Routing & Navigation

The Angular `Router` enables navigation from one view to the next as users perform application tasks.

This guide covers the router's primary features, illustrating them through the evolution of a small application that you can run live in the browser.

Overview

The browser is a familiar model of application navigation:

- Enter a URL in the address bar and the browser navigates to a corresponding page.
- Click links on the page and the browser navigates to a new page.
- Click the browser's back and forward buttons and the browser navigates backward and forward through the history of pages you've seen.

The Angular `Router` ("the router") borrows from this model. It can interpret a browser URL as an instruction to navigate to a client-generated view. It can pass optional parameters along to the supporting view component that help it decide what specific content to present. You can bind the router to links on a page and it will navigate to the appropriate application view when the user clicks a link. You can navigate imperatively when the user clicks a button, selects from a drop box, or in response to some other stimulus from any source. And the router logs activity in the browser's history journal so the back and forward buttons work as well.

{@a basics}

The Basics

This guide proceeds in phases, marked by milestones, starting from a simple two-pager and building toward a modular, multi-view design with child routes.

An introduction to a few core router concepts will help orient you to the details that follow.

{@a basics-base-href}

<base href>

Most routing applications should add a `<base>` element to the `index.html` as the first child in the `<head>` tag to tell the router how to compose navigation URLs.

If the `app` folder is the application root, as it is for the sample application, set the `href` value *exactly* as shown here.

{@a basics-router-imports}

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. It is in its own library package, `@angular/router`. Import what you need from it as you would from any other Angular package.

You'll learn about more options in the [details below](#browser-url-styles).

{@a basics-config}

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 four route definitions, configures the router via the `RouterModule.forRoot` method, and adds the result to the `AppModule`'s `imports` array.

{@a example-config}

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, allowing 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` will use that value to find and present the hero whose `id` is 42. You'll learn more about route parameters later in this guide.

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'll use the resolve guard to retrieve *dynamic* data later in the guide.

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, will display the `HeroesListComponent`.

The `**` path in the last route is a **wildcard**. The router will select this route if the requested URL doesn't match any paths for routes defined earlier in the configuration. This is useful for displaying a "404 - Not Found" page or redirecting to another route.

The order of the routes in the configuration matters and this is by design. The router uses a **first-match wins** strategy when matching routes, so more specific routes should be placed above less specific routes. In the configuration above, routes with a static path are listed first, followed by an empty path route, that matches the default route. The wildcard route comes last because it matches *every URL* and should be selected *only* if no other routes are matched first.

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. This should only be used for *debugging* purposes. You set the `enableTracing: true` option in the object passed as the second argument to the `RouterModule.forRoot()` method.

{@a basics-router-outlet}

Router outlet

Given this configuration, when the browser URL for this application becomes `/heroes`, the router matches that URL to the route path `/heroes` and displays the `HeroListComponent` *after* a `RouterOutlet` that you've placed in the host view's HTML.

<router-outlet></router-outlet> <!-- Routed views go here -->

{@a basics-router-links}

Router links

Now you have routes configured and a place to render them, but how do you navigate? The URL could arrive directly from the browser address bar. But most of the time you navigate as a result of some user action such as the click of an anchor tag.

Consider the following template:

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 (the *link parameters array*). The router resolves that array into a complete URL.

The **`RouterLinkActive`** directive on each anchor tag helps visually distinguish the anchor for the currently selected "active" route. The router adds the `active` CSS class to the element when the associated *RouterLink* becomes active. You can add this directive to the anchor or to its parent element.

{@a basics-router-state}

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.

{@a activated-route}

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](#resolve-guard).
`paramMap`	An `Observable` that contains a [map](api/router/ParamMap) of the required and [optional parameters](#optional-route-parameters) specific to the route. The map supports retrieving single and multiple values from the same parameter.
`queryParamMap`	An `Observable` that contains a [map](api/router/ParamMap) of the [query parameters](#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](#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](#child-routing-component).
`firstChild`	Contains the first `ActivatedRoute` in the list of this route's child routes.
`children`	Contains all the [child routes](#child-routing-component) activated under the current route.

Two older properties are still available. They are less capable than their replacements, discouraged, and may be deprecated in a future Angular version. **`params`** — An `Observable` that contains the required and [optional parameters](#optional-route-parameters) specific to the route. Use `paramMap` instead. **`queryParams`** — An `Observable` that contains the [query parameters](#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](api/router/NavigationStart) triggered when navigation starts.
`RoutesRecognized`	An [event](api/router/RoutesRecognized) triggered when the Router parses the URL and the routes are recognized.
`RouteConfigLoadStart`	An [event](api/router/RouteConfigLoadStart) triggered before the `Router` [lazy loads](#asynchronous-routing) a route configuration.
`RouteConfigLoadEnd`	An [event](api/router/RouteConfigLoadEnd) triggered after a route has been lazy loaded.
`NavigationEnd`	An [event](api/router/NavigationEnd) triggered when navigation ends successfully.
`NavigationCancel`	An [event](api/router/NavigationCancel) triggered when navigation is canceled. This is due to a [Route Guard](#guards) returning false during navigation.
`NavigationError`	An [event](api/router/NavigationError) triggered when navigation fails due to an unexpected error.

These events are logged to the console when the `enableTracing` option is enabled also. Since the events are provided as an `Observable`, you can `filter()` for events of interest and `subscribe()` to them to make decisions based on the sequence of events in the navigation process.

{@a basics-summary}

Summary

The application has a configured router. The shell component has a `RouterOutlet` where it can display views produced by the router. It has `RouterLink`s that users can click to navigate via the router.

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.

{@a sample-app-intro}

The sample application

This guide describes development of a multi-page routed sample application. Along the way, it highlights

design decisions and describes key features of the router such as:

- Organizing the application features into modules.
- Navigating to a component (*Heroes* link to "Heroes List").
- Including a route parameter (passing the Hero `id` while routing to the "Hero Detail").
- Child routes (the *Crisis Center* has its own routes).
- The `CanActivate` guard (checking route access).
- The `CanActivateChild` guard (checking child route access).
- The `CanDeactivate` guard (ask permission to discard unsaved changes).
- The `Resolve` guard (pre-fetching route data).
- Lazy loading feature modules.
- The `CanLoad` guard (check before loading feature module assets).

The guide proceeds as a sequence of milestones as if you were building the app step-by-step. But, it is not a tutorial and it glosses over details of Angular application construction that are more thoroughly covered elsewhere in the documentation.

The full source for the final version of the app can be seen and downloaded from the .

The sample application in action

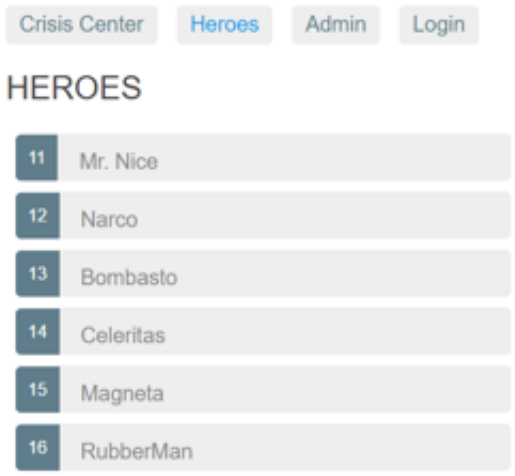
Imagine an application that helps the *Hero Employment Agency* run its business. Heroes need work and the agency finds crises for them to solve.

The application has three main feature areas:

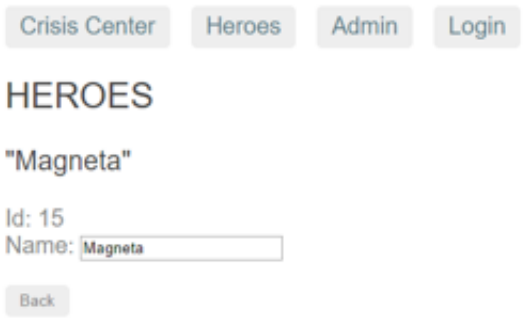
1. A *Crisis Center* for maintaining the list of crises for assignment to heroes.
2. A *Heroes* area for maintaining the list of heroes employed by the agency.
3. An *Admin* area to manage the list of crises and heroes.

Try it by clicking on this live example link.

Once the app warms up, you'll see a row of navigation buttons and the *Heroes* view with its list of heroes.



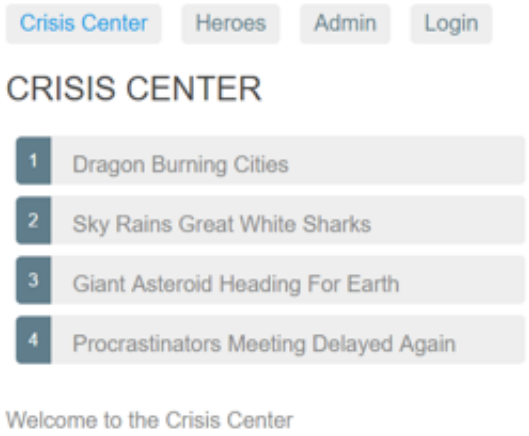
Select one hero and the app takes you to a hero editing screen.



Alter the name. Click the "Back" button and the app returns to the heroes list which displays the changed hero name. Notice that the name change took effect immediately.

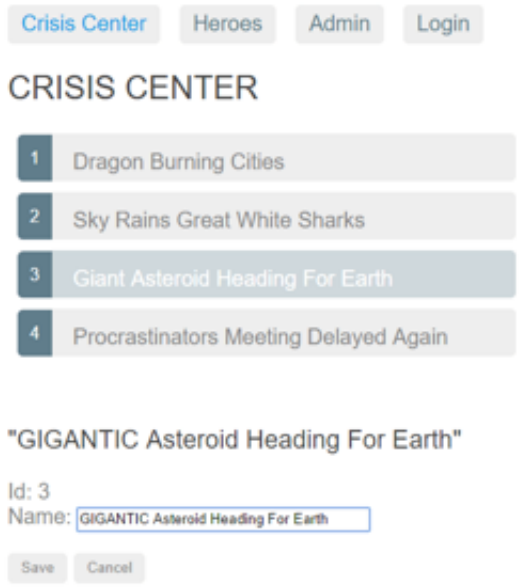
Had you clicked the browser's back button instead of the "Back" button, the app would have returned you to the heroes list as well. Angular app navigation updates the browser history as normal web navigation does.

Now click the *Crisis Center* link for a list of ongoing crises.



Select a crisis and the application takes you to a crisis editing screen. The *Crisis Detail* appears in a child view on the same page, beneath the list.

Alter the name of a crisis. Notice that the corresponding name in the crisis list does *not* change.



Unlike *Hero Detail*, which updates as you type, *Crisis Detail* changes are temporary until you either save or discard them by pressing the "Save" or "Cancel" buttons. Both buttons navigate back to the *Crisis Center* and its list of crises.

Do not click either button yet. Click the browser back button or the "Heroes" link instead.

Up pops a dialog box.

You can say "OK" and lose your changes or click "Cancel" and continue editing.

