Material Routing Angular

Información Importante

Este material fue extraído de la documentación oficial de Angular el día 15/06/2020 y al momento de la lectura es posible que existan actualizaciones, por eso recomendamos encarecidamente consultar regularmente el sitio oficial.

Documentación oficial: https://angular.io/guide/router

In-app navigation: routing to views

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

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

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

Prerequisites

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

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

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

Generate an app with routing enabled

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

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

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

Adding components for routing

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

```
ng generate component first
```

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

```
ng generate component second
```

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

```
<base href>
```

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

Importing your new components

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

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

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

Defining a basic route

There are three fundamental building blocks to creating a route.

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

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

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

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


   export class AppModule { }
```

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

   3.

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

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

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

```
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

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

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



];
```

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

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



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

Route order

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

Getting route information

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

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

To get information from a route:

1. Import `ActivatedRoute` and `ParamMap` to your component.
2. In the component class (excerpt)

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

3. These `import` statements add several important elements that your component needs. To learn more about each, see the following API pages:
    - `Router`
    - `ActivatedRoute`
    - `ParamMap`
4. Inject an instance of `ActivatedRoute` by adding it to your application's constructor:
5. In the component class (excerpt)

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

6. Update the `ngOnInit()` method to access the `ActivatedRoute` and track the `id` parameter:
7. In the component (excerpt)

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

```
}
```

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

Setting up wildcard routes

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

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

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

The two asterisks, `**`, indicate to Angular that this `routes` definition is a wildcard route. For the component property, you can define any component in your application. Common choices include an application-specific `PageNotFoundComponent`, which you can define to display a 404 page to your users; or a redirect to your application's main component. A wildcard route is the last route because it matches any URL. For more detail on why order matters for routes, see Route order.

Displaying a 404 page

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

```
const routes: Routes = [
  { path: 'first-component', component: FirstComponent },
  { path: 'second-component', component: SecondComponent },
  { path: '',   redirectTo: '/first-component', pathMatch: 'full' }, //
redirect to `first-component`
```

```
  { path: '**', component: FirstComponent },
  { path: '**', component: PageNotFoundComponent },  // Wildcard route for a
404 page


  ];
```

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

Setting up redirects

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

```
const routes: Routes = [
  { path: 'first-component', component: FirstComponent },
  { path: 'second-component', component: SecondComponent },
  { path: '',   redirectTo: '/first-component', pathMatch: 'full' }, //
redirect to `first-component`
  { path: '**', component: FirstComponent },


  ];
```

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

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

Nesting routes

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

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

In the template

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

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


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

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

AppRoutingModule (excerpt)

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

       ] },
```

Using relative paths

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

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

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


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

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

Specifying a relative route

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

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

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


   }
```

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

Accessing query parameters and fragments

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

First, import the following members in the component you want to navigate from.
Component import statements (excerpt)

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

   import { switchMap } from 'rxjs/operators';
```

Next inject the activated route service:
Component (excerpt)

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

Configure the class so that you have an observable, `heroes$`, a `selectedId` to hold the `id` number of the hero, and the heroes in the `ngOnInit()`, add the following code to get the `id` of the selected hero. This code snippet assumes that you have a heroes list, a hero service, a function to get your heroes, and the HTML to render your list and details, just as in the Tour of Heroes example.
Component 1 (excerpt)

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

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

```
    })
  );
}
```

Next, in the component that you want to navigate to, import the following members.
Component 2 (excerpt)

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

   import { Observable } from 'rxjs';
```

Inject `ActivatedRoute` and `Router` in the constructor of the component class so they are available to this component:
Component 2 (excerpt)

```
item$: Observable;

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

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

  gotoItems(item: Item) {
    let heroId = item ? hero.id : null;
    // Pass along the item id if available
    // so that the HeroList component can select that item.
    this.router.navigate(['/heroes', { id: itemId }]);
  }
```

Lazy loading

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

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

Preventing unauthorized access

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

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

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

Create a service for your guard:

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

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

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


   }
```

