{@a router-tutorial}

Router tutorial: tour of heroes

This tutorial provides an extensive overview of the Angular router. In this tutorial, you build upon a basic router configuration to explore features such as child routes, route parameters, lazy load NgModules, guard routes, and preloading data to improve the user experience.

For a working example of the final version of the app, see the .

{@a router-tutorial-objectives}

Objectives

This guide describes development of a multi-page routed sample application. Along the way, it highlights 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 an `NgModule`.
- The `CanLoad` guard (check before loading feature module assets).

This guide proceeds as a sequence of milestones as if you were building the application step-by-step, but assumes you are familiar with basic Angular concepts. For a general introduction to angular, see the Getting Started. For a more in-depth overview, see the Tour of Heroes tutorial.

Prerequisites

To complete this tutorial, you should have a basic understanding of the following concepts:

- JavaScript
- HTML
- CSS
- Angular CLI

You might find the Tour of Heroes tutorial helpful, but it is not required.

The sample application in action

The sample application for this tutorial helps the Hero Employment Agency find crises for heroes 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.

The application renders with a row of navigation buttons and the *Heroes* view with its list of heroes.

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

Alter the name. Click the "Back" button and the application 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 application's "Back" button, the application would have returned you to the heroes list as well. Angular application 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 component 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.

Click the browser back button or the "Heroes" link to activate a dialog.

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 covered later in the guide.

{@a getting-started}

Milestone 1: Getting started

Begin with a basic version of the application that navigates between two empty views.

{@a import}

Generate a sample application with the Angular CLI.

ng new angular-router-sample

Define Routes

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

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.

The first route does the following:

- 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.

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

Generate the `CrisisList` and `HeroList` components so that the router has something to render.

ng generate component crisis-list

ng generate component hero-list

Replace the contents of each component with the following sample HTML.

Register `Router` and `Routes`

To use the `Router`, you must first register the `RouterModule` from the `@angular/router` package. Define an array of routes, `appRoutes`, and pass them to the `RouterModule.forRoot()` method. The `RouterModule.forRoot()` method returns a module that contains 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.

Note: The `RouterModule.forRoot()` method is a pattern used to register application-wide providers. Read more about application-wide providers in the Singleton services guide.

Adding the configured `RouterModule` to the `AppModule` is sufficient for minimal route configurations. However, as the application grows, refactor the routing configuration into a separate file and create a Routing Module. A routing module is a special type of `Service Module` dedicated to routing.

Registering the `RouterModule.forRoot()` in the `AppModule imports` array makes the `Router` service available everywhere in the application.

{@a shell}

Add the Router Outlet

The root `AppComponent` is the application shell. It has a title, a navigation bar with two links, and a router outlet where the router renders components.

The router outlet serves as a placeholder where the routed components are rendered.

{@a shell-template}

The corresponding component template looks like this:

{@a wildcard}

Define a Wildcard route

You've created two routes in the application 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. Thus, the router selects this wildcard 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* strategy. Because a wildcard route is the least specific route, place it last in the route configuration.

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

The application fails 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 and have it navigate to a `PageNotFoundComponent`.

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

ng generate component page-not-found

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 redirect}

Set up redirects

When the application launches, the initial URL in the browser bar is by default:

localhost:4200

That doesn't match any of the hard-coded 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 application is the list of heroes. The application should navigate there as if the user clicked the "Heroes" link or pasted `localhost:4200/heroes` into the address bar.

Add a `redirect` route that translates the initial relative URL (`''`) to the default path (`/heroes`) you want.

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.

The browser address bar shows `.../heroes` as if you'd navigated there directly.

A redirect route requires a `pathMatch` property to tell the router how to match a URL to the path of a route. 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'`.

{@a pathmatch}

Spotlight on pathMatch

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. This doesn't apply to this sample application because if the `pathMatch` value were `'prefix'`, every URL would match `''`.

Try setting it to `'prefix'` and clicking the `Go to sidekicks` button. Because that's a bad URL, 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 is 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.

Milestone 1 wrap up

Your sample application can switch between two views when the user clicks a link.

Milestone 1 covered 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 are 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 application launches with an empty path.