Behind this behavior is the router's `CanDeactivate` guard. The guard gives you a chance to clean-up or ask the user's permission before navigating away from the current view.

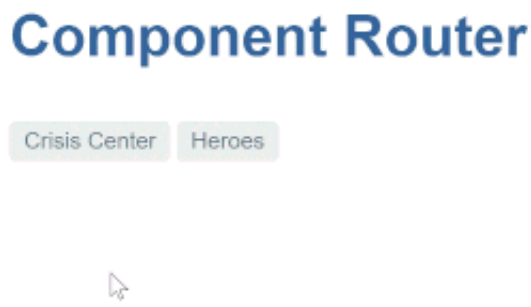
The `Admin` and `Login` buttons illustrate other router capabilities to be covered later in the guide. This short introduction will do for now.

Proceed to the first application milestone.

{@a getting-started}

Milestone 1: Getting started with the router

Begin with a simple version of the app that navigates between two empty views.



{@a base-href}

Set the *<base href>*

The router uses the browser's history.pushState for navigation. Thanks to `pushState`, you can make in-app URL paths look the way you want them to look, e.g. `localhost:3000/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](#browser-url-styles) Appendix, learn why HTML5 style is preferred, 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`** *exactly* as shown here.

Live example note
A live coding environment like Plunker sets the application base address dynamically so you can't specify a fixed address. That's why the example code replaces the `` with a script that writes the `` tag on the fly. <script>document.write('<base href="' + document.location + '" />');</script> You only need this trick for the live example, not production code.

{@a import}

Importing from the router library

Begin by importing some symbols from the router library. The Router is in its own `@angular/router` package. It's not part of the Angular core. The router is an optional service because not all applications need routing and, depending on your requirements, you may need a different routing library.

You teach the router how to navigate by configuring it with routes.

{@a route-config}

Define routes

A router must be configured with a list of route definitions.

The first configuration defines an array of two routes with simple paths leading to the `CrisisListComponent` and `HeroListComponent`.

Each definition translates to a Route object which has two things: a `path`, the URL path segment for this route; and a `component`, the component associated with this route.

The router draws upon its registry of definitions when the browser URL changes or when application code tells the router to navigate along a route path.

In simpler terms, you might say this of the first route:

- When the browser's location URL changes to match the path segment `/crisis-center`, then the router activates an instance of the `CrisisListComponent` and displays its view.
- When the application requests navigation to the path `/crisis-center`, the router activates an instance of `CrisisListComponent`, displays its view, and updates the browser's address location and history with the URL for that path.

Here is the first configuration. Pass the array of routes, `appRoutes`, to the `RouterModule.forRoot` method. It returns a module, containing the configured `Router` service provider, plus other providers that the routing library requires. Once the application is bootstrapped, the `Router` performs the initial navigation based on the current browser URL.

Adding the configured `RouterModule` to the `AppModule` is sufficient for simple route configurations. As the application grows, you'll want to refactor the routing configuration into a separate file and create a **[Routing Module](#routing-module)**, a special type of `Service Module` dedicated to the purpose of routing in feature

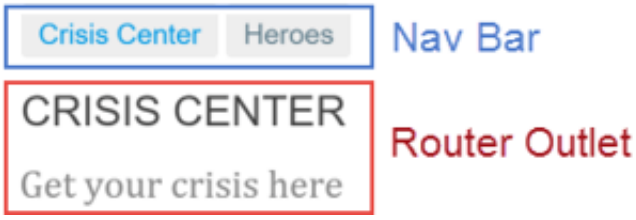
modules.

Providing the `RouterModule` in the `AppModule` makes the Router available everywhere in the application.

{@a shell}

The *AppComponent* shell

The root `AppComponent` is the application shell. It has a title, a navigation bar with two links, and a *router outlet* where the router swaps views on and off the page. Here's what you get:



{@a shell-template}

The corresponding component template looks like this:

{@a router-outlet}

RouterOutlet

The `RouterOutlet` is a directive from the router library that marks the spot in the template where the router should display the views for that outlet.

The router adds the `` element to the DOM and subsequently inserts the navigated view element immediately _after_ the ``.

{@a router-link}

RouterLink binding

Above the outlet, within the anchor tags, you see attribute bindings to the `RouterLink` directive that look like `routerLink="..."` .

The links in this example each have a string path, the path of a route that you configured earlier. There are no route parameters yet.

You can also add more contextual information to the `RouterLink` by providing query string parameters or a

URL fragment for jumping to different areas on the page. Query string parameters are provided through the `[queryParams]` binding which takes an object (e.g. `{ name: 'value' }`), while the URL fragment takes a single value bound to the `[fragment]` input binding.

Learn about the how you can also use the _link parameters array_ in the [appendix below](#link-parameters-array).

{@a router-link-active}

RouterLinkActive binding

On each anchor tag, you also see property bindings to the `RouterLinkActive` directive that look like `routerLinkActive="..."`.

The template expression to the right of the equals (=) contains a space-delimited string of CSS classes that the Router will add when this link is active (and remove when the link is inactive). You can also 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.

The `RouterLinkActive` directive toggles css classes for active `RouterLink` s based on the current `RouterState`. This cascades 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.

{@a router-directives}

Router directives

`RouterLink`, `RouterLinkActive` and `RouterOutlet` are directives provided by the Angular `RouterModule` package. They are readily available for you to use in the template.

The current state of `app.component.ts` looks like this:

{@a wildcard}

Wildcard route

You've created two routes in the app so far, one to `/crisis-center` and the other to `/heroes`. Any other URL causes the router to throw an error and crash the app.

Add a **wildcard** route to intercept invalid URLs and handle them gracefully. A *wildcard* route has a path consisting of two asterisks. It matches *every* URL. The router will select *this* route if it can't match a route earlier in the configuration. A wildcard route can navigate to a custom "404 Not Found" component or redirect to an existing route.

The router selects the route with a [_first match wins_](#example-config) strategy. Wildcard routes are the least specific routes in the route configuration. Be sure it is the _last_ route in the configuration.

To test this feature, add a button with a `RouterLink` to the `HeroListComponent` template and set the link to `"/sidekicks"`.

The application will fail if the user clicks that button because you haven't defined a `"/sidekicks"` route yet.

Instead of adding the `"/sidekicks"` route, define a `wildcard` route instead and have it navigate to a simple `PageNotFoundComponent`.

Create the `PageNotFoundComponent` to display when users visit invalid URLs.

As with the other components, add the `PageNotFoundComponent` to the `AppModule` declarations.

Now when the user visits `/sidekicks`, or any other invalid URL, the browser displays "Page not found". The browser address bar continues to point to the invalid URL.

{@a default-route}

The *default* route to heroes

When the application launches, the initial URL in the browser bar is something like:

localhost:3000

That doesn't match any of the concrete configured routes which means the router falls through to the wildcard route and displays the `PageNotFoundComponent`.

The application needs a **default route** to a valid page. The default page for this app is the list of heroes. The app should navigate there as if the user clicked the "Heroes" link or pasted `localhost:3000/heroes` into the address bar.

{@a redirect}

Redirecting routes

The preferred solution is to add a `redirect` route that translates the initial relative URL (`''`) to the desired default path (`/heroes`). The browser address bar shows `.../heroes` as if you'd navigated there directly.

Add the default route somewhere *above* the wildcard route. It's just above the wildcard route in the following excerpt showing the complete `appRoutes` for this milestone.

A redirect route requires a `pathMatch` property to tell the router how to match a URL to the path of a route. The router throws an error if you don't. In this app, the router should select the route to the `HeroListComponent` only when the *entire URL* matches `''` , so set the `pathMatch` value to `'full'` .

Technically, `pathMatch = 'full'` results in a route hit when the *remaining*, unmatched segments of the URL match `''`. In this example, the redirect is in a top level route so the *remaining* URL and the *entire* URL are the same thing. The other possible `pathMatch` value is `'prefix'` which tells the router to match the redirect route when the *remaining* URL ***begins*** with the redirect route's _prefix_ path. Don't do that here. If the `pathMatch` value were `'prefix'`, _every_ URL would match `''`. Try setting it to `'prefix'` then click the `Go to sidekicks` button. Remember that's a bad URL and you should see the "Page not found" page. Instead, you're still on the "Heroes" page. Enter a bad URL in the browser address bar. You're instantly re-routed to `/heroes`. _Every_ URL, good or bad, that falls through to _this_ route definition will be a match. The default route should redirect to the `HeroListComponent` _only_ when the _entire_ url is `''`. Remember to restore the redirect to `pathMatch = 'full'`. Learn more in Victor Savkin's [post on redirects](http://victorsavkin.com/post/146722301646/angular-router-empty-paths-componentless-routes).

Basics wrap up

You've got a very basic navigating app, one that can switch between two views when the user clicks a link.

You've learned how to do the following:

- Load the router library.
- Add a nav bar to the shell template with anchor tags, `routerLink` and `routerLinkActive` directives.
- Add a `router-outlet` to the shell template where views will be displayed.
- Configure the router module with `RouterModule.forRoot` .
- Set the router to compose HTML5 browser URLs.
- handle invalid routes with a `wildcard` route.
- navigate to the default route when the app launches with an empty path.

The rest of the starter app is mundane, with little interest from a router perspective. Here are the details for readers inclined to build the sample through to this milestone.

The starter app's structure looks like this:

router-sample
src
app
app.component.ts
app.module.ts
crisis-list.component.ts
hero-list.component.ts
not-found.component.ts
main.ts
index.html
styles.css
tsconfig.json
node_modules ...
package.json

Here are the files discussed in this milestone.

{@a routing-module}

Milestone 2: *Routing module*

In the initial route configuration, you provided a simple setup with two routes used to configure the application for routing. This is perfectly fine for simple routing. As the application grows and you make use of more `Router` features, such as guards, resolvers, and child routing, you'll naturally want to refactor the routing configuration into its own file. We recommend moving the routing information into a special-purpose module called a *Routing Module*.

The **Routing Module** has several characteristics:

- Separates routing concerns from other application concerns.
- Provides a module to replace or remove when testing the application.
- Provides a well-known location for routing service providers including guards and resolvers.
- Does **not** declare components.

{@a routing-refactor}

Refactor the routing configuration into a *routing module*

Create a file named `app-routing.module.ts` in the `/app` folder to contain the routing module.

Import the `CrisisListComponent` and the `HeroListComponent` components just like you did in the `app.module.ts`. Then move the `Router` imports and routing configuration, including `RouterModule.forRoot`, into this routing module.

Following convention, add a class name `AppRoutingModule` and export it so you can import it later in `AppModule`.

Finally, re-export the Angular `RouterModule` by adding it to the module `exports` array. By re-exporting the `RouterModule` here and importing `AppRoutingModule` in `AppModule`, the components declared in `AppModule` will have access to router directives such as `RouterLink` and `RouterOutlet`.

After these steps, the file should look like this.

Next, update the `app.module.ts` file, first importing the newly created `AppRoutingModule` from `app-routing.module.ts`, then replacing `RouterModule.forRoot` in the `imports` array with the `AppRoutingModule`.