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

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


   }
```

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

Router tutorial: tour of heroes

While the Getting Started: Tour of Heroes tutorial introduces general Angular concepts, this Router Tutorial goes into greater detail regarding Angular's routing capabilities. This tutorial guides you through building upon basic router configuration to create child routes, read route parameters, lazy load NgModules, guard routes, and preload data to improve the user experience.

Router tutorial overview

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 app 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.

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

The sample application in action

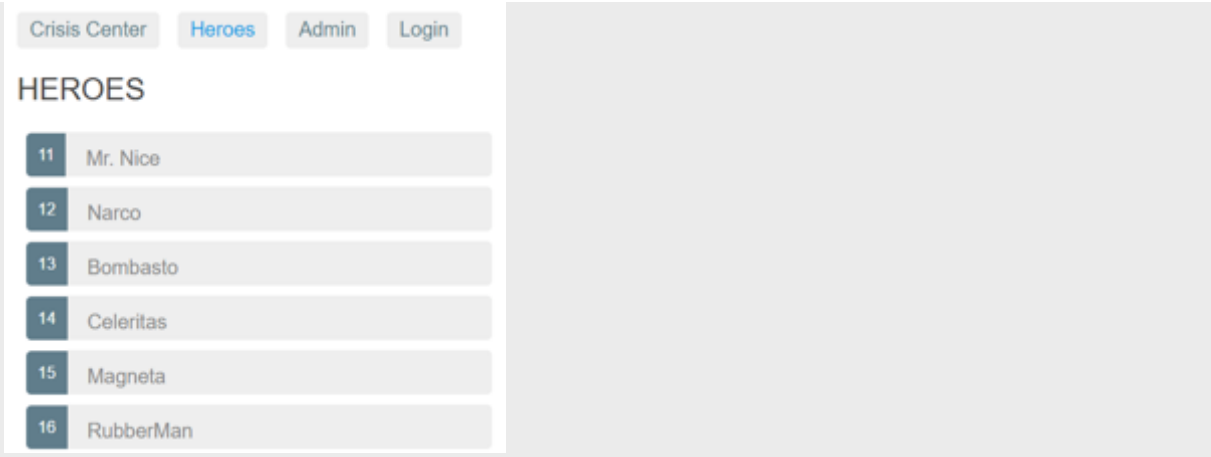
The sample application for this tutorial helps the Hero Employment Agency find crises for heroes to solve.

The application has three main feature areas:

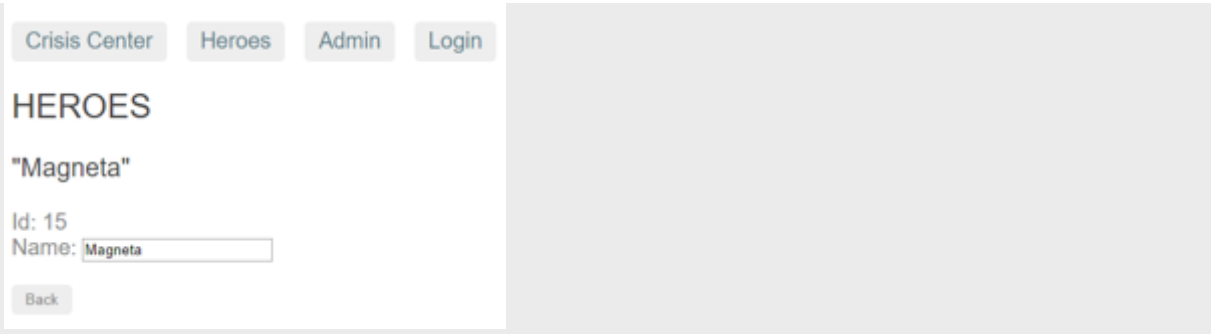
1. A *Crisis Center* for maintaining the list of crises for assignment to heroes.
2. A *Heroes* area for maintaining the list of heroes employed by the agency.
3. An *Admin* area to manage the list of crises and heroes.

