Routing

In web development, *routing* means splitting the application into different areas usually based on rules that are derived from the current URL in the browser.

For instance, if we visit the `/` path of a website, we may be visiting the **home route** of that website. Or if we visit `/about` we want to render the "about page", and so on.

Why Do We Need Routing?

Defining routes in our application is useful because we can:

- separate different areas of the app;
- maintain the state in the app;
- protect areas of the app based on certain rules;

For example, imagine we are writing an inventory application similar to the one we described in previous chapters.

When we first visit the application, we might see a search form where we can enter a search term and get a list of products that match that term.

After that, we might click a given product to visit that product's details page.

Because our app is client-side, it's not technically required that we change the URL when we change "pages". But it's worth thinking about for a minute: what would be the consequences of using the same URL for all pages?

- You wouldn't be able to refresh the page and keep your location within the app
- You wouldn't be able to bookmark a page and come back to it later
- You wouldn't be able to share the URL of that page with others

Or put in a positive light, routing lets us define a URL string that specifies where within our app a user should be.

In our inventory example we could determine a series of different routes for each activity, for instance:

The initial root URL could be represented by `http://our-app/`. When we visit this page, we could be redirected to our "home" route at `http://our-app/home`.

When accessing the 'About Us' area, the URL could become `http://our-app/about`. This way if we sent the URL `http://our-app/about` to another user they would see same page.

How client-side routing works

Perhaps you've written server-side routing code before (though, it isn't necessary to complete this chapter). Generally with server-side routing, the HTTP request comes in and the server will render a different controller depending on the incoming URL.

For instance, with Express.js[59] you might write something like this:

```
var express = require('express');
var router = express.Router();

// define the about route
router.get('/about', function(req, res) {
  res.send('About us');
});
```

Or with Ruby on Rails[60] you might have:

```
# routes.rb
get '/about', to: 'pages#about'

# PagesController.rb
class PagesController < ActionController::Base
  def about
    render
  end
end
```

The pattern varies per framework, but in both of these cases you have a **server** that accepts a request and *routes* to a **controller** and the controller runs a specific **action**, depending on the path and parameters.

Client-side routing is very similar in concept but different in implementation. With client-side routing **we're not necessarily making a request to the server** on every URL change. With our Angular apps, we refer to them as "Single Page Apps" (SPA) because our server only gives us a single page and it's our JavaScript that renders the different pages.

So how can we have different routes in our JavaScript code?

[59] http://expressjs.com/guide/routing.html
[60] http://rubyonrails.org/

The beginning: using anchor tags

Client-side routing started out with a clever hack: Instead of using a normal server-side URL for a page in our SPA, we use the *anchor tag* as the client-side URL.

As you may already know, anchor tags were traditionally used to link directly to a place *within* the webpage and make the browser scroll all the way to where that anchor was defined. For instance, if we define an anchor tag in an HTML page:

```
1  <!-- ... lots of page content here ... -->
2  <a name="about"><h1>About</h1></a>
```

And we visited the URL `http://something/#about`, the browser would jump straight to that H1 tag that identified by the `about` anchor.

The clever move for client-side frameworks used for SPAs was to take the anchor tags and use them represent the routes within the app by formatting them as paths.

For example, the `about` route for an SPA would be something like `http://something/#/about`. This is what is known as **hash-based routing**.

What's neat about this trick is that it looks like a "normal" URL because we're starting our anchor with a slash (`/about`).

The evolution: HTML5 client-side routing

With the introduction of HTML5, browsers acquired the ability to programmatically create new browser history entries that change the displayed URL *without the need for a new request.*

This is achieved using the `history.pushState` method that exposes the browser's navigational history to JavaScript.

So now, instead of relying on the anchor hack to navigate routes, modern frameworks can rely on `pushState` to perform history manipulation without reloads.

Angular 1 Note: This way of routing already works in Angular 1, but it needs to be explicitly enabled using `$locationProvider.html5Mode(true)`.

In Angular, however, the HTML5 is the default mode. Later in this chapter we show how to change from HTML5 mode to the old anchor tag mode.

There's two things you need to be aware of when using HTML5 mode routing, though

1. Not all browsers support HTML5 mode routing, so if you need to support older browsers you might be stuck with hash-based routing for a while.
2. **The server has to support HTML5 based routing**.

It may not be immediately clear why the server has to support HTML5 based-routing, we'll talk more about why later in this chapter.

Writing our first routes

The Angular docs recommends using HTML5 mode routing[61]. But due to the challenges mentioned in the previous section we will for simplicity be using hash based routing in our examples.

In Angular we configure routes by mapping *paths* to the component that will handle them.

Let's create a small app that has multiple routes. On this sample application we will have 3 routes:

- A main page route, using the `/#/home` path;
- An about page, using the `/#/about` path;
- A contact us page, using the `/#/contact` path;

And when the user visits the root path (`/#/`), it will redirect to the home path.

Components of Angular routing

There are three main components that we use to configure routing in Angular:

- `Routes` describes the routes our application supports
- `RouterOutlet` is a "placeholder" component that shows Angular where to put the content of each route
- `RouterLink` directive is used to link to routes

Let's look at each one more closely.

Imports

In order to use the router in Angular, we import constants from the `@angular/router` package:

[61] https://angular.io/docs/ts/latest/guide/router.html#!#browser-url-styles

code/routes/routing/src/app/app.module.ts

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

Now we can define our router configuration.

Routes

To define routes for our application, create a `Routes` configuration and then use `RouterModule.forRoot(routes)` to provide our application with the dependencies necessary to use the router. First, let's look at the routes definitions:

code/routes/routing/src/app/app.module.ts

```
const routes: Routes = [
  // basic routes
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent },
  { path: 'contactus', redirectTo: 'contact' },

  // authentication demo
  { path: 'login', component: LoginComponent },
  {
    path: 'protected',
    component: ProtectedComponent,
    canActivate: [ LoggedInGuard ]
  },

  // nested
  {
    path: 'products',
    component: ProductsComponent,
    children: childRoutes
  }
];
```

Notice a few things about the routes:

- `path` specifies the URL this route will handle
- `component` is what ties a given route path to a component that will handle the route
- the optional `redirectTo` is used to redirect a given path to an existing route

We'll dive into the details of each route in this chapter, but at a high-level, the goal of routes is to specify which component will handle a given path.

Redirections

When we use `redirectTo` on a route definition, it will tell the router that when we visit the `path` of the route, we want the browser to be redirected to another route.

In our sample code above, if we visit the `root` path at http://localhost:4200/#/[62], we'll be redirected to the route `home`.