The starter application's structure looks like this:

```
angular-router-sample
```

```
<div class='file'>
  src
</div>

<div class='children'>

  <div class='file'>
    app
  </div>

  <div class='children'>

    <div class='file'>
      crisis-list
    </div>

    <div class='children'>
```

```
  <div class='file'>

    crisis-list.component.css

  </div>

  <div class='file'>

    crisis-list.component.html

  </div>

  <div class='file'>

    crisis-list.component.ts

  </div>

</div>

<div class='file'>
  hero-list
</div>

<div class='children'>

  <div class='file'>

    hero-list.component.css

  </div>

  <div class='file'>

    hero-list.component.html

  </div>

  <div class='file'>

    hero-list.component.ts

  </div>

</div>
```

```
  <div class='file'>
    page-not-found
  </div>

  <div class='children'>

    <div class='file'>

      page-not-found.component.css

    </div>

    <div class='file'>

      page-not-found.component.html

    </div>

    <div class='file'>

      page-not-found.component.ts

    </div>

  </div>

  <div class='file'>
    app.component.css
  </div>

  <div class='file'>
    app.component.html
  </div>

  <div class='file'>
    app.component.ts
  </div>

  <div class='file'>
    app.module.ts
  </div>

</div>

<div class='file'>
```

```
    main.ts
  </div>

  <div class='file'>
    index.html
  </div>

  <div class='file'>
    styles.css
  </div>

  <div class='file'>
    tsconfig.json
  </div>

</div>

<div class='file'>
  node_modules ...
</div>

<div class='file'>
  package.json
</div>
```

Here are the files in this milestone.

{@a routing-module}

Milestone 2: *Routing module*

This milestone shows you how to configure a special-purpose module called a *Routing Module*, which holds your application's routing configuration.

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 such as guards and resolvers.
- Does not declare components.

{@a integrate-routing}

Integrate routing with your app

The sample routing application does not include routing by default. When you use the Angular CLI to create a project that does use routing, set the `--routing` option for the project or application, and for each NgModule. When you create

or initialize a new project (using the CLI `ng new` command) or a new application (using the `ng generate app` command), specify the `--routing` option. This tells the CLI to include the `@angular/router` npm package and create a file named `app-routing.module.ts`. You can then use routing in any NgModule that you add to the project or application.

For example, the following command generates an NgModule that can use routing.

```
ng generate module my-module --routing
```

This creates a separate file named `my-module-routing.module.ts` to store the NgModule's routes. The file includes an empty `Routes` object that you can fill with routes to different components and NgModules.

{@a routing-refactor}

Refactor the routing configuration into a routing module

Create an `AppRouting` module in the `/app` folder to contain the routing configuration.

ng generate module app-routing –module app –flat

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

Re-export the Angular `RouterModule` by adding it to the module `exports` array. By re-exporting the `RouterModule` here, the components declared in `AppModule` 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 by removing `RouterModule.forRoot` in the `imports` array.

Later, this guide shows you how to create multiple routing modules and import those routing modules in the correct 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}

Benefits of a routing module

The routing module, often called the `AppRoutingModule`, replaces the routing configuration in the root or feature module.

The routing module is helpful as your application grows and when the configuration includes specialized guard and resolver services.

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

Most applications should implement a routing module for 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

This milestone covers the following:

- Organizing the application and routes into feature areas using modules.
- Navigating imperatively from one component to another.
- Passing required and optional information in route parameters.

This sample application recreates the heroes feature in the "Services" section of the Tour of Heroes tutorial, and reuses much of the code from the .

A typical application has multiple feature areas, each dedicated to a particular business purpose with its own folder.

This section shows you how refactor the application into different feature modules, import them into the main module and navigate among them.

{@a heroes-functionality}

Add heroes functionality

Follow these steps:

- To manage the heroes, create a `HeroesModule` with routing in the heroes folder and register it with the root `AppModule`.

ng generate module heroes/heroes –module app –flat –routing

- Move the placeholder `hero-list` folder that's in the `app` folder into the `heroes` folder.
- Copy the contents of the `heroes/heroes.component.html` from the "Services" tutorial into the `hero-list.component.html` template.
  - Re-label the `<h2>` to `<h2>HEROES</h2>`.
  - Delete the `<app-hero-detail>` component at the bottom of the template.

- Copy the contents of the `heroes/heroes.component.css` from the live example into the `hero-list.component.css` file.
- Copy the contents of the `heroes/heroes.component.ts` from the live example into the `hero-list.component.ts` file.
  - Change the component class name to `HeroListComponent`.
  - Change the `selector` to `app-hero-list`.

Selectors are not required for routed components because components are dynamically inserted when the page is rendered. However, they are useful for identifying and targeting them in your HTML element tree.

