the following members.

Component 2 (excerpt)

```
import { Router, ActivatedRoute, ParamMap } from '@angular/router';
import { Observable } from 'rxjs';
```

Inject `ActivatedRoute` and `Router` in the constructor of the component class so they are available to this component:

Component 2 (excerpt)

```
hero$: Observable;

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

  ngOnInit() {
    const heroId = this.route.snapshot.paramMap.get('id');
    this.hero$ = this.service.getHero(heroId);
  }

  gotoItems(hero: Hero) {
    const heroId = hero ? hero.id : null;
    // Pass along the hero id if available
```

```
    // so that the HeroList component can select that item.
    this.router.navigate(['/heroes', { id: heroId }]);
  }
```

Lazy loading

You can configure your routes to lazy load modules, which means that Angular only loads modules as needed, rather than loading all modules when the app launches. Additionally, you can preload parts of your app in the background to improve the user experience.

For more information on lazy loading and preloading see the dedicated guide Lazy loading NgModules.

Preventing unauthorized access

Use route guards to prevent users from navigating to parts of an app without authorization. The following route guards are available in Angular:

- `CanActivate`
- `CanActivateChild`
- `CanDeactivate`
- `Resolve`
- `CanLoad`

To use route guards, consider using component-less routes as this facilitates guarding child routes.

Create a service for your guard:

```
ng generate guard your-guard
```

In your guard class, implement the guard you want to use. The following example uses `CanActivate` to guard the route.

Component (excerpt)

```
export class YourGuard implements CanActivate {
  canActivate(
    next: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): boolean {
      // your  logic goes here
  }
}
```

In your routing module, use the appropriate property in your `routes` configuration. Here, `canActivate` tells the router to mediate navigation to this particular route.

Routing module (excerpt)

```
{
  path: '/your-path',
  component: YourComponent,
  canActivate: [YourGuard],
}
```

For more information with a working example, see the routing tutorial section on route guards.

Link parameters array

A link parameters array holds the following ingredients for router navigation:

- The path of the route to the destination component.
- Required and optional route parameters that go into the route URL.

You can bind the `RouterLink` directive to such an array like this:

src/app/app.component.ts (h-anchor)

```
<a [routerLink]="['/heroes']">Heroes</a>
```

The following is a two-element array when specifying a route parameter:

src/app/heroes/hero-list/hero-list.component.html (nav-to-detail)

```
<a [routerLink]="['/hero', hero.id]">
  <span class="badge">{{ hero.id }}</span>{{ hero.name }}
</a>
```

You can provide optional route parameters in an object, as in `{ foo: 'foo' }`:

src/app/app.component.ts (cc-query-params)

```
<a [routerLink]="['/crisis-center', { foo: 'foo' }]">Crisis Center</a>
```

These three examples cover the needs of an app with one level of routing. However, with a child router, such as in the crisis center, you create new link array possibilities.

The following minimal `RouterLink` example builds upon a specified default child route for the crisis center.

src/app/app.component.ts (cc-anchor-w-default)

```
<a [routerLink]="['/crisis-center']">Crisis Center</a>
```

Note the following:

- The first item in the array identifies the parent route (`/crisis-center`).

- There are no parameters for this parent route.
- There is no default for the child route so you need to pick one.
- You're navigating to the `CrisisListComponent`, whose route path is `/`, but you don't need to explicitly add the slash.

Consider the following router link that navigates from the root of the application down to the Dragon Crisis:

src/app/app.component.ts (Dragon-anchor)

```
<a [routerLink]="['/crisis-center', 1]">Dragon Crisis</a>
```

- The first item in the array identifies the parent route (`/crisis-center`).
- There are no parameters for this parent route.
- The second item identifies the child route details about a particular crisis (`/:id`).
- The details child route requires an `id` route parameter.
- You added the `id` of the Dragon Crisis as the second item in the array (`1`).
- The resulting path is `/crisis-center/1`.

You could also redefine the `AppComponent` template with Crisis Center routes exclusively:

src/app/app.component.ts (template)

```
template: `
  <h1 class="title">Angular Router</h1>
  <nav>
    <a [routerLink]="['/crisis-center']">Crisis Center</a>
    <a [routerLink]="['/crisis-center/1', { foo: 'foo' }]">Dragon Crisis</a>
    <a [routerLink]="['/crisis-center/2']">Shark Crisis</a>
  </nav>
  <router-outlet></router-outlet>
`
```

In summary, you can write applications with one, two or more levels of routing. The link parameters array affords the flexibility to represent any routing depth and any legal sequence of route paths, (required) router parameters, and (optional) route parameter objects.

`LocationStrategy` and browser URL styles

When the router navigates to a new component view, it updates the browser's location and history with a URL for that view. As this is a strictly local URL the browser won't send this URL to the server and will not reload the page.