Another example is the `contactus` route:

code/routes/routing/src/app/app.module.ts

```
32  { path: 'contactus', redirectTo: 'contact' },
```

In this case, if we visit the URL http://localhost:4200/#/contactus[63], we'll see that the browser redirects to `/contact`.

Sample Code The complete code for the examples in this section can be found in the `routes/routing` folder of the sample code. That folder contains a `README.md`, which gives instructions for building and running the project.

There are many different `imports` required for routing and we don't list every single one in every code example below. However we do list the filename and line number from which almost every example is taken from. If you're having trouble figuring out how to `import` a particular class, open up the code using your editor to see the entire code listing.

Try running the code while reading this section and feel free play around to get a deeper insight about how it all works.

Installing our Routes

Now that we have our `Routes` `routes`, we need to install it. To use the routes in our app we do two things to our `NgModule`:

1. Import the `RouterModule`
2. Install the routes using `RouterModule.forRoot(routes)` in the `imports` of our `NgModule`

Here's our routes configured into our `NgModule` for this app:

[62] http://localhost:4200/#/

[63] http://localhost:4200/#/contactus

code/routes/routing/src/app/app.module.ts

```
26  const routes: Routes = [
27    // basic routes
28    { path: '', redirectTo: 'home', pathMatch: 'full' },
29    { path: 'home', component: HomeComponent },
30    { path: 'about', component: AboutComponent },
31    { path: 'contact', component: ContactComponent },
32    { path: 'contactus', redirectTo: 'contact' },
```

code/routes/routing/src/app/app.module.ts

```
59    imports: [
60      BrowserModule,
61      FormsModule,
62      HttpModule,
63      RouterModule.forRoot(routes), // <-- routes
64
65      // added this for our child module
66      ProductsModule
67    ],
```

`RouterOutlet` using `<router-outlet>`

When we change routes, we want to keep our outer "layout" template and only substitute the "inner section" of the page with the route's component.

In order to describe to Angular where in our page we want to render the contents for each route, we use the `RouterOutlet` directive.

Our component `@Component` has a `template` which specifies some `div` structure, a section for `Navigation`, and a directive called `router-outlet`.

The `router-outlet` element indicates where the contents of each route component will be rendered.

We are are able to use the `router-outlet` directive in our template because we imported the `RouterModule` in our `NgModule`.

Here's the component and template for the navigation wrapper of our app:

code/routes/routing/src/app/app.component.ts

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  constructor(private router: Router) {
  };
}
```

and the template:

code/routes/routing/src/app/app.component.html

```
<div class="page-header">
  <div class="container">
    <h1>Router Sample</h1>
    <div class="navLinks">
      <a [routerLink]="['/home']">Home</a>
      <a [routerLink]="['/about']">About Us</a>
      <a [routerLink]="['/contact']">Contact Us</a>
      |
      <a [routerLink]="['/products']">Products</a>
      <a [routerLink]="['/login']">Login</a>
      <a [routerLink]="['/protected']">Protected</a>
    </div>
  </div>
</div>

<div id="content">
  <div class="container">
    <router-outlet></router-outlet>
  </div>
</div>
```

If we look at the template above, you will note the `router-outlet` element right below the navigation menu. When we visit `/home`, that's where `HomeComponent` template will be rendered. The same happens for the other components.

`RouterLink` using `[routerLink]`

Now that we know where route templates will be rendered, how do we tell Angular to navigate to a given route?

We might try linking to the routes directly using pure HTML:

```
<a href="/#/home">Home</a>
```

But if we do this, we'll notice that clicking the link triggers a page reload and that's definitely not what we want when programming single page apps.

To solve this problem, Angular provides a solution that can be used to link to routes **with no page reload**: the `RouterLink` directive.

This directive allows you to write links using a special syntax:

code/routes/routing/src/app/app.component.html

```
    <h1>Router Sample</h1>
    <div class="navLinks">
      <a [routerLink]="['/home']">Home</a>
      <a [routerLink]="['/about']">About Us</a>
      <a [routerLink]="['/contact']">Contact Us</a>
      |
```

We can see on the left-hand side the `[routerLink]` that applies the directive to the current element (in our case `a` tags).

Now, on the right-hand side we have an array with the route path as the first element, like `"['/home']"` or `"['/about']"` that will indicate which route to navigate to when we click the element.

It might seem a little odd that the value of `routerLink` is a string with an array containing a string (`"['/home']"`, for example). This is because there are more things you can provide when linking to routes, but we'll look at this into more detail when we talk about child routes and route parameters.

For now, we're only using routes names from the root app component.

Putting it all together

So now that we have all the basic pieces, let's make them work together to transition from one route to the other.

The first thing we need to write for our application is the `index.html` file.

Here's the full code for that:

code/routes/routing/src/index.html

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>Routing</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root>Loading...</app-root>
</body>
</html>
```

The code should be familiar by now, with the exception of this line:

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

This line declares the `base` HTML tag. This tag is traditionally used to tell the browser where to look for images and other resources declared using relative paths.

It turns out Angular Router also relies on this tag to determine how to construct its routing information.

For instance, if we have a route with a path of `/hello` and our `base` element declares `href="/app"`, the application will use `/app/#` as the concrete path.

Sometimes though, coders of an Angular application don't have access to the `head` section of the application HTML. This is true for instance, when reusing headers and footers of a larger, pre-existing application.

Fortunately there is a workaround for this case. You can declare the application base path programmatically, when configuring our `NgModule` by using the `APP_BASE_HREF` provider:

```
@NgModule({
  declarations: [ RoutesDemoApp ],
  imports: [
    BrowserModule,
    RouterModule.forRoot(routes) // <-- routes
  ],
  bootstrap: [ RoutesDemoApp ],
  providers: [
    { provide: LocationStrategy, useClass: HashLocationStrategy },
    { provide: APP_BASE_HREF, useValue: '/' } // <--- this right here
  ]
})
```

Putting `{ provide: APP_BASE_HREF, useValue: '/' }` in the `providers` is the equivalent of using `<base href="/">` on our application HTML header.

When deploying to production we can also set the value of the base-href by using the `--base-href` command-line option

Creating the Components

Before we get to the main app component, let's create 3 simple components, one for each of the routes.