Later in this guide you will create [multiple routing modules](#hero-routing-module) and discover that you must import those routing modules [in the correct order](#routing-module-order).

The application continues to work just the same, and you can use `AppRoutingModule` as the central place to maintain future routing configuration.

{@a why-routing-module}

Do you need a *Routing Module*?

The *Routing Module replaces* the routing configuration in the root or feature module. *Either* configure routes in the Routing Module *or* within the module itself but not in both.

The Routing Module is a design choice whose value is most obvious when the configuration is complex and includes specialized guard and resolver services. It can seem like overkill when the actual configuration is dead simple.

Some developers skip the Routing Module (for example, `AppRoutingModule`) when the configuration is simple and merge the routing configuration directly into the companion module (for example, `AppModule`).

Choose one pattern or the other and follow that pattern consistently.

Most developers should always implement a Routing Module for the sake of consistency. It keeps the code clean when configuration becomes complex. It makes testing the feature module easier. Its existence calls

attention to the fact that a module is routed. It is where developers expect to find and expand routing configuration.

{@a heroes-feature}

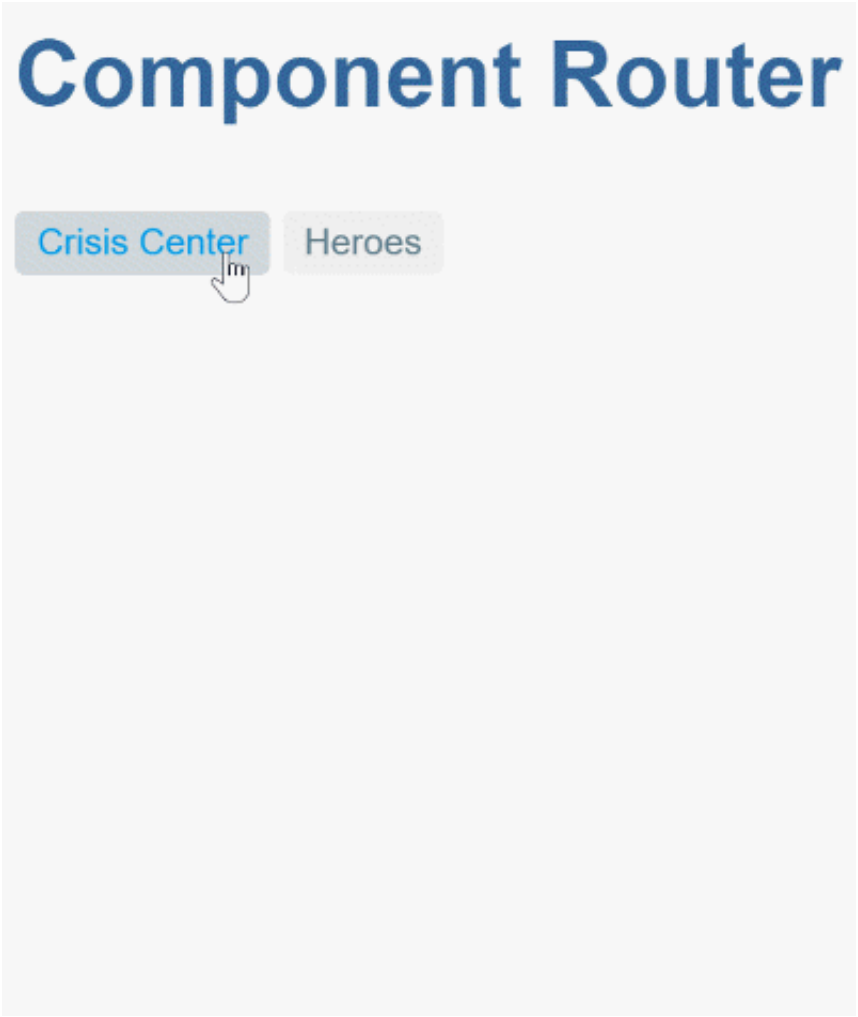
Milestone 3: Heroes feature

You've seen how to navigate using the `RouterLink` directive. Now you'll learn the following:

- Organize the app and routes into *feature areas* using modules.
- Navigate imperatively from one component to another.
- Pass required and optional information in route parameters.

This example recreates the heroes feature in the "Services" episode of the Tour of Heroes tutorial, and you'll be copying much of the code from the .

Here's how the user will experience this version of the app:



A typical application has multiple *feature areas*, each dedicated to a particular business purpose.

While you could continue to add files to the `src/app/` folder, that is unrealistic and ultimately not maintainable. Most developers prefer to put each feature area in its own folder.

You are about to break up the app into different *feature modules*, each with its own concerns. Then you'll import into the main module and navigate among them.

{@a heroes-functionality}

Add heroes functionality

Follow these steps:

- Create the `src/app/heroes` folder; you'll be adding files implementing *hero management* there.
- Delete the placeholder `hero-list.component.ts` that's in the `app` folder.
- Create a new `hero-list.component.ts` under `src/app/heroes`.
- Copy into it the contents of the `app.component.ts` from the "Services" tutorial.
- Make a few minor but necessary changes:
    - Delete the `selector` (routed components don't need them).
    - Delete the `<h1>`.
    - Relabel the `<h2>` to `<h2>HEROES</h2>`.
    - Delete the `<hero-detail>` at the bottom of the template.
    - Rename the `AppComponent` class to `HeroListComponent`.
- Copy the `hero-detail.component.ts` and the `hero.service.ts` files into the `heroes` subfolder.
- Create a (pre-routing) `heroes.module.ts` in the heroes folder that looks like this:

When you're done, you'll have these *hero management* files:

src/app/heroes
hero-detail.component.ts
hero-list.component.ts
hero.service.ts
heroes.module.ts

{@a hero-routing-requirements}

Hero feature routing requirements

The heroes feature has two interacting components, the hero list and the hero detail. The list view is self-sufficient; you navigate to it, it gets a list of heroes and displays them.

The detail view is different. It displays a particular hero. It can't know which hero to show on its own. That information must come from outside.

When the user selects a hero from the list, the app should navigate to the detail view and show that hero. You tell the detail view which hero to display by including the selected hero's id in the route URL.

{@a hero-routing-module}

Hero feature route configuration

Create a new `heroes-routing.module.ts` in the `heroes` folder using the same techniques you learned while creating the `AppRoutingModule`.

Put the routing module file in the same folder as its companion module file. Here both `heroes-routing.module.ts` and `heroes.module.ts` are in the same `src/app/heroes` folder. Consider giving each feature module its own route configuration file. It may seem like overkill early when the feature routes are simple. But routes have a tendency to grow more complex and consistency in patterns pays off over time.

Import the hero components from their new locations in the `src/app/heroes/` folder, define the two hero routes, and export the `HeroRoutingModule` class.

Now that you have routes for the `Heroes` module, register them with the `Router` via the `RouterModule` *almost* as you did in the `AppRoutingModule`.

There is a small but critical difference. In the `AppRoutingModule`, you used the static **`RouterModule.forRoot`** method to register the routes and application level service providers. In a feature module you use the static **`forChild`** method.

Only call `RouterModule.forRoot` in the root `AppRoutingModule` (or the `AppModule` if that's where you register top level application routes). In any other module, you must call the **`RouterModule.forChild`** method to register additional routes.

{@a adding-routing-module}

Add the routing module to the *HeroesModule*

Add the `HeroRoutingModule` to the `HeroModule` just as you added `AppRoutingModule` to the `AppModule`.

Open `heroes.module.ts`. Import the `HeroRoutingModule` token from `heroes-routing.module.ts` and add it to the `imports` array of the `HeroesModule`. The finished `HeroesModule` looks like this:

{@a remove-duplicate-hero-routes}

Remove duplicate hero routes

The hero routes are currently defined in *two* places: in the `HeroesRoutingModule`, by way of the `HeroesModule`, and in the `AppRoutingModule`.

Routes provided by feature modules are combined together into their imported module's routes by the router. This allows you to continue defining the feature module routes without modifying the main route configuration.

But you don't want to define the same routes twice. Remove the `HeroListComponent` import and the `/heroes` route from the `app-routing.module.ts`.

Leave the default and the wildcard routes! These are concerns at the top level of the application itself.

{@a merge-hero-routes}

Import hero module into AppModule

The heroes feature module is ready, but the application doesn't know about the `HeroesModule` yet. Open `app.module.ts` and revise it as follows.

Import the `HeroesModule` and add it to the `imports` array in the `@NgModule` metadata of the `AppModule`.

Remove the `HeroListComponent` from the `AppModule`'s `declarations` because it's now provided by the `HeroesModule`. This is important. There can be only *one* owner for a declared component. In this case, the `Heroes` module is the owner of the `Heroes` components and is making them available to components in the `AppModule` via the `HeroesModule`.

As a result, the `AppModule` no longer has specific knowledge of the hero feature, its components, or its route details. You can evolve the hero feature with more components and different routes. That's a key benefit of creating a separate module for each feature area.

After these steps, the `AppModule` should look like this:

{@a routing-module-order}

Module import order matters

Look at the module `imports` array. Notice that the `AppRoutingModule` is *last*. Most importantly, it comes *after* the `HeroesModule`.

The order of route configuration matters. The router accepts the first route that matches a navigation request path.

When all routes were in one `AppRoutingModule`, you put the default and wildcard routes last, after the `/heroes` route, so that the router had a chance to match a URL to the `/heroes` route *before* hitting the wildcard route and navigating to "Page not found".

The routes are no longer in one file. They are distributed across two modules, `AppRoutingModule` and `HeroesRoutingModule`.

Each routing module augments the route configuration *in the order of import*. If you list `AppRoutingModule` first, the wildcard route will be registered *before* the hero routes. The wildcard route — which matches *every* URL — will intercept the attempt to navigate to a hero route.

Reverse the routing modules and see for yourself that a click of the heroes link results in "Page not found". Learn about inspecting the runtime router configuration [below](#inspect-config "Inspect the router config").

{@a route-def-with-parameter}

Route definition with a parameter

Return to the `HeroesRoutingModule` and look at the route definitions again. The route to `HeroDetailComponent` has a twist.

Notice the `:id` token in the path. That creates a slot in the path for a **Route Parameter**. In this case, the router will insert the `id` of a hero into that slot.

If you tell the router to navigate to the detail component and display "Magneta", you expect a hero id to appear in the browser URL like this:

localhost:3000/hero/15

If a user enters that URL into the browser address bar, the router should recognize the pattern and go to the same "Magneta" detail view.

Route parameter: Required or optional?
Embedding the route parameter token, `:id`, in the route definition path is a good choice for this scenario because the `id` is *required* by the `HeroDetailComponent` and because the value `15` in the path clearly

distinguishes the route to "Magneta" from a route for some other hero.

{@a route-parameters}

Setting the route parameters in the list view

After navigating to the `HeroDetailComponent`, you expect to see the details of the selected hero. You need *two* pieces of information: the routing path to the component and the hero's `id`.

Accordingly, the *link parameters array* has *two* items: the routing *path* and a *route parameter* that specifies the `id` of the selected hero.