Modern HTML5 browsers support history.pushState, a technique that changes a browser's location and history without triggering a server page request. The router can compose a "natural" URL that is indistinguishable from one that would otherwise require a page load.

Here's the Crisis Center URL in this "HTML5 pushState" style:

```
localhost:3002/crisis-center/
```

Older browsers send page requests to the server when the location URL changes unless the change occurs after a "#" (called the "hash"). Routers can take advantage of this exception by composing in-application route URLs with hashes. Here's a "hash URL" that routes to the Crisis Center.

```
localhost:3002/src/#/crisis-center/
```

The router supports both styles with two `LocationStrategy` providers:

1. `PathLocationStrategy`—the default "HTML5 pushState" style.
2. `HashLocationStrategy`—the "hash URL" style.

The `RouterModule.forRoot()` function sets the `LocationStrategy` to the `PathLocationStrategy`, which makes it the default strategy. You also have the option of switching to the `HashLocationStrategy` with an override during the bootstrapping process.

> For more information on providers and the bootstrap process, see Dependency Injection.

Choosing a routing strategy

You must choose a routing strategy early in the development of you project because once the application is in production, visitors to your site use and depend on application URL references.

Almost all Angular projects should use the default HTML5 style. It produces URLs that are easier for users to understand and it preserves the option to do server-side rendering.

Rendering critical pages on the server is a technique that can greatly improve perceived responsiveness when the app first loads. An app that would otherwise take ten or more seconds to start could be rendered on the server and delivered to the user's device in less than a second.

This option is only available if application URLs look like normal web URLs without hashes (#) in the middle.

`<base href>`

The router uses the browser's history.pushState for navigation. `pushState` allows you to customize in-app URL paths; for example, `localhost:4200/crisis-center`. The in-app URLs can be indistinguishable from server URLs.

Modern HTML5 browsers were the first to support `pushState` which is why many people refer to these URLs as "HTML5 style" URLs.

HTML5 style navigation is the router default. In the LocationStrategy and browser URL styles section, learn why HTML5 style is preferable, how to adjust its behavior, and how to switch to the older hash (#) style, if necessary.

You must add a <base href> element to the app's `index.html` for `pushState` routing to work. The browser uses the `<base href>` value to prefix relative URLs when referencing CSS files, scripts, and images.

Add the `<base>` element just after the `<head>` tag. If the `app` folder is the application root, as it is for this application, set the `href` value in `index.html` as shown here.

src/index.html (base-href)

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

HTML5 URLs and the `<base href>`

The guidelines that follow will refer to different parts of a URL. This diagram outlines what those parts refer to:

```
foo://example.com:8042/over/there?name=ferret#nose
\_/   \______________/\_________/ \_________/ \__/
 |           |            |            |        |
scheme     authority      path        query   fragment
```

While the router uses the HTML5 pushState style by default, you must configure that strategy with a `<base href>`.

The preferred way to configure the strategy is to add a <base href> element tag in the `<head>` of the `index.html`.

src/index.html (base-href)

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

Without that tag, the browser may not be able to load resources (images, CSS, scripts) when "deep linking" into the app.

Some developers may not be able to add the `<base>` element, perhaps because they don't have access to `<head>` or the `index.html`.

Those developers may still use HTML5 URLs by taking the following two steps:

1. Provide the router with an appropriate `APP_BASE_HREF` value.
2. Use root URLs (URLs with an `authority`) for all web resources: CSS, images, scripts, and template HTML files.

- The `<base href>` `path` should end with a "/", as browsers ignore characters in the `path` that follow the right-most "/".
- If the `<base href>` includes a `query` part, the `query` is only used if the `path` of a link in the page is empty and has no `query`. This means that a `query` in the `<base href>` is only included when using `HashLocationStrategy`.
- If a link in the page is a root URL (has an `authority`), the `<base href>` is not used. In this way, an `APP_BASE_HREF` with an authority will cause all links created by Angular to ignore the `<base href>` value.
- A fragment in the `<base href>` is *never* persisted.

For more complete information on how `<base href>` is used to construct target URIs, see the RFC section on transforming references.