HomeComponent

The `HomeComponent` will just have an `h1` tag that says "`Welcome!`". Here's the full code for our `HomeComponent`:

code/routes/routing/src/app/home/home.component.ts

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.css']
})
export class HomeComponent implements OnInit {

  constructor() { }
```

```
11
12    ngOnInit() {
13    }
14
15  }
```

And template:

code/routes/routing/src/app/home/home.component.html

```
1  <h1>Welcome Home!</h1>
```

AboutComponent

Similarly, the `AboutComponent` will just have a basic `h1`:

code/routes/routing/src/app/about/about.component.ts

```
1  import { Component, OnInit } from '@angular/core';
2
3  @Component({
4    selector: 'app-about',
5    templateUrl: './about.component.html',
6    styleUrls: ['./about.component.css']
7  })
8  export class AboutComponent implements OnInit {
9
10   constructor() { }
11
12   ngOnInit() {
13   }
14
15 }
```

And template:

code/routes/routing/src/app/about/about.component.html

```
1  <h1>About Us</h1>
```

ContactComponent

And, likewise with `AboutComponent`:

code/routes/routing/src/app/contact/contact.component.ts

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-contact',
  templateUrl: './contact.component.html',
  styleUrls: ['./contact.component.css']
})
export class ContactComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }

}
```

And template:

code/routes/routing/src/app/contact/contact.component.html

```
<h1>Contact Us</h1>
```

Nothing really very interesting about those components, so let's move on to the main `app.module.ts` file.

Application Component

Now we need to create the root-level "application" component that will tie everything together.

We start with the imports we'll need, both from the `core` and `router` bundles:

code/routes/routing/src/app/app.module.ts

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

Next step is to import the three components we created above:

code/routes/routing/src/app/app.module.ts

```
15 import { AppComponent } from './app.component';
16 import { HomeComponent } from './home/home.component';
17 import { ContactComponent } from './contact/contact.component';
18 import { AboutComponent } from './about/about.component';
```

For our root component, we're going to use two router directives: `RouterOutlet` and the `RouterLink`. Those directives, along with all other common router directives are imported when we put `RouterModule` in the `imports` section of our `NgModule`.

As a recap, the `RouterOutlet` directive is then used to indicate where in our template the route contents should be rendered. That's represented by the `<router-outlet></router-outlet>` snippet in our `AppComponent` template.

The `RouterLink` directive is used to create navigation links to our routes:

code/routes/routing/src/app/app.component.html

```
1  <div class="page-header">
2    <div class="container">
3      <h1>Router Sample</h1>
4      <div class="navLinks">
5        <a [routerLink]="['/home']">Home</a>
6        <a [routerLink]="['/about']">About Us</a>
7        <a [routerLink]="['/contact']">Contact Us</a>
8        |
9        <a [routerLink]="['/products']">Products</a>
10       <a [routerLink]="['/login']">Login</a>
11       <a [routerLink]="['/protected']">Protected</a>
12     </div>
13   </div>
14 </div>
15
16 <div id="content">
17   <div class="container">
18     <router-outlet></router-outlet>
19   </div>
20 </div>
```

Using `[routerLink]` will instruct Angular to take ownership of the `click` event and then initiate a route switch to the right place, based on the route definition.

Configuring the Routes

Next, we declare the routes creating an array of objects that conform to the `Routes` type:

code/routes/routing/src/app/app.module.ts

```
26 const routes: Routes = [
27   // basic routes
28   { path: '', redirectTo: 'home', pathMatch: 'full' },
29   { path: 'home', component: HomeComponent },
30   { path: 'about', component: AboutComponent },
31   { path: 'contact', component: ContactComponent },
32   { path: 'contactus', redirectTo: 'contact' },
```

code/routes/routing/src/app/app.module.ts

```
50 @NgModule({
51   declarations: [
52     AppComponent,
53     HomeComponent,
54     ContactComponent,
55     AboutComponent,
56     LoginComponent,
57     ProtectedComponent,
58   ],
59   imports: [
60     BrowserModule,
61     FormsModule,
62     HttpModule,
63     RouterModule.forRoot(routes), // <-- routes
64
65     // added this for our child module
66     ProductsModule
67   ],
68   providers: [
69     // uncomment this for "hash-bang" routing
70     // { provide: LocationStrategy, useClass: HashLocationStrategy }
71     AUTH_PROVIDERS,
72     LoggedInGuard
73   ],
74   bootstrap: [AppComponent]
75 })
76 export class AppModule { }
```

Notice that we put all necessary components in our `declarations`. If we're going to route to a component, then it needs to be declared in *some* `NgModule` (either this module or imported).

In our `imports` we have `RouterModule.forRoot(routes)`. `RouterModule.forRoot(routes)` is a function that will take our routes, configure the router, and return a list of dependencies like `RouteRegistry`, `Location`, and several other classes that are necessary to make routing work.

In our `providers` we have this:

```
1     { provide: LocationStrategy, useClass: HashLocationStrategy }
```

Let's take an in depth look of what we want to achieve with this line.

Routing Strategies

The way the Angular application parses and creates paths from and to route definitions is called *location strategy*.

In Angular 1 this is called *routing modes* instead

The default strategy is `PathLocationStrategy`, which is what we call HTML5 routing. While using this strategy, routes are represented by regular paths, like `/home` or `/contact`.

We can change the location strategy used for our application by binding the `LocationStrategy` class to a new, concrete strategy class.

Instead of using the default `PathLocationStrategy` we can also use the `HashLocationStrategy`.

The reason we're using the hash strategy as a default is because if we were using HTML5 routing, our URLs would end up being regular paths (that is, not using hash/anchor tags).

This way, the routes would work when you click a link and navigate on the client side, let's say from `/about` to `/contact`.

If we were to refresh the page, instead of asking the server for the root URL, which is what is being served, instead we'd be asking for `/about` or `/contact`. Because there's no known page at `/about` the server would return a 404.

This default strategy works with hash based paths, like `/#/home` or `/#/contact` that the server understands as being the `/` path. (This is also the default mode in Angular 1.)

Let's say you want to use HTML5 mode in production, how do you set this up?

In order to use HTML5 mode routing, you have to configure your server to redirect every "missing" route to the root URL.