Try it by clicking on this live example link / download example.

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



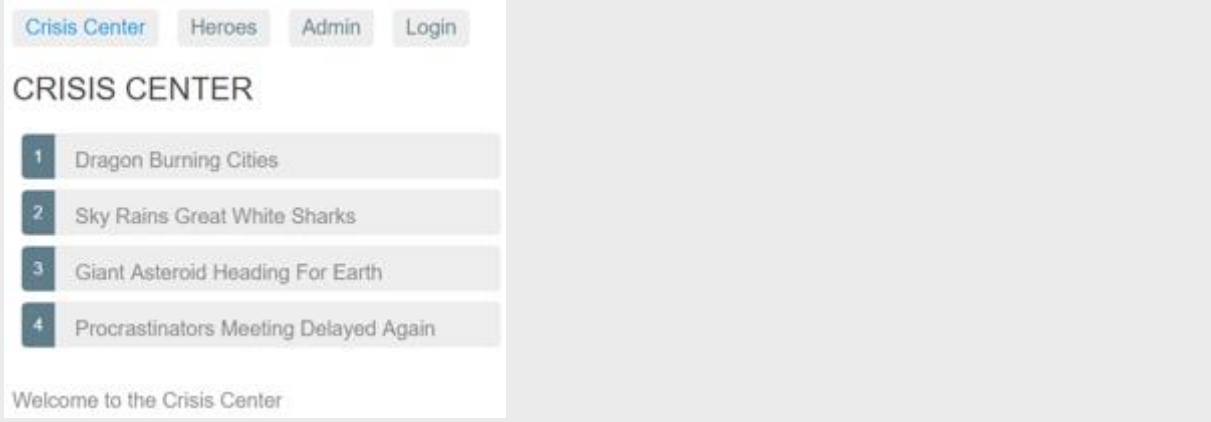
Select one hero and the app takes you to a hero editing screen.



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.

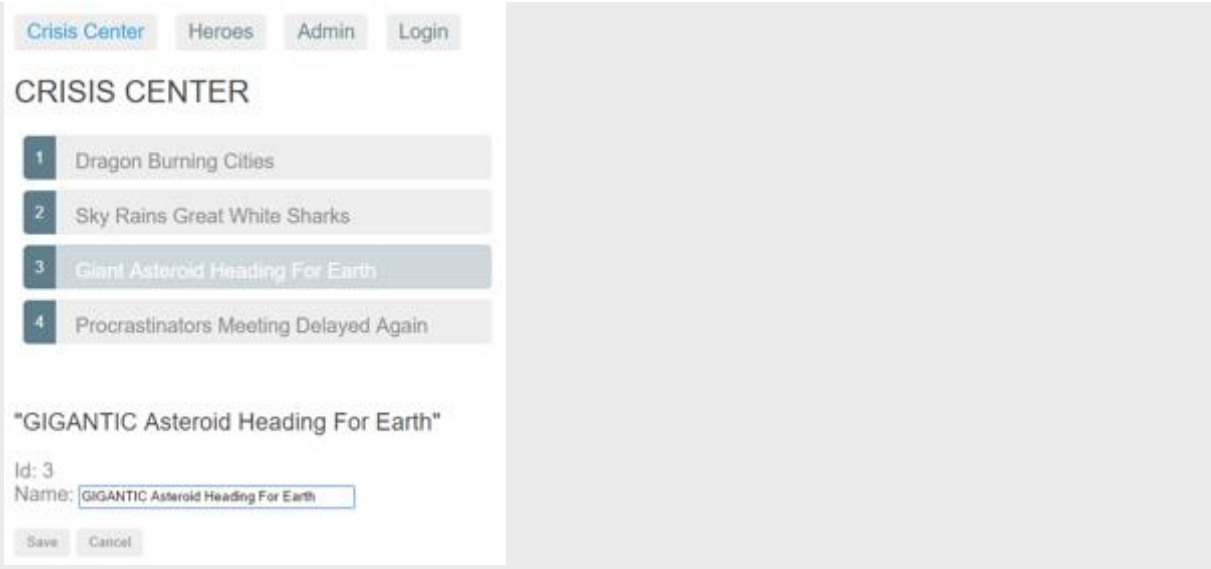
Had you clicked the browser's back button instead of the app's "Back" button, the app would have returned you to the heroes list as well. Angular app navigation updates the browser history as normal web navigation does.

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



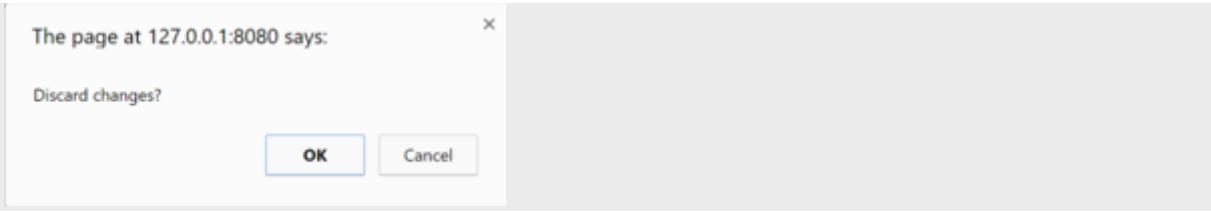
Select a crisis and the application takes you to a crisis editing screen. The *Crisis Detail* appears in a child component on the same page, beneath the list.

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.



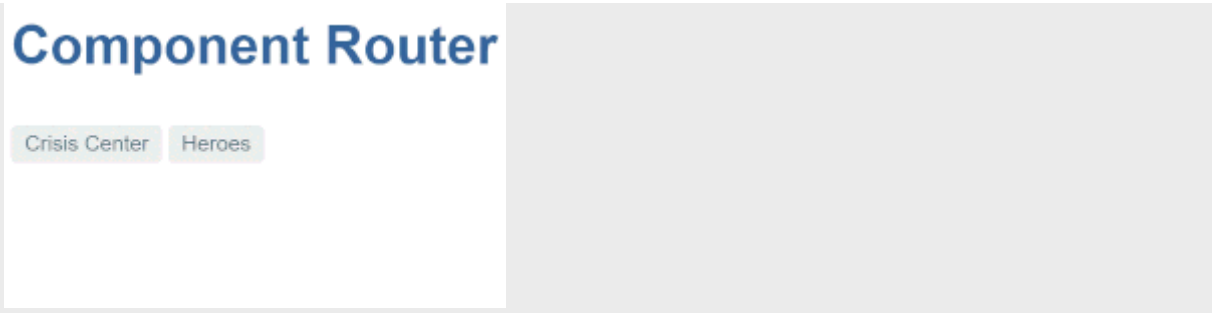
You can say "OK" and lose your changes or click "Cancel" and continue editing.

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.

Milestone 1: Getting started

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



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 sample HTML below.

src/app/crisis-list/crisis-list.component.htmlsrc/app/hero-list/hero-list.component.html