- Copy the `hero-detail` folder, the `hero.ts`, `hero.service.ts`, and `mock-heroes.ts` files into the `heroes` subfolder.
- Copy the `message.service.ts` into the `src/app` folder.
- Update the relative path import to the `message.service` in the `hero.service.ts` file.

Next, update the `HeroesModule` metadata.

- Import and add the `HeroDetailComponent` and `HeroListComponent` to the `declarations` array in the `HeroesModule`.

The hero management file structure is as follows:

```
src/app/heroes
```

```
<div class='file'>
  hero-detail
</div>

  <div class='children'>

    <div class='file'>
      hero-detail.component.css
    </div>

    <div class='file'>
      hero-detail.component.html
    </div>

    <div class='file'>
      hero-detail.component.ts
    </div>

  </div>

<div class='file'>
  hero-list
```

```
</div>

  <div class='children'>

    <div class='file'>
      hero-list.component.css
    </div>

    <div class='file'>
      hero-list.component.html
    </div>

    <div class='file'>
      hero-list.component.ts
    </div>

  </div>

<div class='file'>
  hero.service.ts
</div>

<div class='file'>
  hero.ts
</div>

<div class='file'>
  heroes-routing.module.ts
</div>

<div class='file'>
  heroes.module.ts
</div>

<div class='file'>
  mock-heroes.ts
</div>

</div>
```

{@a hero-routing-requirements}

Hero feature routing requirements The heroes feature has two interacting components, the hero list and the hero detail. When you navigate to list view, it gets a list of heroes and displays them. When you click on a hero, the detail view has to display that particular hero.

You tell the detail view which hero to display by including the selected hero's ID in the route URL.

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

Now that you have routes for the `Heroes` module, register them with the `Router` using the `RouterModule` as you did in the `AppRoutingModule`, with an important 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.

The updated `HeroesRoutingModule` looks like this:

Consider giving each feature module its own route configuration file. Though the feature routes are currently minimal, routes have a tendency to grow more complex even in small applications.

{@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 lets you continue defining the feature module routes without modifying the main route configuration.

Remove the `HeroListComponent` import and the `/heroes` route from the `app-routing.module.ts`.

Leave the default and the wildcard routes as these are still in use at the top level of the application.

{@a merge-hero-routes}

Remove heroes declarations Because the `HeroesModule` now provides the `HeroListComponent`, remove it from the `AppModule`'s `declarations` array. Now that you have a separate `HeroesModule`, you can evolve the hero feature with more components and different routes.

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

{@a routing-module-order}

Module import order

Notice that in the module `imports` array, the `AppRoutingModule` is last and comes *after* the `HeroesModule`.

The order of route configuration is important because 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".

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

Reverse the routing modules to see a click of the heroes link resulting in "Page not found". Learn about inspecting the runtime router configuration below.

Route Parameters

{@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 an `:id` token in the path.

The `:id` token creates a slot in the path for a Route Parameter. In this case, this configuration causes the router to 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:4200/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:4200/hero/15`.

The router extracts the route parameter (`id:15`) from the URL and supplies it to the `HeroDetailComponent` using the `ActivatedRoute` service.

{@a activated-route-in-action}

`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}

Add the services as private variables to the constructor so that Angular injects them (makes them visible to the component).

In the `ngOnInit()` method, use the `ActivatedRoute` service to retrieve the parameters for the route, pull the hero `id` from the parameters, and retrieve the hero to display.

When the map changes, `paramMap` gets 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.

The `switchMap` operator does two things. It flattens the `Observable<Hero>` that `HeroService` returns and cancels previous pending 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`.

`AsyncPipe` handles the observable subscription 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.

</td>

get(name)

Returns the parameter name value (a `string`) if present, or `null` if the parameter name is

</td>

getAll(name)

Returns a `string array` of the parameter name value if found, or an empty `array` if the pa

</td>

keys

Returns a `string array` of all parameter names in the map.

</td>

{@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.

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 wouldn't want the router to remove the current `HeroDetailComponent` instance from the DOM only to re-create it for the next `id` as this would re-render the view. For better UX, the router re-uses the same component instance and updates the parameter.

Because `ngOnInit()` is only called once per component instantiation, you can detect when the route parameters change from *within the same instance* using the observable `paramMap` property.

When subscribing to an observable in a component, you almost always unsubscribe when the component is destroyed.