Angular CLI supports this natively, but know that it doesn't necessarily work by default on your server. In the `routes/routing` project you can use HTML5 routes by simply doing `ng serve`

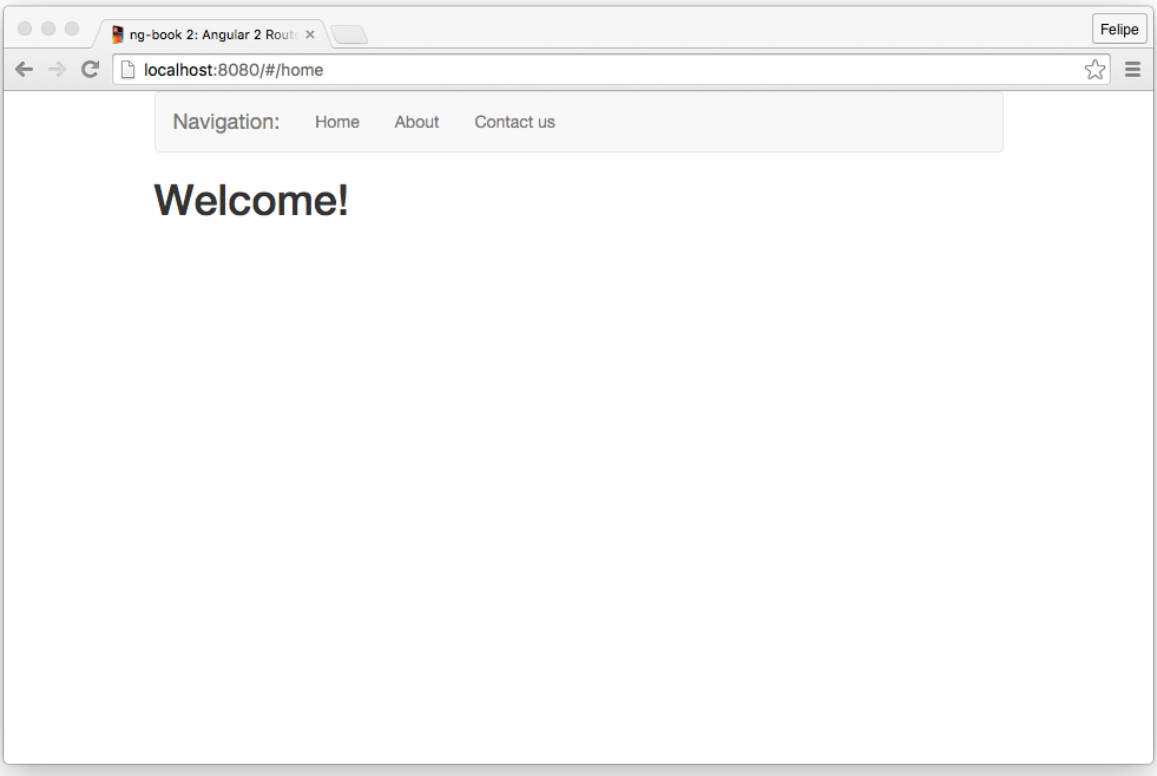
If we wanted to make our example application work with this new strategy, first we have to import `LocationStrategy` and `HashLocationStrategy` and then add that location strategy to the providers of our `NgModule`.

You could write your own strategy if you wanted to. All you need to do is extend the `LocationStrategy` class and implement the methods. A good way to start is reading the Angular source for the `HashLocationStrategy` or `PathLocationStrategy` classes.

Running the application

You can now go into the application root folder (`code/routes/routing`) and run `npm start` to boot the application.

When you type http://localhost:4200/[64] into your browser you should see the home route rendered:

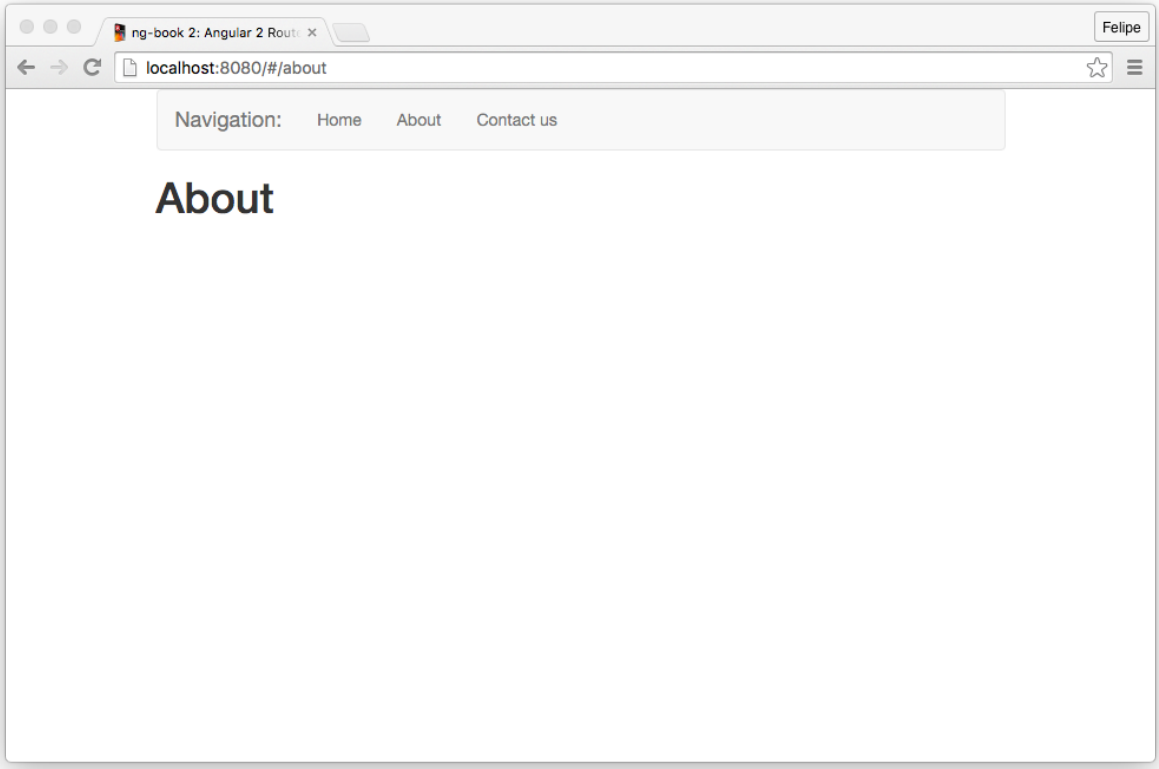


Home Route

[64] http://localhost:4200/

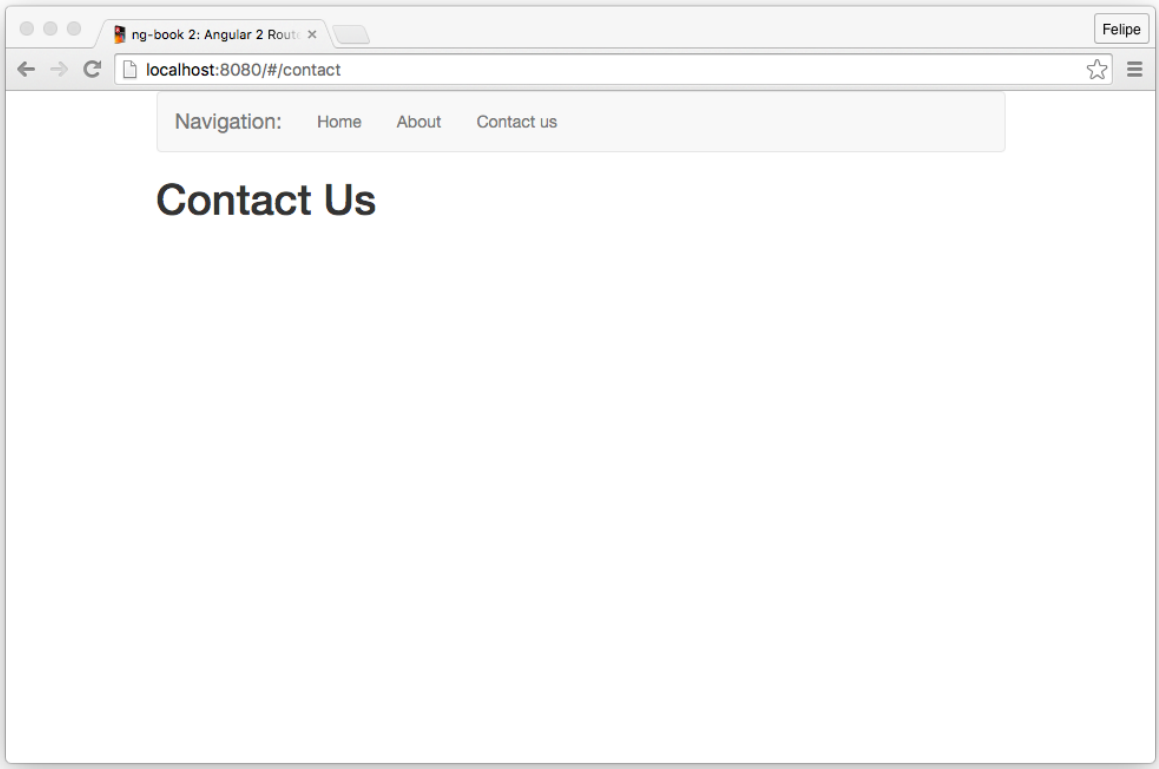
Notice that the URL in the browser was redirected to http://localhost:4200/home[65].

Now clicking each link will render the appropriate routes:



About Route

[65] http://localhost:4200/home



Contact Us Route

Route Parameters

In our apps we often want to navigate to a specific resource. For instance, say we had a news website and we had many articles. Each article may have an ID, and if we had an article with ID `3` then we might navigate to that article by visiting the URL:

```
/articles/3
```

And if we had an article with an ID of `4` we would access it at

```
/articles/4
```

and so on.

Obviously we're not going to want to write a route for each article, but instead we want to use a variable, or *route parameter*. We can specify that a route takes a parameter by putting a colon `:` in front of the path segment like this:

```
/route/:param
```

So in our example news site, we might specify our route as:

`/product/:id`

To add a parameter to our router configuration, we specify the route path like this:

```
const routes: Routes = [
  { path: 'product/:id', component: ProductComponent },
];
```

When we visit the route `/product/123`, the `123` part will be passed as the `id` route parameter to our route.

But how can we retrieve the parameter for a given route? That's where we use route parameters.

ActivatedRoute

In order to use route parameters, we need to first import `ActivatedRoute`:

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

Next, we inject the `ActivatedRoute` into the constructor of our component. For example, let's say we have a `Routes` that specifies the following:

```
const routes: Routes = [
  { path: 'product/:id', component: ProductComponent }
];
```

Then when we write the `ProductComponent`, we add the `ActivatedRoute` as one of the constructor arguments:

```
export class ProductComponent {
  id: string;

  constructor(private route: ActivatedRoute) {
    route.params.subscribe(params => { this.id = params['id']; });
  }
}
```

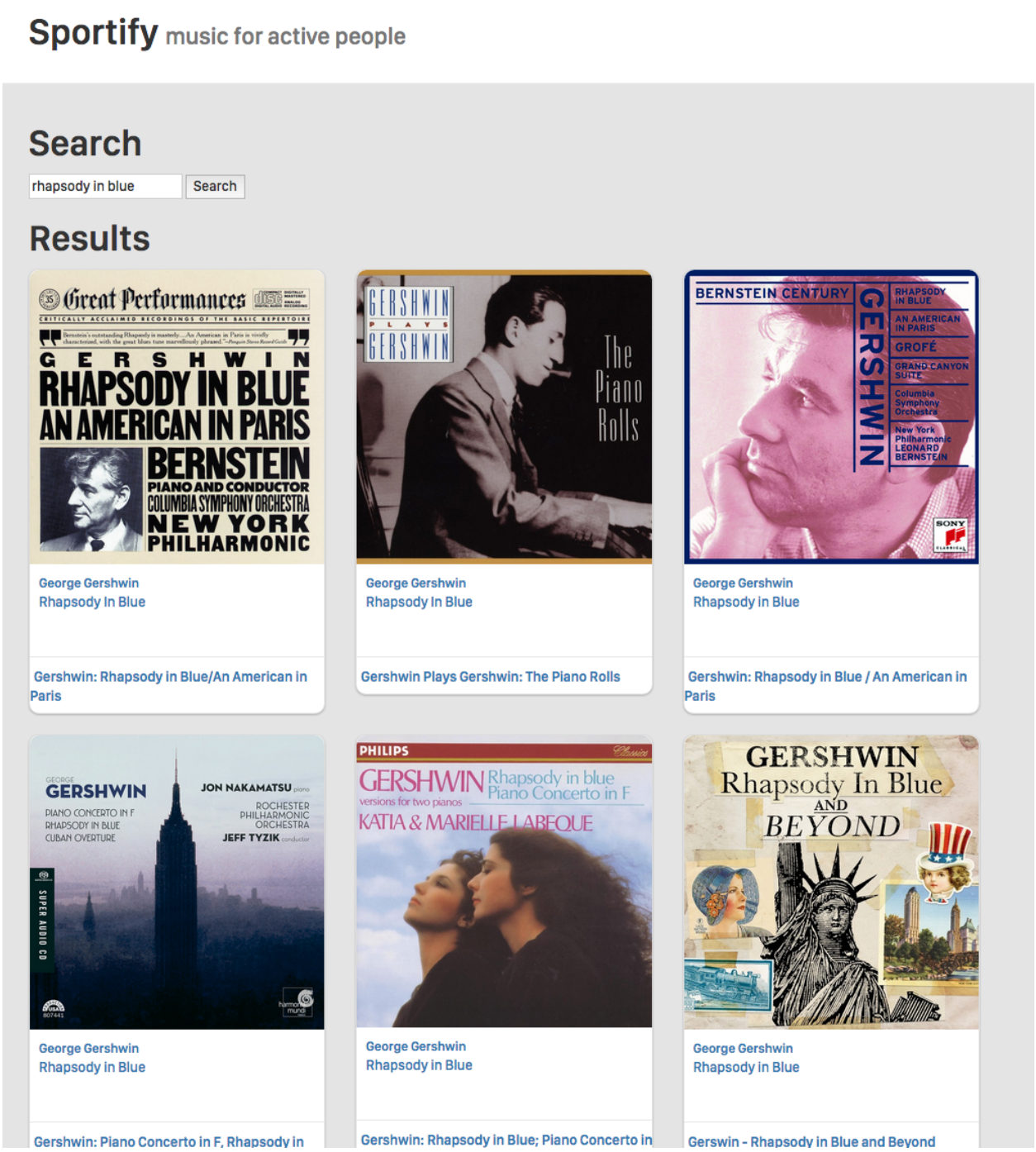
Notice that `route.params` is an *observable*. We can extract the value of the param into a hard value by using `.subscribe`. In this case, we assign the value of `params['id']` to the `id` instance variable on the component.