```
<h2>CRISIS CENTER</h2>

  <p>Get your crisis here</p>
```

Register `Router` and `Routes`

In order 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 containa 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.

src/app/app.module.ts (first-config)

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

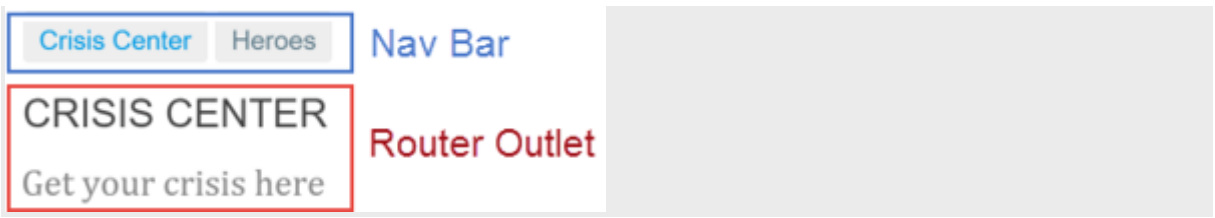
```
import { AppComponent }          from './app.component';
import { CrisisListComponent }   from './crisis-list/crisis-list.component';
import { HeroListComponent }     from './hero-list/hero-list.component';

const appRoutes: Routes = [
  { path: 'crisis-center', component: CrisisListComponent },
  { path: 'heroes', component: HeroListComponent },
];