HashLocationStrategy

You can use `HashLocationStrategy` by providing the `useHash: true` in an object as the second argument of the `RouterModule.forRoot()` in the `AppModule`.

src/app/app.module.ts (hash URL strategy)

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { Routes, RouterModule } from '@angular/router';

import { AppComponent } from './app.component';
import { PageNotFoundComponent } from './page-not-found/page-not-found.component';

const routes: Routes = [

];

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    RouterModule.forRoot(routes, { useHash: true })  // .../#/crisis-center/
  ],
  declarations: [
    AppComponent,
    PageNotFoundComponent
  ],
  providers: [

  ],
  bootstrap: [ AppComponent ]
```

```
})
export class AppModule { }
```

Router Reference

The folllowing sections highlight some core router concepts.

Router imports

The Angular Router is an optional service that presents a particular component view for a given URL. It is not part of the Angular core and thus is in its own library package, `@angular/router`.

Import what you need from it as you would from any other Angular package.

src/app/app.module.ts (import)

```
import { RouterModule, Routes } from '@angular/router';
```

> For more on browser URL styles, see `LocationStrategy` and browser URL styles.

Configuration

A routed Angular application has one singleton instance of the `Router` service. When the browser's URL changes, that router looks for a corresponding `Route` from which it can determine the component to display.

A router has no routes until you configure it. The following example creates five route definitions, configures the router via the `RouterModule.forRoot()` method, and adds the result to the `AppModule`'s `imports` array.

src/app/app.module.ts (excerpt)

```
const appRoutes: Routes = [
  { path: 'crisis-center', component: CrisisListComponent },
  { path: 'hero/:id',      component: HeroDetailComponent },
  {
    path: 'heroes',
    component: HeroListComponent,
    data: { title: 'Heroes List' }
  },
  { path: '',
    redirectTo: '/heroes',
    pathMatch: 'full'
  },
  { path: '**', component: PageNotFoundComponent }
];
```

```
@NgModule({
  imports: [
    RouterModule.forRoot(
      appRoutes,
      { enableTracing: true } // <-- debugging purposes only
    )
    // other imports here
  ],
  ...
})
export class AppModule { }
```

The `appRoutes` array of routes describes how to navigate. Pass it to the `RouterModule.forRoot()` method in the module `imports` to configure the router.

Each `Route` maps a URL `path` to a component. There are no leading slashes in the path. The router parses and builds the final URL for you, which allows you to use both relative and absolute paths when navigating between application views.

The `:id` in the second route is a token for a route parameter. In a URL such as `/hero/42`, "42" is the value of the `id` parameter. The corresponding `HeroDetailComponent` uses that value to find and present the hero whose `id` is 42.

The `data` property in the third route is a place to store arbitrary data associated with this specific route. The data property is accessible within each activated route. Use it to store items such as page titles, breadcrumb text, and other read-only, static data. You can use the resolve guard to retrieve dynamic data.

The empty path in the fourth route represents the default path for the application—the place to go when the path in the URL is empty, as it typically is at the start. This default route redirects to the route for the `/heroes` URL and, therefore, displays the `HeroesListComponent`.

If you need to see what events are happening during the navigation lifecycle, there is the `enableTracing` option as part of the router's default configuration. This outputs each router event that took place during each navigation lifecycle to the browser console. Use `enableTracing` only for debugging purposes. You set the `enableTracing: true` option in the object passed as the second argument to the `RouterModule.forRoot()` method.

Router outlet

The `RouterOutlet` is a directive from the router library that is used like a component. It acts as a placeholder that marks the spot in the template where the router should display the components for that outlet.

```
<router-outlet></router-outlet>
<!-- Routed components go here -->
```

Given the configuration above, when the browser URL for this application becomes `/heroes`, the router matches that URL to the route path `/heroes` and displays the `HeroListComponent` as a sibling element to the `RouterOutlet` that you've placed in the host component's template.

Router links

To navigate as a result of some user action such as the click of an anchor tag, use `RouterLink`.

Consider the following template:

src/app/app.component.html