Now when we visit `/product/230`, our component's `id` attribute will receive `230`.

Music Search App

Let's now work on a more complex application. We will build a music search application that has the following features:

1. **Search for tracks** that match a given term
2. Show **matching tracks** in a grid
3. Show **artist details** when the artist name is clicked
4. Show **album details** and show a list of tracks when the album name is clicked
5. Show **song details** allow the user to **play a preview** when the song name is clicked



The Search View of our Music App

The routes we will need for this application will be:

- `/search` - search form and results
- `/artists/:id` - artist info, represented by a Spotify ID

- `/albums/:id` - album info, with a list of tracks using the Spotify ID
- `/tracks/:id` - track info and preview, also using the Spotify ID

Sample Code The complete code for the examples in this section can be found in the `routes/music` folder of the sample code. That folder contains a `README.md`, which gives instructions for building and running the project.

We will use the Spotify API[66] to get information about tracks, artists and albums.

First Steps

The first file we need work on is `app.module.ts`. Let's start by importing classes we'll use from Angular:

code/routes/music/src/app/app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';
import {
  RouterModule,
  Routes
} from '@angular/router';
import {
  LocationStrategy,
  HashLocationStrategy,
  APP_BASE_HREF
} from '@angular/common';

import { AppComponent } from './app.component';
import { AlbumComponent } from './album/album.component';
import { ArtistComponent } from './artist/artist.component';
```

Now that we have the imports there, let's think about the components we'll use for each route.

- For the `Search` route, we'll create a `SearchComponent`. This component will talk to the Spotify API to perform the search and then display the results on a grid.

[66] https://developer.spotify.com/web-api

- For the `Artists` route, we'll create an `ArtistComponent` which will show the artist's information
- For the `Albums` route, we'll create an `AlbumComponent` which will show the list of tracks in the album
- For the `Tracks` route, we'll create a `TrackComponent` which will show the track and let us play a preview of the song

Since this new component will need to interact with the Spotify API, it seems like we need to build a service that uses the `http` module to call out to the API server.

Everything in our app depends on the data, so let's build the `SpotifyService` first.

The `SpotifyService`

You can find the full code for the final version of the `SpotifyService` in the `routes/music/src/app` folder of the sample code.

The first method we'll implement is `searchTrack` which will search for track, given a search term.

One of the endpoints documented on Spotify API docs is the Search endpoint[67].

This endpoint does exactly what we want: it takes a query (using the `q` parameter) and a `type` parameter.

Query in this case is the search term. And since we're searching for songs, we should use `type=track`.

Here's what a first version of the service could look like:

```
1  class SpotifyService {
2    constructor(public http: Http) {
3    }
4
5    searchTrack(query: string) {
6      let params: string = [
7        `q=${query}`,
8        `type=track`
9      ].join("&");
10     let queryURL: string = `https://api.spotify.com/v1/search?${params}`;
11     return this.http.request(queryURL).map(res => res.json());
12   }
13 }
```

[67] https://developer.spotify.com/web-api/search-item/

This code performs an HTTP `GET` request to the URL https://api.spotify.com/v1/search[68], passing our `query` as the search term and `type` hardcoded to `track`.

This `http` call returns an `Observable`. We are going one step further and using the RxJS function `map` to transform the result we would get (which is an `http` module's `Response` object) and parsing it as JSON, resulting on an object.

Any function that calls `searchTrack` then will have to use the `Observable` API to subscribe to the response like this:

```
service
  .searchTrack('query')
  .subscribe((res: any) => console.log('Got object', res))
```

The `SearchComponent`

Now that we have a service that will perform track searches, we can start coding the `SearchComponent`.

Again, we start with an import section:

code/routes/music/src/app/search/search.component.ts

```
/*
 * Angular
 */

import {Component, OnInit} from '@angular/core';
import {
  Router,
  ActivatedRoute,
} from '@angular/router';

/*
 * Services
 */
import {SpotifyService} from '../spotify.service';
```

Here we're importing, among other things, the `SpotifyService` class we just created.

The goal here is to render each resulting track side by side on a card like below:

[68] https://api.spotify.com/v1/search

Music App Card

We then start coding the component. We're using `search` as the selector, making a few imports and using the following template. The template is a bit long because we're putting some reasonable styles on it using the CSS framework Bootstrap[69], but it isn't particularly complicated, relative to what we've done so far:

code/routes/music/src/app/search/search.component.html

```
<h1>Search</h1>

<p>
  <input type="text" #newquery
    [value]="query"
    (keydown.enter)="submit(newquery.value)">
  <button (click)="submit(newquery.value)">Search</button>
</p>

<div *ngIf="results">
  <div *ngIf="!results.length">
    No tracks were found with the term '{{ query }}'
```

[69] http://getbootstrap.com

```
13    </div>
14
15    <div *ngIf="results.length">
16      <h1>Results</h1>
17
18      <div class="row">
19        <div class="col-sm-6 col-md-4" *ngFor="let t of results">
20          <div class="thumbnail">
21            <div class="content">
22              <img src="{{ t.album.images[0].url }}" class="img-responsive">
23              <div class="caption">
24                <h3>
25                  <a [routerLink]="['/artists', t.artists[0].id]">
26                    {{ t.artists[0].name }}
27                  </a>
28                </h3>
29                <br>
30                <p>
31                  <a [routerLink]="['/tracks', t.id]">
32                    {{ t.name }}
33                  </a>
34                </p>
35              </div>
36              <div class="attribution">
37                <h4>
38                  <a [routerLink]="['/albums', t.album.id]">
39                    {{ t.album.name }}
40                  </a>
41                </h4>
42              </div>
43            </div>
44          </div>
45        </div>
46      </div>
47    </div>
48  </div>
```

The Search Field

Let's break down the HTML template a bit.

This first section will have the search field:

code/routes/music/src/app/search/search.component.html