The router composes the destination URL from the array like this: `localhost:3000/hero/15`.

How does the target `HeroDetailComponent` learn about that `id`? Don't analyze the URL. Let the router do it. The router extracts the route parameter (`id:15`) from the URL and supplies it to the `HeroDetailComponent` via the `ActivatedRoute` service.

{@a activated-route}

Activated Route in action

Import the `Router`, `ActivatedRoute`, and `ParamMap` tokens from the router package.

Import the `switchMap` operator because you need it later to process the `Observable` route parameters.

{@a hero-detail-ctor}

As usual, you write a constructor that asks Angular to inject services that the component requires and reference them as private variables.

Later, in the `ngOnInit` method, you use the `ActivatedRoute` service to retrieve the parameters for the route, pull the hero `id` from the parameters and retrieve the hero to display.

The `paramMap` processing is a bit tricky. When the map changes, you `get()` the `id` parameter from the changed parameters.

Then you tell the `HeroService` to fetch the hero with that `id` and return the result of the `HeroService` request.

You might think to use the RxJS `map` operator. But the `HeroService` returns an `Observable<Hero>`. So you flatten the `Observable` with the `switchMap` operator instead.

The `switchMap` operator also cancels previous in-flight requests. If the user re-navigates to this route with a new `id` while the `HeroService` is still retrieving the old `id`, `switchMap` discards that old request and returns the hero for the new `id`.

The observable `Subscription` will be handled by the `AsyncPipe` and the component's `hero` property will be (re)set with the retrieved hero.

ParamMap API

The `ParamMap` API is inspired by the URLSearchParams interface. It provides methods to handle parameter access for both route parameters (`paramMap`) and query parameters (`queryParamMap`).

Member	Description
`has(name)`	Returns `true` if the parameter name is in the map of parameters.
`get(name)`	Returns the parameter name value (a `string`) if present, or `null` if the parameter name is not in the map. Returns the _first_ element if the parameter value is actually an array of values.
`getAll(name)`	Returns a `string array` of the parameter name value if found, or an empty `array` if the parameter name value is not in the map. Use `getAll` when a single parameter could have multiple values.
`keys`	Returns a `string array` of all parameter names in the map.

{@a reuse}

Observable *paramMap* and component reuse

In this example, you retrieve the route parameter map from an `Observable` . That implies that the route parameter map can change during the lifetime of this component.

They might. By default, the router re-uses a component instance when it re-navigates to the same component type without visiting a different component first. The route parameters could change each time.

Suppose a parent component navigation bar had "forward" and "back" buttons that scrolled through the list of heroes. Each click navigated imperatively to the `HeroDetailComponent` with the next or previous `id` .

You don't want the router to remove the current `HeroDetailComponent` instance from the DOM only to re-create it for the next `id` . That could be visibly jarring. Better to simply re-use the same component instance and update the parameter.

Unfortunately, `ngOnInit` is only called once per component instantiation. You need a way to detect when the route parameters change from *within the same instance*. The observable `paramMap` property handles that beautifully.

When subscribing to an observable in a component, you almost always arrange to unsubscribe when the component is destroyed. There are a few exceptional observables where this is not necessary. The `ActivatedRoute` observables are among the exceptions. The `ActivatedRoute` and its observables are insulated from the `Router` itself. The `Router` destroys a routed component when it is no longer needed and the injected `ActivatedRoute` dies with it. Feel free to unsubscribe anyway. It is harmless and never a bad practice.

{@a snapshot}

***Snapshot*: the *no-observable* alternative**

This application won't re-use the `HeroDetailComponent`. The user always returns to the hero list to select another hero to view. There's no way to navigate from one hero detail to another hero detail without visiting the list component in between. Therefore, the router creates a new `HeroDetailComponent` instance every time.

When you know for certain that a `HeroDetailComponent` instance will *never, never, ever* be re-used, you can simplify the code with the *snapshot*.

The `route.snapshot` provides the initial value of the route parameter map. You can access the parameters directly without subscribing or adding observable operators. It's much simpler to write and read:

Remember: you only get the _initial_ value of the parameter map with this technique. Stick with the observable `paramMap` approach if there's even a chance that the router could re-use the component. This sample stays with the observable `paramMap` strategy just in case.

{@a nav-to-list}

Navigating back to the list component

The `HeroDetailComponent` has a "Back" button wired to its `gotoHeroes` method that navigates imperatively back to the `HeroListComponent`.

The router `navigate` method takes the same one-item *link parameters array* that you can bind to a `[routerLink]` directive. It holds the *path to the* `HeroListComponent`:

{@a optional-route-parameters}

Route Parameters: Required or optional?

Use *route parameters* to specify a *required* parameter value *within* the route URL as you do when navigating to the `HeroDetailComponent` in order to view the hero with *id* 15:

localhost:3000/hero/15

You can also add *optional* information to a route request. For example, when returning to the heroes list from the hero detail view, it would be nice if the viewed hero was preselected in the list.

You'll implement this feature in a moment by including the viewed hero's `id` in the URL as an optional parameter when returning from the `HeroDetailComponent`.

Optional information takes other forms. Search criteria are often loosely structured, e.g., `name='wind*'`. Multiple values are common— `after='12/31/2015' & before='1/1/2017'` —in no particular order — `before='1/1/2017' & after='12/31/2015'` — in a variety of formats — `during='currentYear'`.

These kinds of parameters don't fit easily in a URL *path*. Even if you could define a suitable URL token scheme, doing so greatly complicates the pattern matching required to translate an incoming URL to a named route.

Optional parameters are the ideal vehicle for conveying arbitrarily complex information during navigation. Optional parameters aren't involved in pattern matching and afford flexibility of expression.

The router supports navigation with optional parameters as well as required route parameters. Define *optional* parameters in a separate object *after* you define the required route parameters.

In general, prefer a *required route parameter* when the value is mandatory (for example, if necessary to distinguish one route path from another); prefer an *optional parameter* when the value is optional, complex, and/or multivariate.

{@a optionally-selecting}

Heroes list: optionally selecting a hero

When navigating to the `HeroDetailComponent` you specified the *required* `id` of the hero-to-edit in the *route parameter* and made it the second item of the *link parameters array*.

The router embedded the `id` value in the navigation URL because you had defined it as a route parameter with an `:id` placeholder token in the route `path` :

When the user clicks the back button, the `HeroDetailComponent` constructs another *link parameters array* which it uses to navigate back to the `HeroListComponent` .

This array lacks a route parameter because you had no reason to send information to the `HeroListComponent` .

Now you have a reason. You'd like to send the id of the current hero with the navigation request so that the `HeroListComponent` can highlight that hero in its list. This is a *nice-to-have* feature; the list will display perfectly well without it.

Send the `id` with an object that contains an *optional* `id` parameter. For demonstration purposes, there's an extra junk parameter (`foo`) in the object that the `HeroListComponent` should ignore. Here's the revised navigation statement:

The application still works. Clicking "back" returns to the hero list view.

Look at the browser address bar.

It should look something like this, depending on where you run it:

localhost:3000/heroes;id=15;foo=foo

The `id` value appears in the URL as (`;id=15;foo=foo`), not in the URL path. The path for the "Heroes" route doesn't have an `:id` token.

The optional route parameters are not separated by "?" and "&" as they would be in the URL query string. They are **separated by semicolons ";"** This is *matrix URL* notation — something you may not have seen before.

Matrix URL notation is an idea first introduced in a [1996 proposal](http://www.w3.org/DesignIssues/MatrixURIs.html) by the founder of the web, Tim Berners-Lee. Although matrix notation never made it into the HTML standard, it is legal and it became popular among browser routing systems as a way to isolate parameters belonging to parent and child routes. The Router is such a system and provides support for the matrix notation across browsers. The syntax may seem strange to you but users are unlikely to notice or care as long as the URL can be emailed and pasted into a browser address bar as this one can.

{@a route-parameters-activated-route}

Route parameters in the *ActivatedRoute* service

The list of heroes is unchanged. No hero row is highlighted.

The *does* highlight the selected row because it demonstrates the final state of the application which includes the steps you're *about* to cover. At the moment this guide is describing the state of affairs *prior* to those steps.

The `HeroListComponent` isn't expecting any parameters at all and wouldn't know what to do with them. You can change that.

Previously, when navigating from the `HeroListComponent` to the `HeroDetailComponent`, you subscribed to the route parameter map `Observable` and made it available to the `HeroDetailComponent` in the `ActivatedRoute` service. You injected that service in the constructor of the `HeroDetailComponent`.

This time you'll be navigating in the opposite direction, from the `HeroDetailComponent` to the `HeroListComponent`.

First you extend the router import statement to include the `ActivatedRoute` service symbol:

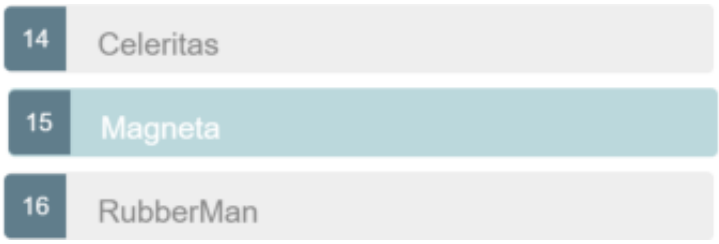
Import the `switchMap` operator to perform an operation on the `Observable` of route parameter map.

Then you inject the `ActivatedRoute` in the `HeroListComponent` constructor.

The `ActivatedRoute.paramMap` property is an `Observable` map of route parameters. The `paramMap` emits a new map of values that includes `id` when the user navigates to the component. In `ngOnInit` you subscribe to those values, set the `selectedId`, and get the heroes.

Update the template with a class binding. The binding adds the `selected` CSS class when the comparison returns `true` and removes it when `false`. Look for it within the repeated `<li>` tag as shown here:

When the user navigates from the heroes list to the "Magneta" hero and back, "Magneta" appears selected:



The optional `foo` route parameter is harmless and continues to be ignored.

{@a route-animation}

Adding animations to the routed component

The heroes feature module is almost complete, but what is a feature without some smooth transitions?

This section shows you how to add some animations to the `HeroDetailComponent`.

First import `BrowserAnimationsModule`:

Create an `animations.ts` file in the root `src/app/` folder. The contents look like this:

This file does the following:

- Imports the animation symbols that build the animation triggers, control state, and manage transitions between states.
- Exports a constant named `slideInDownAnimation` set to an animation trigger named `routeAnimation`; animated components will refer to this name.
- Specifies the *wildcard state*, `*`, that matches any animation state that the route component is in.
- Defines two *transitions*, one to ease the component in from the left of the screen as it enters the application view (`:enter`), the other to animate the component down as it leaves the application view (`:leave`).

You could create more triggers with different transitions for other route components. This trigger is sufficient for the current milestone.

Back in the `HeroDetailComponent`, import the `slideInDownAnimation` from `'./animations.ts'`. Add the `HostBinding` decorator to the imports from `@angular/core`; you'll need it in a moment.

Add an `animations` array to the `@Component` metadata's that contains the `slideInDownAnimation`.