However, `ActivatedRoute` observables are among the exceptions because `ActivatedRoute` and its observables are insulated from the `Router` itself. The

`Router` destroys a routed component when it is no longer needed. This means all the component's members will also be destroyed, including the injected `ActivatedRoute` and the subscriptions to its `Observable` properties.

The `Router` does not `complete` any `Observable` of the `ActivatedRoute` so any `finalize` or `complete` blocks will not run. If you need to handle something in a `finalize`, you still need to unsubscribe in `ngOnDestroy`. You also have to unsubscribe if your observable pipe has a delay with code you do not want to run after the component is destroyed.

{@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 be re-used, you can use `snapshot`.

`route.snapshot` provides the initial value of the route parameter map. You can access the parameters directly without subscribing or adding observable operators as in the following:

`snapshot` only gets the initial value of the parameter map with this technique. Use the observable `paramMap` approach if there's a possibility that the router could re-use the component. This tutorial sample application uses with the observable `paramMap`.

{@a nav-to-list}

Navigating back to the list component

The `HeroDetailComponent` "Back" button uses the `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:4200/hero/15

You can also add optional information to a route request. For example, when returning to the `hero-detail.component.ts` list from the hero detail view, it would be nice if the viewed hero were preselected in the list.

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

Optional information can also include other forms such as:

- Loosely structured search criteria; for example, `name='wind*'`.
- Multiple values; for example, `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'`.

As these kinds of parameters don't fit smoothly in a URL path, you can use optional parameters 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, use a required route parameter when the value is mandatory (for example, if necessary to distinguish one route path from another); and 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 previously you didn't need to send information to the `HeroListComponent`.

Now, send the `id` of the current hero with the navigation request so that the `HeroListComponent` can highlight that hero in its list.

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:4200/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.

Matrix URL notation is an idea first introduced in a 1996 proposal 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. As such, the Router provides support for the matrix notation across browsers.

{@a route-parameters-activated-route}

Route parameters in the `ActivatedRoute` service

In its current state of development, the list of heroes is unchanged. No hero row is highlighted.

The `HeroListComponent` needs code that expects parameters.

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, 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.

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:

Add some styles to apply when the hero is selected.

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 the router continues to ignore it.

{@a route-animation}

Adding routable animations

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

First, import the `BrowserAnimationsModule` and add it to the `imports` array:

Next, add a `data` object to the routes for `HeroListComponent` and `HeroDetailComponent`. Transitions are based on `states` and you use the `animation` data from the route to provide a named animation `state` for the transitions.

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 `slideInAnimation` set to an animation trigger named `routeAnimation`.
- Defines one transition when switching back and forth from the `heroes` and `hero` routes to ease the component in from the left of the screen as it enters the application view (`:enter`), the other to animate the component to the right as it leaves the application view (`:leave`).

Back in the `AppComponent`, import the `RouterOutlet` token from the `@angular/router` package and the `slideInAnimation` from `'./animations.ts`.

Add an `animations` array to the `@Component` metadata that contains the `slideInAnimation`.

To use the routable animations, wrap the `RouterOutlet` inside an element, use the `@routeAnimation` trigger, and bind it to the element.

For the `@routeAnimation` transitions to key off states, provide it with the `data` from the `ActivatedRoute`. The `RouterOutlet` is exposed as an `outlet` template variable, so you bind a reference to the router outlet. This example uses a variable of `routerOutlet`.

The `@routeAnimation` property is bound to the `getAnimationData()` which returns the animation property from the `data` provided by the primary route. The `animation` property matches the `transition` names you used in the `slideInAnimation` defined in `animations.ts`.

When switching between the two routes, the `HeroDetailComponent` and `HeroListComponent` now ease in from the left when routed to, and slide to the right when navigating away.

{@a milestone-3-wrap-up}

Milestone 3 wrap up

This section covered the following:

- Organizing the application into feature areas.
- Navigating imperatively from one component to another.
- Passing information along in route parameters and subscribe to them in the component.
- Importing the feature area NgModule into the `AppModule`.
- Applying routable animations based on the page.