```
<p>
  <input type="text" #newquery
    [value]="query"
    (keydown.enter)="submit(newquery.value)">
  <button (click)="submit(newquery.value)">Search</button>
</p>
```

Here we have the input field and we're binding its DOM element `value` property the `query` property of our component.

We also give this element a template variable named `#newquery`. We can now access the value of this input within our template code by using `newquery.value`.

The button will trigger the `submit` method of the component, passing the value of the input field as a parameter.

We also want to trigger `submit` when the user hits "Enter" so we bind to the `keydown.enter` event on the input.

Search Results and Links

The next section displays the results. We're relying on the `NgFor` directive to iterate through each track from our results object:

code/routes/music/src/app/search/search.component.html

```
    <div class="row">
      <div class="col-sm-6 col-md-4" *ngFor="let t of results">
        <div class="thumbnail">
```

For each track, we display the artist name:

code/routes/music/src/app/search/search.component.html

```
              <h3>
                <a [routerLink]="['/artists', t.artists[0].id]">
                  {{ t.artists[0].name }}
                </a>
              </h3>
```

Notice how we're using the `RouterLink` directive to redirect to `['/artists', t.artists[0].id]`.

This is how we set *route parameters* for a given route. Say we have an artist with an id `abc123`. When this link is clicked, the app would then navigate to `/artist/abc123` (where `abc123` is the `:id` parameter).

Further down we'll show how we can retrieve this value inside the component that handles this route.

Now we display the track:

code/routes/music/src/app/search/search.component.html

```
30              <p>
31                <a [routerLink]="['/tracks', t.id]">
32                  {{ t.name }}
33                </a>
34              </p>
```

And the album:

code/routes/music/src/app/search/search.component.html

```
38                <a [routerLink]="['/albums', t.album.id]">
39                  {{ t.album.name }}
40                </a>
41              </h4>
```

`SearchComponent` Class

Let's take a look at the constructor first:

code/routes/music/src/app/search/search.component.ts

```
22 export class SearchComponent implements OnInit {
23   query: string;
24   results: Object;
25
26   constructor(private spotify: SpotifyService,
27               private router: Router,
28               private route: ActivatedRoute) {
29     this.route
30       .queryParams
31       .subscribe(params => { this.query = params['query'] || ''; });
32   }
```

Here we're declaring two properties:

- `query` for current search term and
- `results` for the search results

On the `constructor` we're injecting the `SpotifyService` (that we created above), `Router`, and the `ActivatedRoute` and making them properties of our class.

In our `constructor` we `subscribe` to the `queryParams` property - this lets us access *query parameters*, such as the search term (`params['query']`).

In a URL like: `http://localhost/#/search?query=cats&order=ascending`, `queryParams` gives us the parameters in an object. This means we could access the `order` with `params['order']` (in this case, `ascending`).

Also note that `queryParams` are different than `route.params`. Whereas `route.params` match parameters in the *route* `queryParams` match parameters in the query string.

In this case, if there is no `query` param, we set `this.query` to the empty string.

search

In our `SearchComponent` we will call out to the `SpotifyService` and render the results. There are two cases when we want to run a search:

We want to run a search when the user:

- enters a search query and submits the form
- navigates to this page with a given URL in the query parameters (e.g. someone shared a link or bookmarked the page)

To perform the actual search for both cases, we create the `search` method:

code/routes/music/src/app/search/search.component.ts

```
43   search(): void {
44     console.log('this.query', this.query);
45     if (!this.query) {
46       return;
47     }
48
49     this.spotify
50       .searchTrack(this.query)
51       .subscribe((res: any) => this.renderResults(res));
52   }
```

The `search` function uses the current value of `this.query` to know what to search for. Because we subscribed to the `queryParams` in the constructor, we can be sure that `this.query` will always have the most up-to-date value.

We then subscribe to the `searchTrack Observable` and whenever new results are emitted we call `renderResults`.

code/routes/music/src/app/search/search.component.ts

```
54  renderResults(res: any): void {
55    this.results = null;
56    if (res && res.tracks && res.tracks.items) {
57      this.results = res.tracks.items;
58    }
59  }
```

We declared `results` as a component property. Whenever its value is changed, the view will be automatically updated by Angular.

Searching on Page Load

As we pointed out above, we want to be able to jump straight into the results if the URL includes a search query.

To do that, we are going to implement a hook Angular router provides for us to run whenever our component is initialized.

But isn't that what constructors are for? Well, yes and no. Yes, constructors are used to initialize values, but if you want to write good, testable code, you want to minimize the side effects of *constructing* an object. So keep in mind that you should put your component's initialization login always on a hook like below.

Here's the implementation of the `ngOnInit` method:

code/routes/music/src/app/search/search.component.ts

```
34  ngOnInit(): void {
35    this.search();
36  }
```

> To use `ngOnInit` we imported the `OnInit` class and declared that our component `implements OnInit`.

As you can see, we're just performing the search here. Since the term we're searching for comes from the URL, we're good.

submit

Now let's see what we do when the user submits the form.

code/routes/music/src/app/search/search.component.ts