```
<h1>Angular Router</h1>
<nav>
  <a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
  <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
</nav>
<router-outlet></router-outlet>
```

The `RouterLink` directives on the anchor tags give the router control over those elements. The navigation paths are fixed, so you can assign a string to the `routerLink` (a "one-time" binding).

Had the navigation path been more dynamic, you could have bound to a template expression that returned an array of route link parameters; that is, the link parameters array. The router resolves that array into a complete URL.

Active router links

The `RouterLinkActive` directive toggles CSS classes for active `RouterLink` bindings based on the current `RouterState`.

On each anchor tag, you see a property binding to the `RouterLinkActive` directive that looks like `routerLinkActive="..."`.

The template expression to the right of the equal sign, =, contains a space-delimited string of CSS classes that the Router adds when this link is active (and removes when the link is inactive). You set the `RouterLinkActive` directive to a string of classes such as `[routerLinkActive]="'active fluffy'"` or bind it to a component property that returns such a string.

Active route links cascade down through each level of the route tree, so parent and child router links can be active at the same time. To override this behavior, you can bind to the `[routerLinkActiveOptions]` input binding with the `{ exact: true }` expression. By using `{ exact: true }`, a given `RouterLink` will only be active if its URL is an exact match to the current URL.

Router state

After the end of each successful navigation lifecycle, the router builds a tree of `ActivatedRoute` objects that make up the current state of the router. You can access the current `RouterState` from anywhere in the application using the `Router` service and the `routerState` property.

Each `ActivatedRoute` in the `RouterState` provides methods to traverse up and down the route tree to get information from parent, child and sibling routes.

Activated route

The route path and parameters are available through an injected router service called the ActivatedRoute. It has a great deal of useful information including:

Property	Description
`url`	An `Observable` of the route path(s), represented as an array of strings for each part of the route path.
`data`	An `Observable` that contains the `data` object provided for the route. Also contains any resolved values from the resolve guard.
`paramMap`	An `Observable` that contains a map of the required and optional parameters specific to the route. The map supports retrieving single and multiple values from the same parameter.
`queryParamMap`	An `Observable` that contains a map of the query parameters available to all routes. The map supports retrieving single and multiple values from the query parameter.
`fragment`	An `Observable` of the URL fragment available to all routes.
`outlet`	The name of the `RouterOutlet` used to render the route. For an unnamed outlet, the outlet name is primary.
`routeConfig`	The route configuration used for the route that contains the origin path.
`parent`	The route's parent `ActivatedRoute` when this route is a child route.
`firstChild`	Contains the first `ActivatedRoute` in the list of this route's child routes.
`children`	Contains all the child routes activated under the current route.

Two older properties are still available, however, their replacements are preferable as they may be deprecated in a future Angular version.

- `params`: An `Observable` that contains the required and optional parameters specific to the route. Use `paramMap` instead.
- `queryParams`: An `Observable` that contains the query parameters available to all routes. Use `queryParamMap` instead.

Router events

During each navigation, the `Router` emits navigation events through the `Router.events` property. These events range from when the navigation starts and ends to many points in between. The full list of navigation events is

displayed in the table below.

Router Event	Description
`NavigationStart`	An event triggered when navigation starts.
`RouteConfigLoadStart`	An event triggered before the `Router` lazy loads a route configuration.
`RouteConfigLoadEnd`	An event triggered after a route has been lazy loaded.
`RoutesRecognized`	An event triggered when the Router parses the URL and the routes are recognized.
`GuardsCheckStart`	An event triggered when the Router begins the Guards phase of routing.
`ChildActivationStart`	An event triggered when the Router begins activating a route's children.
`ActivationStart`	An event triggered when the Router begins activating a route.
`GuardsCheckEnd`	An event triggered when the Router finishes the Guards phase of routing successfully.
`ResolveStart`	An event triggered when the Router begins the Resolve phase of routing.
`ResolveEnd`	An event triggered when the Router finishes the Resolve phase of routing successfuly.
`ChildActivationEnd`	An event triggered when the Router finishes activating a route's children.
`ActivationEnd`	An event triggered when the Router finishes activating a route.
`NavigationEnd`	An event triggered when navigation ends successfully.
`NavigationCancel`	An event triggered when navigation is canceled. This can happen when a Route Guard returns false during navigation, or redirects by returning a `UrlTree`.
`NavigationError`	An event triggered when navigation fails due to an unexpected error.
`Scroll`	An event that represents a scrolling event.

When you enable the `enableTracing` option, Angular logs these events to the console. For an example of filtering router navigation events, see the router section of the Observables in Angular guide.

Router terminology

Here are the key `Router` terms and their meanings:

Router Part	Meaning
`Router`	Displays the application component for the active URL. Manages navigation from one component to the next.
`RouterModule`	A separate NgModule that provides the necessary service providers and directives for navigating through application views.
`Routes`	Defines an array of Routes, each mapping a URL path to a component.
`Route`	Defines how the router should navigate to a component based on a URL pattern. Most routes consist of a path and a component type.
`RouterOutlet`	The directive (`<router-outlet>`) that marks where the router displays a view.
`RouterLink`	The directive for binding a clickable HTML element to a route. Clicking an element with a `routerLink` directive that is bound to a *string* or a *link parameters array* triggers a navigation.
`RouterLinkActive`	The directive for adding/removing classes from an HTML element when an associated `routerLink` contained on or inside the element becomes active/inactive.
`ActivatedRoute`	A service that is provided to each route component that contains route specific information such as route parameters, static data, resolve data, global query params, and the global fragment.
`RouterState`	The current state of the router including a tree of the currently activated routes together with convenience methods for traversing the route tree.
Link parameters array	An array that the router interprets as a routing instruction. You can bind that array to a `RouterLink` or pass the array as an argument to the `Router.navigate` method.
Routing component	An Angular component with a `RouterOutlet` that displays views based on router navigations.

Version 10.0.10-local+sha.84d1ba792b.