After these changes, the folder structure is as follows:

```
angular-router-sample
```

```
<div class='file'>
  src
</div>

<div class='children'>

  <div class='file'>
    app
  </div>

  <div class='children'>

    <div class='file'>
      crisis-list
    </div>

      <div class='children'>

        <div class='file'>
          crisis-list.component.css
        </div>

        <div class='file'>
```

```
      crisis-list.component.html
    </div>

    <div class='file'>
      crisis-list.component.ts
    </div>

  </div>

<div class='file'>
  heroes
</div>

<div class='children'>

  <div class='file'>
    hero-detail
  </div>

    <div class='children'>

      <div class='file'>
        hero-detail.component.css
      </div>

      <div class='file'>
        hero-detail.component.html
      </div>

      <div class='file'>
        hero-detail.component.ts
      </div>

    </div>

  <div class='file'>
    hero-list
  </div>

    <div class='children'>

      <div class='file'>
        hero-list.component.css
      </div>

      <div class='file'>
```

```
        hero-list.component.html
      </div>

      <div class='file'>
        hero-list.component.ts
      </div>

    </div>

  <div class='file'>
    hero.service.ts
  </div>

  <div class='file'>
    hero.ts
  </div>

  <div class='file'>
    heroes-routing.module.ts
  </div>

  <div class='file'>
    heroes.module.ts
  </div>

  <div class='file'>
    mock-heroes.ts
  </div>

</div>

<div class='file'>
  page-not-found
</div>

<div class='children'>

  <div class='file'>

    page-not-found.component.css

  </div>

  <div class='file'>

    page-not-found.component.html
```

```
    </div>

    <div class='file'>

      page-not-found.component.ts

    </div>

  </div>

</div>

<div class='file'>
  animations.ts
</div>

<div class='file'>
  app.component.css
</div>

<div class='file'>
  app.component.html
</div>

<div class='file'>
  app.component.ts
</div>

<div class='file'>
  app.module.ts
</div>

<div class='file'>
  app-routing.module.ts
</div>

<div class='file'>
  main.ts
</div>

<div class='file'>
  message.service.ts
</div>

<div class='file'>
```

```
    index.html
  </div>

  <div class='file'>
    styles.css
  </div>

  <div class='file'>
    tsconfig.json
  </div>

</div>

<div class='file'>
  node_modules ...
</div>

<div class='file'>
  package.json
</div>
```

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

{@a milestone-4}

Milestone 4: Crisis center feature

This section shows you how to add child routes and use relative routing in your app.

To add more features to the application's current crisis center, take similar steps as for the heroes feature:

- Create a `crisis-center` subfolder in the `src/app` folder.
- Copy the files and folders from `app/heroes` into the new `crisis-center` folder.
- In the new files, change every mention of "hero" to "crisis", and "heroes" to "crises".
- Rename the NgModule files to `crisis-center.module.ts` and `crisis-center-routing.module.ts`.

Use 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.

In keeping with the Separation of Concerns principle, changes to the Crisis Center don'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 application had many feature areas, the component trees might consist of multiple components for those features, each with branches of other, related, components.

{@a child-routing-component}

Child routing component

Generate a `CrisisCenter` component in the `crisis-center` folder:

ng generate component crisis-center/crisis-center

Update the component template with the following markup:

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 minimal because it has no business logic, and its template has no links, just a title and `<router-outlet>` for the crisis center child component.

{@a child-route-config}

Child route configuration

As a host page for the "Crisis Center" feature, generate a `CrisisCenterHome` component in the `crisis-center` folder.

ng generate component crisis-center/crisis-center-home

Update the template with a welcome message to the `Crisis Center`.

Update the `crisis-center-routing.module.ts` you renamed after copying it from `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 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`. The router reuses components by default, so the `Crisis Detail` component is 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 from the list of heroes.

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 as follows:

localhost:4200/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`:

The import order of the modules is important because the order of the routes defined in the modules affects route matching. If the `AppModule` were imported first, its wildcard route (`path: '**'`) would take precedence over the routes defined in `CrisisCenterModule`. For more information, see the section on route order.

Remove the initial crisis center route from the `app-routing.module.ts` because now the `HeroesModule` and the `CrisisCenter` modules provide the feature routes.

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.

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 `../<sibling>` 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 display different content, determined by different routes, all at the same time.

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

That's where a popup goes, 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.

Generate a new component to compose the message.

ng generate component compose-message

It displays a short form with a header, an input box for the message, and two buttons, “Send” and “Cancel”.

Here’s the component, its template, and styles:

It looks similar to any other component in this guide, but there are two key 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` which the section on clearing secondary routes covers.

{@a add-secondary-route}

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

In addition to the `path` and `component` properties, there’s a new property called `outlet`, which is set to `'popup'`. This route now targets the popup outlet and the `ComposeMessageComponent` will display there.

To give users a way to open the popup, add a “Contact” link to the `AppComponent` template.