Then add three `@HostBinding` properties to the class to set the animation and styles for the route component's element.

The `'@routeAnimation'` passed to the first `@HostBinding` matches the name of the `slideInDownAnimation` *trigger*, `routeAnimation`. Set the `routeAnimation` property to `true` because you only care about the `:enter` and `:leave` states.

The other two `@HostBinding` properties style the display and position of the component.

The `HeroDetailComponent` will ease in from the left when routed to and will slide down when navigating

away.

Applying route animations to individual components works for a simple demo, but in a real life app, it is better to animate routes based on _route paths_.

{@a milestone-3-wrap-up}

Milestone 3 wrap up

You've learned how to do the following:

- Organize the app into *feature areas*.
- Navigate imperatively from one component to another.
- Pass information along in route parameters and subscribe to them in the component.
- Import the feature area NgModule into the `AppModule`.
- Apply animations to the route component.

After these changes, the folder structure looks like this:

router-sample
src
app
heroes
hero-detail.component.ts
hero-list.component.ts
hero.service.ts
heroes.module.ts
heroes-routing.module.ts
app.component.ts
app.module.ts
app-routing.module.ts
crisis-list.component.ts
main.ts
index.html
styles.css
tsconfig.json
node_modules ...
package.json

Here are the relevant files for this version of the sample application.

{@a milestone-4}

Milestone 4: Crisis center feature

It's time to add real features to the app's current placeholder crisis center.

Begin by imitating the heroes feature:

- Delete the placeholder crisis center file.
- Create an `app/crisis-center` folder.
- Copy the files from `app/heroes` into the new crisis center folder.
- In the new files, change every mention of "hero" to "crisis", and "heroes" to "crises".

You'll turn the `CrisisService` into a purveyor of mock crises instead of mock heroes:

The resulting crisis center is a foundation for introducing a new concept—**child routing**. You can leave *Heroes* in its current state as a contrast with the *Crisis Center* and decide later if the differences are worthwhile.

In keeping with the *Separation of Concerns* principle, changes to the *Crisis Center* won't affect the `AppModule` or any other feature's component.

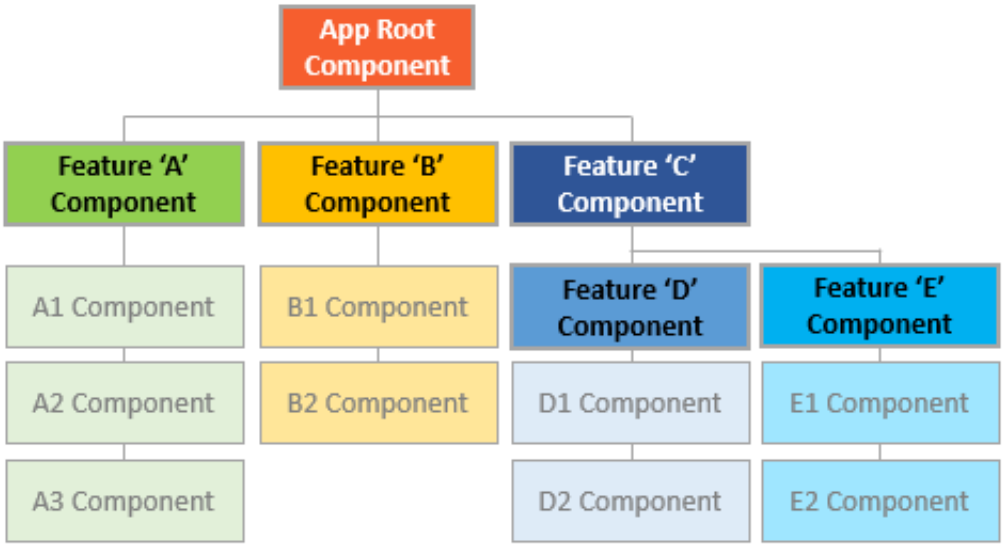
{@a crisis-child-routes}

A crisis center with child routes

This section shows you how to organize the crisis center to conform to the following recommended pattern for Angular applications:

- Each feature area resides in its own folder.
- Each feature has its own Angular feature module.
- Each area has its own area root component.
- Each area root component has its own router outlet and child routes.
- Feature area routes rarely (if ever) cross with routes of other features.

If your app had many feature areas, the app component trees might look like this:



{@a child-routing-component}

Child routing component

Add the following `crisis-center.component.ts` to the `crisis-center` folder:

The `CrisisCenterComponent` has the following in common with the `AppComponent`:

- It is the *root* of the crisis center area, just as `AppComponent` is the root of the entire application.
- It is a *shell* for the crisis management feature area, just as the `AppComponent` is a shell to manage the high-level workflow.

Like most shells, the `CrisisCenterComponent` class is very simple, simpler even than `AppComponent`: it has no business logic, and its template has no links, just a title and `<router-outlet>` for the crisis center child views.

Unlike `AppComponent`, and most other components, it *lacks a selector*. It doesn't *need* one since you don't *embed* this component in a parent template, instead you use the router to *navigate* to it.

{@a child-route-config}

Child route configuration

As a host page for the "Crisis Center" feature, add the following `crisis-center-home.component.ts` to the `crisis-center` folder.

Create a `crisis-center-routing.module.ts` file as you did the `heroes-routing.module.ts` file. This time, you define **child routes** *within* the parent `crisis-center` route.

Notice that the parent `crisis-center` route has a `children` property with a single route containing the `CrisisListComponent`. The `CrisisListComponent` route also has a `children` array with two routes.

These two routes navigate to the crisis center child components, `CrisisCenterHomeComponent` and `CrisisDetailComponent`, respectively.

There are *important differences* in the way the router treats these *child routes*.

The router displays the components of these routes in the `RouterOutlet` of the `CrisisCenterComponent`, not in the `RouterOutlet` of the `AppComponent` shell.

The `CrisisListComponent` contains the crisis list and a `RouterOutlet` to display the `Crisis Center Home` and `Crisis Detail` route components.

The `Crisis Detail` route is a child of the `Crisis List`. Since the router reuses components by default, the `Crisis Detail` component will be re-used as you select different crises. In contrast, back in the `Hero Detail` route, the component was recreated each time you selected a different hero.

At the top level, paths that begin with `/` refer to the root of the application. But child routes *extend* the path of the parent route. With each step down the route tree, you add a slash followed by the route path, unless the path is *empty*.

Apply that logic to navigation within the crisis center for which the parent path is `/crisis-center`.

- To navigate to the `CrisisCenterHomeComponent`, the full URL is `/crisis-center` (`/crisis-center` + `''` + `''`).
- To navigate to the `CrisisDetailComponent` for a crisis with `id=2`, the full URL is `/crisis-center/2` (`/crisis-center` + `''` + `'/2'`).

The absolute URL for the latter example, including the `localhost` origin, is

localhost:3000/crisis-center/2

Here's the complete `crisis-center-routing.module.ts` file with its imports.

{@a import-crisis-module}

Import crisis center module into the *AppModule* routes

As with the `HeroesModule`, you must add the `CrisisCenterModule` to the `imports` array of the `AppModule` *before* the `AppRoutingModule`:

Remove the initial crisis center route from the `app-routing.module.ts`. The feature routes are now provided by the `HeroesModule` and the `CrisisCenter` modules.

The `app-routing.module.ts` file retains the top-level application routes such as the default and wildcard routes.

{@a relative-navigation}

Relative navigation

While building out the crisis center feature, you navigated to the crisis detail route using an **absolute path** that begins with a *slash*.

The router matches such *absolute* paths to routes starting from the top of the route configuration.

You could continue to use absolute paths like this to navigate inside the *Crisis Center* feature, but that pins the links to the parent routing structure. If you changed the parent `/crisis-center` path, you would have to change the link parameters array.

You can free the links from this dependency by defining paths that are **relative** to the current URL segment. Navigation *within* the feature area remains intact even if you change the parent route path to the feature.

Here's an example:

The router supports directory-like syntax in a _link parameters list_ to help guide route name lookup: `./` or `no leading slash` is relative to the current level. `../` to go up one level in the route path. You can combine relative navigation syntax with an ancestor path. If you must navigate to a sibling route, you could use the `../` convention to go up one level, then over and down the sibling route path.

To navigate a relative path with the `Router.navigate` method, you must supply the `ActivatedRoute` to give the router knowledge of where you are in the current route tree.

After the *link parameters array*, add an object with a `relativeTo` property set to the `ActivatedRoute`. The router then calculates the target URL based on the active route's location.

Always specify the complete _absolute_ path when calling router's `navigateByUrl` method.

{@a nav-to-crisis}

Navigate to crisis list with a relative URL

You've already injected the `ActivatedRoute` that you need to compose the relative navigation path.

When using a `RouterLink` to navigate instead of the `Router` service, you'd use the *same* link parameters array, but you wouldn't provide the object with the `relativeTo` property. The `ActivatedRoute` is implicit in a `RouterLink` directive.

Update the `gotoCrises` method of the `CrisisDetailComponent` to navigate back to the *Crisis Center* list using relative path navigation.

Notice that the path goes up a level using the `../` syntax. If the current crisis `id` is `3`, the resulting path back to the crisis list is `/crisis-center/;id=3;foo=foo`.

{@a named-outlets}

Displaying multiple routes in named outlets

You decide to give users a way to contact the crisis center. When a user clicks a "Contact" button, you want to display a message in a popup view.

The popup should stay open, even when switching between pages in the application, until the user closes it by sending the message or canceling. Clearly you can't put the popup in the same outlet as the other pages.

Until now, you've defined a single outlet and you've nested child routes under that outlet to group routes together. The router only supports one primary *unnamed* outlet per template.

A template can also have any number of *named* outlets. Each named outlet has its own set of routes with their own components. Multiple outlets can be displaying different content, determined by different routes, all at the same time.

Add an outlet named "popup" in the `AppComponent`, directly below the unnamed outlet.

That's where a popup will go, once you learn how to route a popup component to it.

{@a secondary-routes}

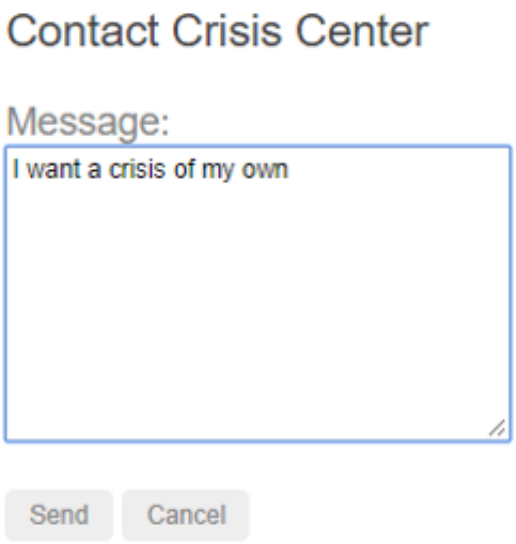
Secondary routes

Named outlets are the targets of *secondary routes*.

Secondary routes look like primary routes and you configure them the same way. They differ in a few key respects.