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    RouterModule.forRoot(
      appRoutes,
      { enableTracing: true } // <-- debugging purposes only
    )
  ],
  declarations: [
    AppComponent,
    HeroListComponent,
    CrisisListComponent,
  ],
  bootstrap: [ AppComponent ]
})


   export class AppModule { }
```

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.

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.

The corresponding component template looks like this:
src/app/app.component.html

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


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

Define a 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. 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"`.
src/app/hero-list/hero-list.component.html (excerpt)

```
<h2>HEROES</h2>
<p>Get your heroes here</p>


  <button routerLink="/sidekicks">Go to sidekicks</button>
```

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`.

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

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

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

```
ng generate component page-not-found
```

src/app/page-not-found.component.html (404 component)

```
<h2>Page not found</h2>
```

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.

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 app is the list of heroes. The app 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 desired default path (`/heroes`).

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.
src/app/app-routing.module.ts (appRoutes)

```
const appRoutes: Routes = [
  { path: 'crisis-center', component: CrisisListComponent },
  { path: 'heroes',         component: HeroListComponent },
  { path: '',   redirectTo: '/heroes', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponent }


 ];
```

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'`.

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 app because if the `pathMatch` value were `'prefix'`, every URL would match `''`.

Try setting it to `'prefix'` and clicking the `Go to sidekicks` button. Since 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](http://vsavkin.tumblr.com/post/146722301646/angular-router-empty-paths-componentless-routes).

Milestone 1 wrap up

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

Milestone 1 has 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 app launches with an empty path.

The starter app's structure looks like this:
angular-router-sample

src

app

crisis-list

crisis-list.component.css

crisis-list.component.html

crisis-list.component.ts

hero-list

hero-list.component.css

hero-list.component.html

hero-list.component.ts

page-not-found

page-not-found.component.css

page-not-found.component.html

page-not-found.component.ts

app.component.css

app.component.html

app.component.ts

app.module.ts

main.ts

index.html

styles.css

tsconfig.json

node_modules ...

package.json

Here are the files in this milestone.

app.component.htmlapp.module.tshero-list/hero-list.component.htmlcrisis-list/crisis-list.component.htmlpage-not-found/page-not-found.component.htmlindex.html

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

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

Milestone 2: *Routing module*

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

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 app, and for each NgModule. When you create or initialize a new project (using the CLI `ng new` command) or a new app (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 app.

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.

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 just 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.
src/app/app-routing.module.ts

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

import { CrisisListComponent }   from './crisis-list/crisis-list.component';
import { HeroListComponent }     from './hero-list/hero-list.component';
import { PageNotFoundComponent } from './page-not-found/page-not-
found.component';

const appRoutes: Routes = [
  { path: 'crisis-center', component: CrisisListComponent },
  { path: 'heroes',        component: HeroListComponent },
  { path: '',   redirectTo: '/heroes', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponent }
];

@NgModule({
  imports: [
    RouterModule.forRoot(
      appRoutes,
      { enableTracing: true } // <-- debugging purposes only
    )
  ],
  exports: [
    RouterModule
  ]
})


   export class AppRoutingModule {}
```

Next, update the `app.module.ts` file by removing `RouterModule.forRoot` in the `imports` array.
src/app/app.module.ts

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

import { AppComponent }      from './app.component';
import { AppRoutingModule } from './app-routing.module';

import { CrisisListComponent }   from './crisis-list/crisis-list.component';
import { HeroListComponent }     from './hero-list/hero-list.component';
import { PageNotFoundComponent } from './page-not-found/page-not-
found.component';

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    AppRoutingModule
  ],
  declarations: [
    AppComponent,
    HeroListComponent,
    CrisisListComponent,
    PageNotFoundComponent
  ],
  bootstrap: [ AppComponent ]
})


   export class AppModule { }
```

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.

Benfits 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 app 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 apps 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.