Although the `compose` route is configured to the “popup” outlet, that’s not sufficient for connecting 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.

In other words, when the user clicks this link, the router displays the component associated with the `compose` route in the `popup` outlet.

This `outlets` object within an outer object was unnecessary when there was only one route and one unnamed outlet.

The router assumed that your route specification targeted the unnamed primary outlet and created these objects for you.

Routing to a named outlet revealed a router feature: you can target multiple outlets with multiple routes in the same `RouterLink` directive.

{@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 relevant 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 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 and the router will generate the URLs to go with it.

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

{@a clear-secondary-routes}

Clearing secondary routes Like regular outlets, secondary outlets persists until you navigate away to a new component.

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.

The `closePopup()` method again:

Clicking the "send" or "cancel" buttons clears the popup view. The `closePopup()` function 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}

{@a milestone-5-route-guards}

Milestone 5: Route guards

At the moment, any user can navigate anywhere in the application any time, but sometimes you need to control access to different parts of your application for various reasons, some of which might include the following:

- 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 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.
- If it returns a `UrlTree`, the current navigation cancels and a new navigation is initiated to the `UrlTree` returned.

Note: The guard can also tell the router to navigate elsewhere, effectively canceling the current navigation. When doing so inside a guard, the guard should return `false`;

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`.

Note: The observable provided to the `Router` automatically completes after retrieving the first value.

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` guards first, from the deepest child route to the top. Then it checks the `CanActivate` and `CanActivateChild` 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 are 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 This section guides you through extending the crisis center with some new administrative features. Start by adding a new feature module named `AdminModule`.

Generate an `admin` folder with a feature module file and a routing configuration file.

ng generate module admin –routing

Next, generate the supporting components.

ng generate component admin/admin-dashboard

ng generate component admin/admin

ng generate component admin/manage-crises

ng generate component admin/manage-heroes

The admin feature file structure looks like this:

`src/app/admin`

```
<div class='file'>
  admin
</div>

  <div class='children'>

    <div class='file'>
      admin.component.css
    </div>
```

```
    <div class='file'>
      admin.component.html
    </div>

    <div class='file'>
      admin.component.ts
    </div>

  </div>

<div class='file'>
  admin-dashboard
</div>

  <div class='children'>

    <div class='file'>
      admin-dashboard.component.css
    </div>

    <div class='file'>
      admin-dashboard.component.html
    </div>

    <div class='file'>
      admin-dashboard.component.ts
    </div>

  </div>

<div class='file'>
  manage-crises
</div>

  <div class='children'>

    <div class='file'>
      manage-crises.component.css
    </div>

    <div class='file'>
      manage-crises.component.html
    </div>

    <div class='file'>
```

```
      manage-crises.component.ts
    </div>

  </div>

<div class='file'>
  manage-heroes
</div>

  <div class='children'>

    <div class='file'>
      manage-heroes.component.css
    </div>

    <div class='file'>
      manage-heroes.component.html
    </div>

    <div class='file'>
      manage-heroes.component.ts
    </div>

  </div>

<div class='file'>
  admin.module.ts
</div>

<div class='file'>
  admin-routing.module.ts
</div>
```

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.

Although the admin dashboard `RouterLink` only contains a relative slash without an additional URL segment, it is 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.

{@a component-less-route}

Component-less route: grouping routes without a component The initial admin routing configuration:

The child route under the `AdminComponent` has a `path` and a `children` property but it's not using a `component`. This defines a *component-less* route.

To group the `Crisis Center` management routes under the `admin` path a component is unnecessary. Additionally, 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.

Write a `canActivate()` guard method to redirect anonymous users to the login page when they try to enter the admin area.

Generate an `AuthGuard` in the `auth` folder.

ng generate guard auth/auth

To demonstrate the fundamentals, this example only logs to the console, `returns` true immediately, and lets navigation 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, but the guard requires more customization to work fully.

{@a teach-auth}

Authenticate with `AuthGuard` Make the `AuthGuard` mimic authentication.

The `AuthGuard` should call an application service that can login a user and retain information about the current user. Generate a new `AuthService` in the `auth` folder:

ng generate service auth/auth

Update the `AuthService` to log in the user:

Although it doesn't actually log in, 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 stores the URL that the user wanted to access so you can navigate to it after authentication.

To keep things minimal, this example redirects unauthenticated users to `/admin`.

Revise the `AuthGuard` to call the `AuthService`.