- They are independent of each other.
- They work in combination with other routes.
- They are displayed in named outlets.

Create a new component named `ComposeMessageComponent` in `src/app/compose-message.component.ts`. It displays a simple form with a header, an input box for the message, and two buttons, "Send" and "Cancel".



Here's the component and its template:

It looks about the same as any other component you've seen in this guide. There are two noteworthy differences.

Note that the `send()` method simulates latency by waiting a second before "sending" the message and closing the popup.

The `closePopup()` method closes the popup view by navigating to the popup outlet with a `null`. That's a peculiarity covered below.

As with other application components, you add the `ComposeMessageComponent` to the `declarations` of an `NgModule`. Do so in the `AppModule`.

{@a add-secondary-route}

Add a secondary route

Open the `AppRoutingModule` and add a new `compose` route to the `appRoutes`.

The `path` and `component` properties should be familiar. There's a new property, `outlet`, set to `'popup'`. This route now targets the popup outlet and the `ComposeMessageComponent` will display there.

The user needs a way to open the popup. Open the `AppComponent` and add a "Contact" link.

Although the `compose` route is pinned to the "popup" outlet, that's not sufficient for wiring the route to a `RouterLink` directive. You have to specify the named outlet in a *link parameters array* and bind it to the `RouterLink` with a property binding.

The *link parameters array* contains an object with a single `outlets` property whose value is another object keyed by one (or more) outlet names. In this case there is only the "popup" outlet property and its value is another *link parameters array* that specifies the `compose` route.

You are in effect saying, *when the user clicks this link, display the component associated with the `compose` route in the `popup` outlet.*

This `outlets` object within an outer object was completely unnecessary when there was only one route and one _unnamed_ outlet to think about. The router assumed that your route specification targeted the _unnamed_ primary outlet and created these objects for you. Routing to a named outlet has revealed a previously hidden router truth: you can target multiple outlets with multiple routes in the same `RouterLink` directive. You're not actually doing that here. But to target a named outlet, you must use the richer, more verbose syntax.

{@a secondary-route-navigation}

Secondary route navigation: merging routes during navigation

Navigate to the *Crisis Center* and click "Contact". you should see something like the following URL in the browser address bar.

http://.../crisis-center(popup:compose)

The interesting part of the URL follows the `...`:

- The `crisis-center` is the primary navigation.
- Parentheses surround the secondary route.
- The secondary route consists of an outlet name (`popup`), a `colon` separator, and the secondary route path (`compose`).

Click the *Heroes* link and look at the URL again.

http://.../heroes(popup:compose)

The primary navigation part has changed; the secondary route is the same.

The router is keeping track of two separate branches in a navigation tree and generating a representation of that tree in the URL.

You can add many more outlets and routes, at the top level and in nested levels, creating a navigation tree with many branches. The router will generate the URL to go with it.

You can tell the router to navigate an entire tree at once by filling out the `outlets` object mentioned above. Then pass that object inside a *link parameters array* to the `router.navigate` method.

Experiment with these possibilities at your leisure.

{@a clear-secondary-routes}

Clearing secondary routes

As you've learned, a component in an outlet persists until you navigate away to a new component. Secondary outlets are no different in this regard.

Each secondary outlet has its own navigation, independent of the navigation driving the primary outlet. Changing a current route that displays in the primary outlet has no effect on the popup outlet. That's why the popup stays visible as you navigate among the crises and heroes.

Clicking the "send" or "cancel" buttons *does* clear the popup view. To see how, look at the `closePopup()` method again:

It navigates imperatively with the `Router.navigate()` method, passing in a link parameters array.

Like the array bound to the *Contact* `RouterLink` in the `AppComponent`, this one includes an object with an `outlets` property. The `outlets` property value is another object with outlet names for keys. The only named outlet is `'popup'`.

This time, the value of `'popup'` is `null`. That's not a route, but it is a legitimate value. Setting the popup `RouterOutlet` to `null` clears the outlet and removes the secondary popup route from the current URL.

{@a guards}

Milestone 5: Route guards

At the moment, *any* user can navigate *anywhere* in the application *anytime*. That's not always the right thing to do.

- Perhaps the user is not authorized to navigate to the target component.
- Maybe the user must login (*authenticate*) first.
- Maybe you should fetch some data before you display the target component.
- You might want to save pending changes before leaving a component.

- You might ask the user if it's OK to discard pending changes rather than save them.

You can add *guards* to the route configuration to handle these scenarios.

A guard's return value controls the router's behavior:

- If it returns `true`, the navigation process continues.
- If it returns `false`, the navigation process stops and the user stays put.

The guard can also tell the router to navigate elsewhere, effectively canceling the current navigation.

The guard *might* return its boolean answer synchronously. But in many cases, the guard can't produce an answer synchronously. The guard could ask the user a question, save changes to the server, or fetch fresh data. These are all asynchronous operations.

Accordingly, a routing guard can return an `Observable<boolean>` or a `Promise<boolean>` and the router will wait for the observable to resolve to `true` or `false`.

The router supports multiple guard interfaces:

- `CanActivate` to mediate navigation *to* a route.
- `CanActivateChild` to mediate navigation *to* a child route.
- `CanDeactivate` to mediate navigation *away* from the current route.
- `Resolve` to perform route data retrieval *before* route activation.
- `CanLoad` to mediate navigation *to* a feature module loaded *asynchronously*.

You can have multiple guards at every level of a routing hierarchy. The router checks the `CanDeactivate` and `CanActivateChild` guards first, from the deepest child route to the top. Then it checks the `CanActivate` guards from the top down to the deepest child route. If the feature module is loaded asynchronously, the `CanLoad` guard is checked before the module is loaded. If *any* guard returns false, pending guards that have not completed will be canceled, and the entire navigation is canceled.

There are several examples over the next few sections.

{@a can-activate-guard}

CanActivate: requiring authentication

Applications often restrict access to a feature area based on who the user is. You could permit access only to authenticated users or to users with a specific role. You might block or limit access until the user's account is

activated.

The `CanActivate` guard is the tool to manage these navigation business rules.

Add an admin feature module

In this next section, you'll extend the crisis center with some new *administrative* features. Those features aren't defined yet. But you can start by adding a new feature module named `AdminModule` .

Create an `admin` folder with a feature module file, a routing configuration file, and supporting components.

The admin feature file structure looks like this:

src/app/admin
admin-dashboard.component.ts
admin.component.ts
admin.module.ts
admin-routing.module.ts
manage-crises.component.ts
manage-heroes.component.ts

The admin feature module contains the `AdminComponent` used for routing within the feature module, a dashboard route and two unfinished components to manage crises and heroes.

Since the admin dashboard `RouterLink` is an empty path route in the `AdminComponent`, it is considered a match to any route within the admin feature area. You only want the `Dashboard` link to be active when the user visits that route. Adding an additional binding to the `Dashboard` routerLink, `[routerLinkActiveOptions]="{ exact: true }"`, marks the `./` link as active when the user navigates to the `/admin` URL and not when navigating to any of the child routes.

The initial admin routing configuration:

{@a component-less-route}

Component-less route: grouping routes without a component

Looking at the child route under the `AdminComponent` , there is a `path` and a `children` property but it's not using a `component` . You haven't made a mistake in the configuration. You've defined a *component-less* route.

The goal is to group the `Crisis Center` management routes under the `admin` path. You don't need a component to do it. A *component-less* route makes it easier to guard child routes.

Next, import the `AdminModule` into `app.module.ts` and add it to the `imports` array to register the admin routes.

Add an "Admin" link to the `AppComponent` shell so that users can get to this feature.

{@a guard-admin-feature}

Guard the admin feature

Currently every route within the *Crisis Center* is open to everyone. The new *admin* feature should be accessible only to authenticated users.

You could hide the link until the user logs in. But that's tricky and difficult to maintain.

Instead you'll write a `canActivate()` guard method to redirect anonymous users to the login page when they try to enter the admin area.

This is a general purpose guard—you can imagine other features that require authenticated users—so you create an `auth-guard.service.ts` in the application root folder.

At the moment you're interested in seeing how guards work so the first version does nothing useful. It simply logs to console and `returns` true immediately, allowing navigation to proceed:

Next, open `admin-routing.module.ts`, import the `AuthGuard` class, and update the admin route with a `canActivate` guard property that references it:

The admin feature is now protected by the guard, albeit protected poorly.

{@a teach-auth}

Teach *AuthGuard* to authenticate

Make the `AuthGuard` at least pretend to authenticate.

The `AuthGuard` should call an application service that can login a user and retain information about the current user. Here's a demo `AuthService`:

Although it doesn't actually log in, it has what you need for this discussion. It has an `isLoggedIn` flag to tell you whether the user is authenticated. Its `login` method simulates an API call to an external service by returning an Observable that resolves successfully after a short pause. The `redirectUrl` property will store the attempted URL so you can navigate to it after authenticating.

Revise the `AuthGuard` to call it.

Notice that you *inject* the `AuthService` and the `Router` in the constructor. You haven't provided the `AuthService` yet but it's good to know that you can inject helpful services into routing guards.

This guard returns a synchronous boolean result. If the user is logged in, it returns true and the navigation continues.

The `ActivatedRouteSnapshot` contains the *future* route that will be activated and the `RouterStateSnapshot` contains the *future* `RouterState` of the application, should you pass through the guard check.

If the user is not logged in, you store the attempted URL the user came from using the `RouterStateSnapshot.url` and tell the router to navigate to a login page—a page you haven't created yet. This secondary navigation automatically cancels the current navigation; `checkLogin()` returns `false` just to be clear about that.

{@a add-login-component}

Add the *LoginComponent*

You need a `LoginComponent` for the user to log in to the app. After logging in, you'll redirect to the stored URL if available, or use the default URL. There is nothing new about this component or the way you wire it into the router configuration.

Register a `/login` route in the `login-routing.module.ts` and add the necessary providers to the `providers` array. In `app.module.ts`, import the `LoginComponent` and add it to the `AppModule` `declarations`. Import and add the `LoginRoutingModule` to the `AppModule` imports as well.

Guards and the service providers they require _must_ be provided at the module-level. This allows the Router access to retrieve these services from the `Injector` during the navigation process. The same rule applies for feature modules loaded [asynchronously](#asynchronous-routing).

{@a can-activate-child-guard}

CanActivateChild: guarding child routes

You can also protect child routes with the `CanActivateChild` guard. The `CanActivateChild` guard is similar to the `CanActivate` guard. The key difference is that it runs *before* any child route is activated.

You protected the admin feature module from unauthorized access. You should also protect child routes *within* the feature module.

Extend the `AuthGuard` to protect when navigating between the `admin` routes. Open

`auth-guard.service.ts` and add the `CanActivateChild` interface to the imported tokens from the router package.

Next, implement the `canActivateChild()` method which takes the same arguments as the `canActivate()` method: an `ActivatedRouteSnapshot` and `RouterStateSnapshot`. The `canActivateChild()` method can return an `Observable<boolean>` or `Promise<boolean>` for async checks and a `boolean` for sync checks. This one returns a `boolean`:

Add the same `AuthGuard` to the `component-less` admin route to protect all other child routes at one time instead of adding the `AuthGuard` to each route individually.

