Common Routing Tasks

This topic describes how to implement many of the common tasks associated with adding the Angular router to your application.

{@a basics} ## Generate an application with routing enabled

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

ng new routing-app –routing –defaults

Adding components for routing

To use the Angular router, an application 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`.

{@a basics-base-href}

`<base href>`

This guide works with a CLI-generated Angular application. 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:

import { FirstComponent } from ‘./first/first.component’; import { SecondComponent } from ‘./second/second.component’;

{@a basic-route}

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 application manually or working with an existing, non-CLI application, verify that the imports and configuration are correct. The following is the default `AppModule` using the CLI with the `--routing` flag.

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()`.

1. 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.

1. Add your routes to your application.

Now that you have defined your routes, 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.

{@a route-order}

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.

{@a getting-route-information}

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.

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.

```
<code-example path="router/src/app/heroes/hero-detail/hero-detail.component.ts" region="impo
</code-example>

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

  * [`Router`](api/router)
  * [`ActivatedRoute`](api/router/ActivatedRoute)
  * [`ParamMap`](api/router/ParamMap)
```

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

```
<code-example path="router/src/app/heroes/hero-detail/hero-detail.component.ts" region="acti
</code-example>
```

1. Update the `ngOnInit()` method to access the `ActivatedRoute` and track the `name` parameter:

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

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

{@a wildcard-route-how-to}

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.

{ 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.

{@a 404-page-how-to}

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:

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.

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`.

{@a nesting-routes}

Nesting routes

As your application grows more complex, you might 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`.

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.

{@ setting-the-page-title}

Setting the page title

Each page in your application should have a unique title so that they can be identified in the browser history. The `Router` sets the document's title using the `title` property from the `Route` config.

Note that the `title` property follows the same rules as static route `data` and dynamic values that implement `Resolve`.

You can also provide a custom title strategy by extending the `TitleStrategy`.

{@a using-relative-paths}

Using relative paths

Relative paths let you 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`, use the `../` notation to go up a level.

In addition to `../`, 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`.

The `navigate()` arguments configure the router to use the current route as a basis upon which to append `items`.

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 application 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.

import { ActivatedRoute } from ‘@angular/router’; import { Observable } from ‘rxjs’; import { switchMap } from ‘rxjs/operators’;

Next inject the activated route service:

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.

heroes$: Observable<Hero[]>; 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.

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:

hero$: Observable;

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

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

gotoItems(hero: Hero) { const heroId = hero ? hero.id : null; // Pass along the hero id if available // so that the HeroList component can select that item. this.router.navigate([‘/heroes’, { id: heroId }]); }

{@a lazy-loading}

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 application launches. Additionally, preload parts of your application 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 application 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.

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.

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

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

Link parameters array

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

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

Bind the `RouterLink` directive to such an array like this:

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

Provide optional route parameters in an object, as in `{ foo: 'foo' }`:

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

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

Note the following:

- The first item in the array identifies the parent route (`/crisis-center`).
- There are no parameters for this parent route.
- There is no default for the child route so you need to pick one.
- You're navigating to the `CrisisListComponent`, whose route path is `/`, but you don't need to explicitly add the slash.

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

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

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

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

{@a browser-url-styles}

{@a location-strategy}

`LocationStrategy` and browser URL styles

When the router navigates to a new component view, it updates the browser's location and history with a URL for that view.

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

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

localhost:3002/crisis-center/

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

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

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

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

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

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

Choosing a routing strategy

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

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

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

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

`<base href>`

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

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

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

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

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

HTML5 URLs and the `<base href>`

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

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

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

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

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

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

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

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

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

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

{@a hashlocationstrategy}

`HashLocationStrategy`

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