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 redirect to a login page—a page you haven't created yet. Returning a `UrlTree` tells the `Router` to cancel the current navigation and schedule a new one to redirect the user.

{@a add-login-component}

Add the `LoginComponent` You need a `LoginComponent` for the user to log in to the application. 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 use it in the router configuration.

ng generate component auth/login

Register a `/login` route in the `auth/auth-routing.module.ts`. In `app.module.ts`, import and add the `AuthModule` to the `AppModule` imports.

{@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.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|UrlTree>` or `Promise<boolean|UrlTree>` for async checks and a `boolean` or `UrlTree` for sync checks. This one returns either `true` to let the user access the admin feature module or `UrlTree` to redirect the user to the login page instead:

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 application accepts every change to a hero immediately without validation.

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

When the user navigates away, you can let the user decide what to do with unsaved changes. If the user cancels, you'll stay put and allow more changes. If the user approves, the application can save.

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

You need to stop the navigation while you wait, asynchronously, for the server to return with its answer.

The `CanDeactivate` guard helps you decide what to do with unsaved changes and how to proceed.

{@a cancel-save}

Cancel and save Users update crisis information in the `CrisisDetailComponent`. Unlike the `HeroDetailComponent`, the user changes do not update the crisis entity immediately. Instead, the application 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.

In this scenario, the user could click the heroes link, cancel, push the browser back button, or navigate away without saving.

This example application asks the user to be explicit with a confirmation dialog box that waits asynchronously for the user's response.

You could wait for the user's answer with synchronous, blocking code, however, the application is more responsive—and can do other work—by waiting for the user's answer asynchronously.

Generate a `Dialog` service to handle user confirmation.

ng generate service dialog

Add a `confirm()` method to the `DialogService` to prompt the user to confirm their intent. The `window.confirm` is a blocking action that displays a modal dialog and waits for user interaction.

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}

Generate a guard that checks for the presence of a `canDeactivate()` method in a component—any component.

ng generate guard can-deactivate

Paste the following code into your guard.

While the guard doesn't have to know which component has a deactivate method, it can detect that the `CrisisDetailComponent` component has the `canDeactivate()` method and call it. The guard not knowing the details of any component's deactivation method 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 on the current route).

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

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 application waited until the route was activated to fetch the respective hero or crisis.

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.

To improve this behavior, you can pre-fetch data from the server using a resolver so it's ready the moment the route is activated. This also lets you 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 has been fetched.

{@a fetch-before-navigating}

Fetch data before navigating At the moment, the `CrisisDetailComponent` retrieves the selected crisis. If the crisis is not found, the router 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` did not exist, *before* activating the route and creating the `CrisisDetailComponent`.

Generate a `CrisisDetailResolver` service file within the `Crisis Center` feature area.

ng generate service crisis-center/crisis-detail-resolver

Move the relevant parts of the crisis retrieval logic in `CrisisDetailComponent.ngOnInit()` into the `CrisisDetailResolverService`. Import the `Crisis` model, `CrisisService`, and the `Router` so you can navigate elsewhere if you can't fetch the crisis.

Be explicit and 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 in order to prevent the route from loading until the data is fetched.

If it doesn't return a valid `Crisis`, then return an empty `Observable`, cancel the previous in-progress navigation to the `CrisisDetailComponent`, and navigate the user back to the `CrisisListComponent`. The updated resolver service looks like this:

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

The `CrisisDetailComponent` should no longer fetch the crisis. When you re-configured the route, you changed where the crisis is. Update the `CrisisDetailComponent` to get the crisis from the `ActivatedRoute.data.crisis` property instead;

Note the following three important points:

1. The router's `Resolve` interface is optional. The `CrisisDetailResolverService` doesn't inherit from a base class. The router looks for that method and calls it if found.
2. The router calls the resolver in any case where the user could navigate away so you don't have to code for each use case.
3. Returning an empty `Observable` in at least one resolver cancels navigation.

The relevant Crisis Center code for this milestone follows.

Guards

{@a query-parameters}

{@a fragment}

Query parameters and fragments

In the route parameters section, you only dealt with parameters specific to the route. However, you can use query parameters to get optional parameters available to all routes.

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

Update the `AuthGuard` to provide a `session_id` query that remains 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 preserves and combines the current query parameters with any provided query parameters when navigating.

To navigate to the Admin Dashboard route after logging in, update `admin-dashboard.component.ts` to handle the query parameters and fragment.