{@a can-deactivate-guard}

CanDeactivate: handling unsaved changes

Back in the "Heroes" workflow, the app accepts every change to a hero immediately without hesitation or validation.

In the real world, you might have to accumulate the users changes. You might have to validate across fields. You might have to validate on the server. You might have to hold changes in a pending state until the user confirms them *as a group* or cancels and reverts all changes.

What do you do about unapproved, unsaved changes when the user navigates away? You can't just leave and risk losing the user's changes; that would be a terrible experience.

It's better to pause and let the user decide what to do. If the user cancels, you'll stay put and allow more changes. If the user approves, the app can save.

You still might delay navigation until the save succeeds. If you let the user move to the next screen immediately and the save were to fail (perhaps the data are ruled invalid), you would lose the context of the error.

You can't block while waiting for the server—that's not possible in a browser. You need to stop the navigation while you wait, asynchronously, for the server to return with its answer.

You need the `CanDeactivate` guard.

{@a cancel-save}

Cancel and save

The sample application doesn't talk to a server. Fortunately, you have another way to demonstrate an asynchronous router hook.

Users update crisis information in the `CrisisDetailComponent`. Unlike the `HeroDetailComponent`, the user changes do not update the crisis entity immediately. Instead, the app updates the entity when the user presses the *Save* button and discards the changes when the user presses the *Cancel* button.

Both buttons navigate back to the crisis list after save or cancel.

What if the user tries to navigate away without saving or canceling? The user could push the browser back button or click the heroes link. Both actions trigger a navigation. Should the app save or cancel automatically?

This demo does neither. Instead, it asks the user to make that choice explicitly in a confirmation dialog box that *waits asynchronously for the user's answer*.

You could wait for the user's answer with synchronous, blocking code. The app will be more responsive—and can do other work—by waiting for the user's answer asynchronously. Waiting for the user asynchronously is like waiting for the server asynchronously.

The `DialogService`, provided in the `AppModule` for app-wide use, does the asking.

It returns an `Observable` that *resolves* when the user eventually decides what to do: either to discard changes and navigate away (`true`) or to preserve the pending changes and stay in the crisis editor (`false`).

{@a CanDeactivate}

Create a *guard* that checks for the presence of a `canDeactivate()` method in a component—any component. The `CrisisDetailComponent` will have this method. But the guard doesn't have to know that. The guard shouldn't know the details of any component's deactivation method. It need only detect that the component has a `canDeactivate()` method and call it. This approach makes the guard reusable.

Alternatively, you could make a component-specific `CanDeactivate` guard for the `CrisisDetailComponent`. The `canDeactivate()` method provides you with the current instance of the `component`, the current `ActivatedRoute`, and `RouterStateSnapshot` in case you needed to access some external information. This would be useful if you only wanted to use this guard for this component and needed to get the component's properties or confirm whether the router should allow navigation away from it.

Looking back at the `CrisisDetailComponent`, it implements the confirmation workflow for unsaved changes.

Notice that the `canDeactivate()` method *can* return synchronously; it returns `true` immediately if there is no crisis or there are no pending changes. But it can also return a `Promise` or an `Observable` and the router will wait for that to resolve to truthy (navigate) or falsy (stay put).

Add the `Guard` to the crisis detail route in `crisis-center-routing.module.ts` using the `canDeactivate` array property.

Add the `Guard` to the main `AppRoutingModule` `providers` array so the `Router` can inject it during the navigation process.

Now you have given the user a safeguard against unsaved changes. {@a Resolve}

{@a resolve-guard}

Resolve: pre-fetching component data

In the `Hero Detail` and `Crisis Detail`, the app waited until the route was activated to fetch the respective hero or crisis.

This worked well, but there's a better way. If you were using a real world API, there might be some delay before the data to display is returned from the server. You don't want to display a blank component while waiting for the data.

It's preferable to pre-fetch data from the server so it's ready the moment the route is activated. This also allows you to handle errors before routing to the component. There's no point in navigating to a crisis detail for an `id` that doesn't have a record. It'd be better to send the user back to the `Crisis List` that shows only valid crisis centers.

In summary, you want to delay rendering the routed component until all necessary data have been fetched.

You need a *resolver*.

{@a fetch-before-navigating}

Fetch data before navigating

At the moment, the `CrisisDetailComponent` retrieves the selected crisis. If the crisis is not found, it navigates back to the crisis list view.

The experience might be better if all of this were handled first, before the route is activated. A `CrisisDetailResolver` service could retrieve a `Crisis` or navigate away if the `Crisis` does not exist *before* activating the route and creating the `CrisisDetailComponent`.

Create the `crisis-detail-resolver.service.ts` file within the `Crisis Center` feature area.

Take the relevant parts of the crisis retrieval logic in `CrisisDetailComponent.ngOnInit` and move them into the `CrisisDetailResolver`. Import the `Crisis` model, `CrisisService`, and the

`Router` so you can navigate elsewhere if you can't fetch the crisis.

Be explicit. Implement the `Resolve` interface with a type of `Crisis`.

Inject the `CrisisService` and `Router` and implement the `resolve()` method. That method could return a `Promise`, an `Observable`, or a synchronous return value.

The `CrisisService.getCrisis` method returns an Observable. Return that observable to prevent the route from loading until the data is fetched. The `Router` guards require an Observable to `complete`, meaning it has emitted all of its values. You use the `take` operator with an argument of `1` to ensure that the Observable completes after retrieving the first value from the Observable returned by the `getCrisis` method. If it doesn't return a valid `Crisis`, navigate the user back to the `CrisisListComponent`, canceling the previous in-flight navigation to the `CrisisDetailComponent`.

Import this resolver in the `crisis-center-routing.module.ts` and add a `resolve` object to the `CrisisDetailComponent` route configuration.

Remember to add the `CrisisDetailResolver` service to the `CrisisCenterRoutingModule`'s `providers` array.

The `CrisisDetailComponent` should no longer fetch the crisis. Update the `CrisisDetailComponent` to get the crisis from the `ActivatedRoute.data.crisis` property instead; that's where you said it should be when you re-configured the route. It will be there when the `CrisisDetailComponent` ask for it.

Three critical points

1. The router's `Resolve` interface is optional. The `CrisisDetailResolver` doesn't inherit from a base class. The router looks for that method and calls it if found.

2. Rely on the router to call the resolver. Don't worry about all the ways that the user could navigate away. That's the router's job. Write this class and let the router take it from there.

3. The Observable provided to the Router *must* complete. If the Observable does not complete, the navigation will not continue.

The relevant *Crisis Center* code for this milestone follows.

{@a query-parameters}

{@a fragment}

Query parameters and fragments

In the route parameters example, you only dealt with parameters specific to the route, but what if you wanted optional parameters available to all routes? This is where query parameters come into play.

Fragments refer to certain elements on the page identified with an `id` attribute.

Update the `AuthGuard` to provide a `session_id` query that will remain after navigating to another route.

Add an `anchor` element so you can jump to a certain point on the page.

Add the `NavigationExtras` object to the `router.navigate` method that navigates you to the `/login` route.

You can also preserve query parameters and fragments across navigations without having to provide them again when navigating. In the `LoginComponent`, you'll add an *object* as the second argument in the `router.navigate` function and provide the `queryParamsHandling` and `preserveFragment` to pass along the current query parameters and fragment to the next route.

The `queryParamsHandling` feature also provides a `merge` option, which will preserve and combine the current query parameters with any provided query parameters when navigating.

Since you'll be navigating to the *Admin Dashboard* route after logging in, you'll update it to handle the query parameters and fragment.

Query parameters and *fragments* are also available through the `ActivatedRoute` service. Just like *route parameters*, the query parameters and fragments are provided as an `Observable`. The updated *Crisis Admin* component feeds the `Observable` directly into the template using the `AsyncPipe`.

Now, you can click on the *Admin* button, which takes you to the *Login* page with the provided `queryParamMap` and `fragment`. After you click the login button, notice that you have been redirected to the `Admin Dashboard` page with the query parameters and fragment still intact in the address bar.

You can use these persistent bits of information for things that need to be provided across pages like authentication tokens or session ids.

The `query params` and `fragment` can also be preserved using a `RouterLink` with the `queryParamsHandling` and `preserveFragment` bindings respectively.

{@a asynchronous-routing}

Milestone 6: Asynchronous routing

As you've worked through the milestones, the application has naturally gotten larger. As you continue to build

out feature areas, the overall application size will continue to grow. At some point you'll reach a tipping point where the application takes long time to load.

How do you combat this problem? With asynchronous routing, which loads feature modules *lazily*, on request. Lazy loading has multiple benefits.

- You can load feature areas only when requested by the user.
- You can speed up load time for users that only visit certain areas of the application.
- You can continue expanding lazy loaded feature areas without increasing the size of the initial load bundle.

You're already made part way there. By organizing the application into modules— `AppModule`, `HeroesModule`, `AdminModule` and `CrisisCenterModule` —you have natural candidates for lazy loading.

Some modules, like `AppModule`, must be loaded from the start. But others can and should be lazy loaded. The `AdminModule`, for example, is needed by a few authorized users, so you should only load it when requested by the right people.

{@a lazy-loading-route-config}

Lazy Loading route configuration

Change the `admin` **path** in the `admin-routing.module.ts` from `'admin'` to an empty string, `''`, the *empty path*.

The `Router` supports *empty path* routes; use them to group routes together without adding any additional path segments to the URL. Users will still visit `/admin` and the `AdminComponent` still serves as the *Routing Component* containing child routes.

Open the `AppRoutingModule` and add a new `admin` route to its `appRoutes` array.

Give it a `loadChildren` property (not a `children` property!), set to the address of the `AdminModule`. The address is the `AdminModule` file location (relative to the app root), followed by a `#` separator, followed by the name of the exported module class, `AdminModule`.

When the router navigates to this route, it uses the `loadChildren` string to dynamically load the `AdminModule`. Then it adds the `AdminModule` routes to its current route configuration. Finally, it loads the requested route to the destination admin component.

The lazy loading and re-configuration happen just once, when the route is *first* requested; the module and routes are available immediately for subsequent requests.

Angular provides a built-in module loader that supports SystemJS to load modules asynchronously. If you were using another bundling tool, such as Webpack, you would use the Webpack mechanism for asynchronously loading modules.

Take the final step and detach the admin feature set from the main application. The root `AppModule` must neither load nor reference the `AdminModule` or its files.

In `app.module.ts`, remove the `AdminModule` import statement from the top of the file and remove the `AdminModule` from the NgModule's `imports` array.

{@a can-load-guard}

CanLoad Guard: guarding unauthorized loading of feature modules

You're already protecting the `AdminModule` with a `CanActivate` guard that prevents unauthorized users from accessing the admin feature area. It redirects to the login page if the user is not authorized.

But the router is still loading the `AdminModule` even if the user can't visit any of its components. Ideally, you'd only load the `AdminModule` if the user is logged in.