Query parameters and fragments are also available through the `ActivatedRoute` service. 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. At some point you'll reach a point where the application takes a long time to load.

To remedy this issue, use 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 part of the 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.

Use empty path routes 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 instead of a `children` property. The `loadChildren` property takes a function that returns a promise using the browser's built-in syntax for lazy loading code using dynamic imports `import('...')`. The path is the location of the `AdminModule` (relative to the application root). After the code is requested and loaded, the `Promise` resolves an object that contains the `NgModule`, in this case the `AdminModule`.

Note: When using absolute paths, the `NgModule` file location must begin with `src/app` in order to resolve correctly. For custom path mapping with absolute paths, you must configure the `baseUrl` and `paths` properties in the project `tsconfig.json`.

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`: 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.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

In addition to loading modules on-demand, you can load modules asynchronously with preloading.

The `AppModule` is eagerly loaded when the application starts, meaning that it loads right away. Now the `AdminModule` loads only when the user clicks on a link, which is called lazy loading.

Preloading lets you load modules in the background so that the data is ready to render when the user activates a particular route. 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 application 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 is loaded and ready.

{@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:

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

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

This section guides you through updating the `CrisisCenterModule` to load lazily by default and use the `PreloadAllModules` strategy to load all lazy loaded modules.

{@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 loading.

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

When you visit `http://localhost:4200`, the `/heroes` route loads immediately upon launch and the router starts loading the `CrisisCenterModule` right after the `HeroesModule` loads.

Currently, the `AdminModule` does not preload because `CanLoad` is blocking it.

{@a preload-canload}

`CanLoad` blocks preload The `PreloadAllModules` strategy does not load feature areas protected by a CanLoad guard.

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 as well as guard against unauthorized access, remove the `canLoad()` guard method and rely on the canActivate() guard alone.

{@a custom-preloading}

Custom Preloading Strategy

Preloading every lazy loaded module works well in many situations. However, in consideration of things such as low bandwidth and user metrics, you can use a custom preloading strategy for specific feature modules.

This section guides you through adding 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`.

Generate a new `SelectivePreloadingStrategy` service.

ng generate service selective-preloading-strategy

Replace the contents of `selective-preloading-strategy.service.ts` with the following:

`SelectivePreloadingStrategyService` 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 does preload, it returns the observable returned by calling the loader function. If the route does 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.

As a side-effect, `SelectivePreloadingStrategyService` 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 `SelectivePreloadingStrategyService` into `AppRoutingModule`.
2. Replace the `PreloadAllModules` strategy in the call to `forRoot()` with this `SelectivePreloadingStrategyService`.

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

1. Import the `SelectivePreloadingStrategyService`.
2. Inject it into the dashboard's constructor.
3. Update the template to display the strategy service's `preloadedModules` array.

Now the file is as follows:

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 also logs to the browser's console.

{@a redirect-advanced}

Migrating URLs with redirects

You've setup the routes for navigating around your application and used navigation imperatively and declaratively. 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 were a requirement that links to `heroes` become `superheroes`, you would still want the previous URLs to navigate correctly. You also don't want to update every link in your application, so redirects makes refactoring routes trivial.

{@a url-refactor}

Changing `/heroes` to `/superheroes` This section guides you through migrating the `Hero` routes to new URLs. The `Router` checks for redirects in your configuration before navigating, so each redirect is triggered when needed. To support this change, add redirects from the old routes to the new routes in the `heroes-routing.module`.

Notice two different types of redirects. The first change is from `/heroes` to `/superheroes` without any parameters. The second change is 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 and the fragment when using redirects.

- When using absolute redirects, the `Router` uses 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.

Currently, the empty path route redirects to `/heroes`, which redirects to `/superheroes`. This won't work because the `Router` handles redirects once at each level of routing configuration. This prevents chaining of redirects, which can lead to endless redirect loops.

Instead, update the empty path route in `app-routing.module.ts` to redirect to `/superheroes`.

A `routerLink` isn't tied to route configuration, so update the associated router links to remain active when the new route is active. Update the `app.component.ts` template for the `/heroes` `routerLink`.

Update the `goToHeroes()` method in the `hero-detail.component.ts` to navigate back to `/superheroes` with the optional route parameters.

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

To determine if your routes are actually evaluated in the proper order, you can inspect the router's configuration.

Do this by injecting the router and logging to the console 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}

Final application

For the completed router application, see the for the final source code.

{@a link-parameters-array}