Add a **`CanLoad`** guard that only loads the `AdminModule` once the user is logged in *and* attempts to access the admin feature area.

The existing `AuthGuard` already has the essential logic in its `checkLogin()` method to support the `CanLoad` guard.

Open `auth-guard.service.ts`. Import the `CanLoad` interface from `@angular/router`. Add it to the `AuthGuard` class's `implements` list. Then implement `canLoad()` as follows:

The router sets the `canLoad()` method's `route` parameter to the intended destination URL. The `checkLogin()` method redirects to that URL once the user has logged in.

Now import the `AuthGuard` into the `AppRoutingModule` and add the `AuthGuard` to the `canLoad` array property for the `admin` route. The completed admin route looks like this:

{@a preloading}

Preloading: background loading of feature areas

You've learned how to load modules on-demand. You can also load modules asynchronously with *preloading*.

This may seem like what the app has been doing all along. Not quite. The `AppModule` is loaded when the application starts; that's *eager* loading. Now the `AdminModule` loads only when the user clicks on a link;

that's *lazy* loading.

Preloading is something in between. Consider the *Crisis Center*. It isn't the first view that a user sees. By default, the *Heroes* are the first view. For the smallest initial payload and fastest launch time, you should eagerly load the `AppModule` and the `HeroesModule`.

You could lazy load the *Crisis Center*. But you're almost certain that the user will visit the *Crisis Center* within minutes of launching the app. Ideally, the app would launch with just the `AppModule` and the `HeroesModule` loaded and then, almost immediately, load the `CrisisCenterModule` in the background. By the time the user navigates to the *Crisis Center*, its module will have been loaded and ready to go.

That's *preloading*.

{@a how-preloading}

How preloading works

After each *successful* navigation, the router looks in its configuration for an unloaded module that it can preload. Whether it preloads a module, and which modules it preloads, depends upon the *preload strategy*.

The `Router` offers two preloading strategies out of the box:

- No preloading at all which is the default. Lazy loaded feature areas are still loaded on demand.
- Preloading of all lazy loaded feature areas.

Out of the box, the router either never preloads, or preloads every lazy load module. The `Router` also supports custom preloading strategies for fine control over which modules to preload and when.

In this next section, you'll update the `CrisisCenterModule` to load lazily by default and use the `PreloadAllModules` strategy to load it (and *all other* lazy loaded modules) as soon as possible.

{@a lazy-load-crisis-center}

Lazy load the *crisis center*

Update the route configuration to lazy load the `CrisisCenterModule`. Take the same steps you used to configure `AdminModule` for lazy load.

1. Change the `crisis-center` path in the `CrisisCenterRoutingModule` to an empty string.
2. Add a `crisis-center` route to the `AppRoutingModule`.

3. Set the `loadChildren` string to load the `CrisisCenterModule`.

4. Remove all mention of the `CrisisCenterModule` from `app.module.ts`.

Here are the updated modules *before enabling preload*:

You could try this now and confirm that the `CrisisCenterModule` loads after you click the "Crisis Center" button.

To enable preloading of all lazy loaded modules, import the `PreloadAllModules` token from the Angular router package.

The second argument in the `RouterModule.forRoot` method takes an object for additional configuration options. The `preloadingStrategy` is one of those options. Add the `PreloadAllModules` token to the `forRoot` call:

This tells the `Router` preloader to immediately load *all* lazy loaded routes (routes with a `loadChildren` property).

When you visit `http://localhost:3000`, the `/heroes` route loads immediately upon launch and the router starts loading the `CrisisCenterModule` right after the `HeroesModule` loads.

Surprisingly, the `AdminModule` does *not* preload. Something is blocking it.

{@a preload-canload}

CanLoad blocks preload

The `PreloadAllModules` strategy does not load feature areas protected by a CanLoad guard. This is by design.

You added a `CanLoad` guard to the route in the `AdminModule` a few steps back to block loading of that module until the user is authorized. That `CanLoad` guard takes precedence over the preload strategy.

If you want to preload a module *and* guard against unauthorized access, drop the `canLoad()` guard method and rely on the canActivate() guard alone.

{@a custom-preloading}

Custom Preloading Strategy

Preloading every lazy loaded modules works well in many situations, but it isn't always the right choice, especially on mobile devices and over low bandwidth connections. You may choose to preload only certain

feature modules, based on user metrics and other business and technical factors.

You can control what and how the router preloads with a custom preloading strategy.

In this section, you'll add a custom strategy that *only* preloads routes whose `data.preload` flag is set to `true`. Recall that you can add anything to the `data` property of a route.

Set the `data.preload` flag in the `crisis-center` route in the `AppRoutingModule`.

Add a new file to the project called `selective-preloading-strategy.ts` and define a `SelectivePreloadingStrategy` service class as follows:

`SelectivePreloadingStrategy` implements the `PreloadingStrategy`, which has one method, `preload`.

The router calls the `preload` method with two arguments:

1. The route to consider.
2. A loader function that can load the routed module asynchronously.

An implementation of `preload` must return an `Observable`. If the route should preload, it returns the observable returned by calling the loader function. If the route should *not* preload, it returns an `Observable` of `null`.

In this sample, the `preload` method loads the route if the route's `data.preload` flag is truthy.

It also has a side-effect. `SelectivePreloadingStrategy` logs the `path` of a selected route in its public `preloadedModules` array.

Shortly, you'll extend the `AdminDashboardComponent` to inject this service and display its `preloadedModules` array.

But first, make a few changes to the `AppRoutingModule`.

1. Import `SelectivePreloadingStrategy` into `AppRoutingModule`.
2. Replace the `PreloadAllModules` strategy in the call to `forRoot` with this `SelectivePreloadingStrategy`.
3. Add the `SelectivePreloadingStrategy` strategy to the `AppRoutingModule` providers array so it can be injected elsewhere in the app.

Now edit the `AdminDashboardComponent` to display the log of preloaded routes.

1. Import the `SelectivePreloadingStrategy` (it's a service).
2. Inject it into the dashboard's constructor.

3. Update the template to display the strategy service's `preloadedModules` array.

When you're done it looks like this.

Once the application loads the initial route, the `CrisisCenterModule` is preloaded. Verify this by logging in to the `Admin` feature area and noting that the `crisis-center` is listed in the `Preloaded Modules`. It's also logged to the browser's console.

{@a redirect-advanced}

Migrating URLs with Redirects

You've setup the routes for navigating around your application. You've used navigation imperatively and declaratively to many different routes. But like any application, requirements change over time. You've setup links and navigation to `/heroes` and `/hero/:id` from the `HeroListComponent` and `HeroDetailComponent` components. If there was a requirement that links to `heroes` become `superheroes`, you still want the previous URLs to navigate correctly. You also don't want to go and update every link in your application, so redirects makes refactoring routes trivial.

{@a url-refactor}

Changing /heroes to /superheroes

Let's take the `Hero` routes and migrate them to new URLs. The `Router` checks for redirects in your configuration before navigating, so each redirect is triggered when needed. To support this change, you'll add redirects from the old routes to the new routes in the `heroes-routing.module`.

You'll notice two different types of redirects. The first change is from `/heroes` to `/superheroes` without any parameters. This is a straightforward redirect, unlike the change from `/hero/:id` to `/superhero/:id`, which includes the `:id` route parameter. Router redirects also use powerful pattern matching, so the `Router` inspects the URL and replaces route parameters in the `path` with their appropriate destination. Previously, you navigated to a URL such as `/hero/15` with a route parameter `id` of `15`.

The `Router` also supports [query parameters](#query-parameters) and the [fragment](#fragment) when using redirects. * When using absolute redirects, the `Router` will use the query parameters and the fragment from the redirectTo in the route config. * When using relative redirects, the `Router` use the query params and the fragment from the source URL.

Before updating the `app-routing.module.ts`, you'll need to consider an important rule. Currently, our empty path route redirects to `/heroes`, which redirects to `/superheroes`. This *won't* work and is by

design as the `Router` handles redirects once at each level of routing configuration. This prevents chaining of redirects, which can lead to endless redirect loops.

So instead, you'll update the empty path route in `app-routing.module.ts` to redirect to `/superheroes`.

Since `RouterLink`s aren't tied to route configuration, you'll need to update the associated router links so they remain active when the new route is active. You'll update the `app.component.ts` template for the `/heroes` routerLink.

With the redirects setup, all previous routes now point to their new destinations and both URLs still function as intended.

{@a inspect-config}

Inspect the router's configuration

You put a lot of effort into configuring the router in several routing module files and were careful to list them in the proper order. Are routes actually evaluated as you planned? How is the router really configured?

You can inspect the router's current configuration any time by injecting it and examining its `config` property. For example, update the `AppModule` as follows and look in the browser console window to see the finished route configuration.

{@a final-app}

Wrap up and final app

You've covered a lot of ground in this guide and the application is too big to reprint here. Please visit the where you can download the final source code.

{@a appendices}

Appendices

The balance of this guide is a set of appendices that elaborate some of the points you covered quickly above.

The appendix material isn't essential. Continued reading is for the curious.

{@a link-parameters-array}

Appendix: 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:

You've written a two element array when specifying a route parameter like this:

You can provide optional route parameters in an object like this:

These three examples cover the need for an app with one level routing. The moment you add a child router, such as the crisis center, you create new link array possibilities.

Recall that you specified a default child route for the crisis center so this simple `RouterLink` is fine.

Parse it out.

- The first item in the array identifies the parent route (`/crisis-center`).
- There are no parameters for this parent route so you're done with it.
- 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.
- Voilà! `['/crisis-center']` .

Take it a step further. Consider the following router link that navigates from the root of the application down to the *Dragon Crisis*:

- The first item in the array identifies the parent route (`/crisis-center`).
- There are no parameters for this parent route so you're done with it.
- 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` .

If you wanted to, you could redefine the `AppComponent` template with *Crisis Center* routes exclusively:

In sum, 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.

{@a browser-url-styles}

{@a location-strategy}

Appendix: *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. This is a strictly local URL. The browser shouldn't send this URL to the server and should 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`, making it the default strategy. You can switch to the `HashLocationStrategy` with an override during the bootstrapping process if you prefer it.

Learn about providers and the bootstrap process in the [Dependency Injection guide](guide/dependency-injection#bootstrap).

Which strategy is best?

You must choose a strategy and you need to make the right call early in the project. It won't be easy to change later once the application is in production and there are lots of application URL references in the wild.

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* later.

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.

Stick with the default unless you have a compelling reason to resort to hash routes.

HTML5 URLs and the *<base href>*

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`.

Without that tag, the browser may not be able to load resources (images, CSS, scripts) when "deep linking" into the app. Bad things could happen when someone pastes an application link into the browser's address bar or clicks such a link in an email.

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 two remedial steps:

1. Provide the router with an appropriate [APP*BASE*HREF][] value.
2. Use *root URLs* for all web resources: CSS, images, scripts, and template HTML files.

{@a hashlocationstrategy}

HashLocationStrategy

You can go old-school with the `HashLocationStrategy` by providing the `useHash: true` in an object as the second argument of the `RouterModule.forRoot` in the `AppModule`.