```
38   submit(query: string): void {
39     this.router.navigate(['search'], { queryParams: { query: query } })
40       .then(_ => this.search() );
41   }
```

We're manually telling the router to navigate to the search route, and providing a `query` parameter, then performing the actual search.

Doing things this way gives us a great benefit: if we reload the browser, we're going to see the same search result rendered. We can say that we're **persisting the search term on the URL**.

Putting it all together

Here's the full listing for the `SearchComponent` class:

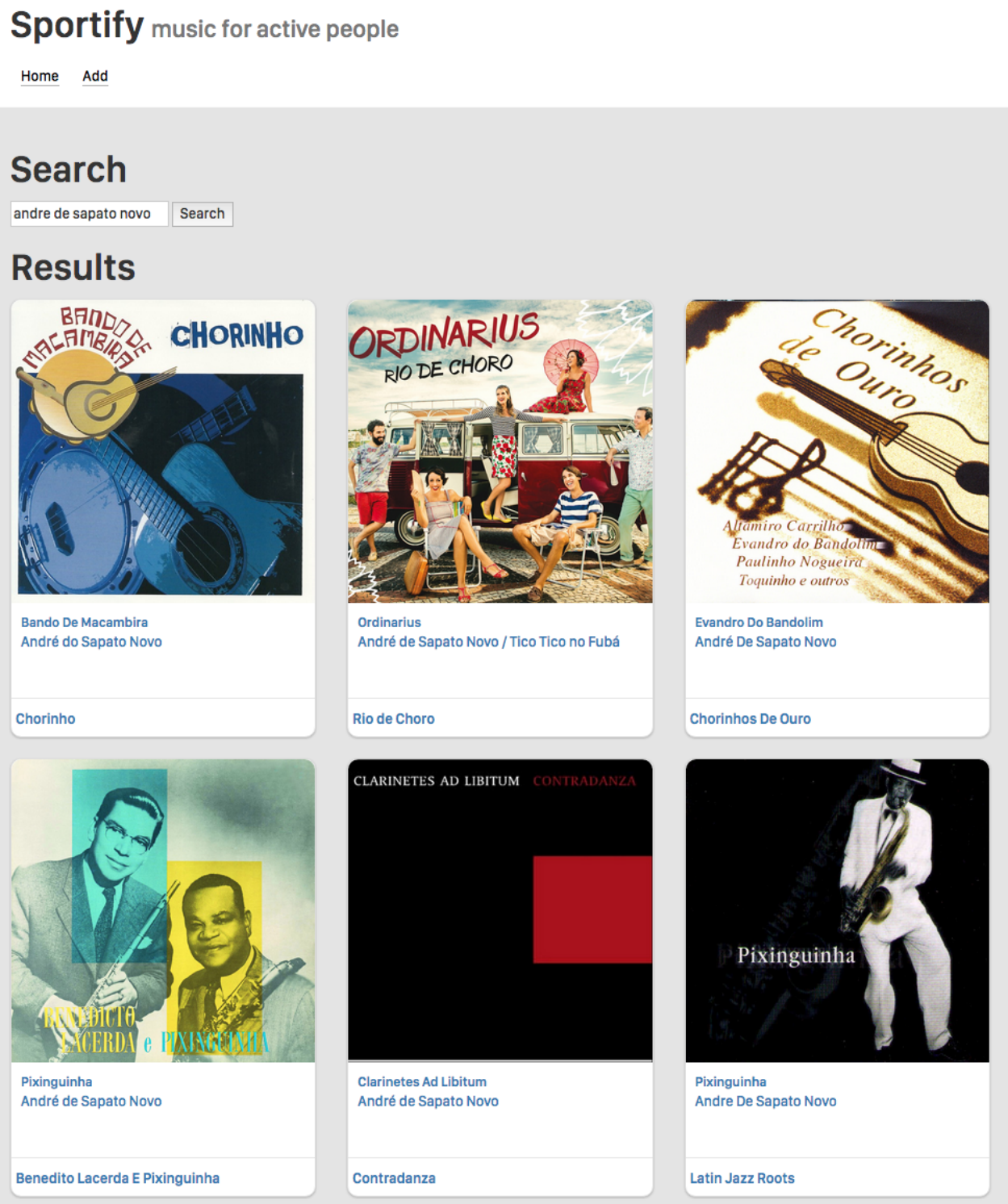
code/routes/music/src/app/search/search.component.ts

```
1  /*
2   * Angular
3   */
4
5  import {Component, OnInit} from '@angular/core';
6  import {
7    Router,
8    ActivatedRoute,
9  } from '@angular/router';
10
11 /*
12  * Services
13  */
14 import {SpotifyService} from '../spotify.service';
15 ;
16
17 @Component({
18   selector: 'app-search',
19   templateUrl: './search.component.html',
20   styleUrls: ['./search.component.css']
21 })
```

```
export class SearchComponent implements OnInit {
  query: string;
  results: Object;

  constructor(private spotify: SpotifyService,
              private router: Router,
              private route: ActivatedRoute) {
    this.route
      .queryParams
      .subscribe(params => { this.query = params['query'] || ''; });
  }

  ngOnInit(): void {
    this.search();
  }

  submit(query: string): void {
    this.router.navigate(['search'], { queryParams: { query: query } })
      .then(_ => this.search() );
  }

  search(): void {
    console.log('this.query', this.query);
    if (!this.query) {
      return;
    }

    this.spotify
      .searchTrack(this.query)
      .subscribe((res: any) => this.renderResults(res));
  }

  renderResults(res: any): void {
    this.results = null;
    if (res && res.tracks && res.tracks.items) {
      this.results = res.tracks.items;
    }
  }
}
```

Trying the search

Now that we have completed the code for the search, let’s try it out:



Trying out Search

We can click the artist, track or album links to navigate to the proper route.

TrackComponent

For the track route, we use the `TrackComponent`. It basically displays the track name, the album cover image and allow the user to play a preview using an HTML5 `audio` tag:

code/routes/music/src/app/track/track.component.html

```
<div *ngIf="track">
  <h1>{{ track.name }}</h1>

  <p>
    <img src="{{ track.album.images[1].url }}">
  </p>

  <p>
    <audio controls src="{{ track.preview_url }}"></audio>
  </p>

  <p><a href (click)="back()">Back</a></p>
</div>
```

Like we did for the search before, we're going to use the Spotify API. Let's refactor the method `searchTrack` and extract two other useful methods we can reuse:

code/routes/music/src/app/spotify.service.ts

```
export class SpotifyService {
  static BASE_URL = 'https://api.spotify.com/v1';

  constructor(private http: Http) {
  }

  query(URL: string, params?: Array<string>): Observable<any[]> {
    let queryURL = `${SpotifyService.BASE_URL}${URL}`;
    if (params) {
      queryURL = `${queryURL}?${params.join('&')}`;
    }

    return this.http.request(queryURL).map((res: any) => res.json());
  }

  search(query: string, type: string): Observable<any[]> {
    return this.query(`/search`, [
      `q=${query}`,
```

```
30        `type=${type}`
31      ]);
32    }
```

Now that we've extracted those methods into the `SpotifyService`, notice how much simpler `searchTrack` becomes:

code/routes/music/src/app/spotify.service.ts

```
34    searchTrack(query: string): Observable<any[]> {
35      return this.search(query, 'track');
36    }
```

Now let's create a method to allow the component we're building retrieve track information, based in the track ID:

code/routes/music/src/app/spotify.service.ts

```
38    getTrack(id: string): Observable<any[]> {
39      return this.query(`/tracks/${id}`);
40    }
```

And now we can now use `getTrack` from a new `ngOnInit` method on the `TrackComponent`:

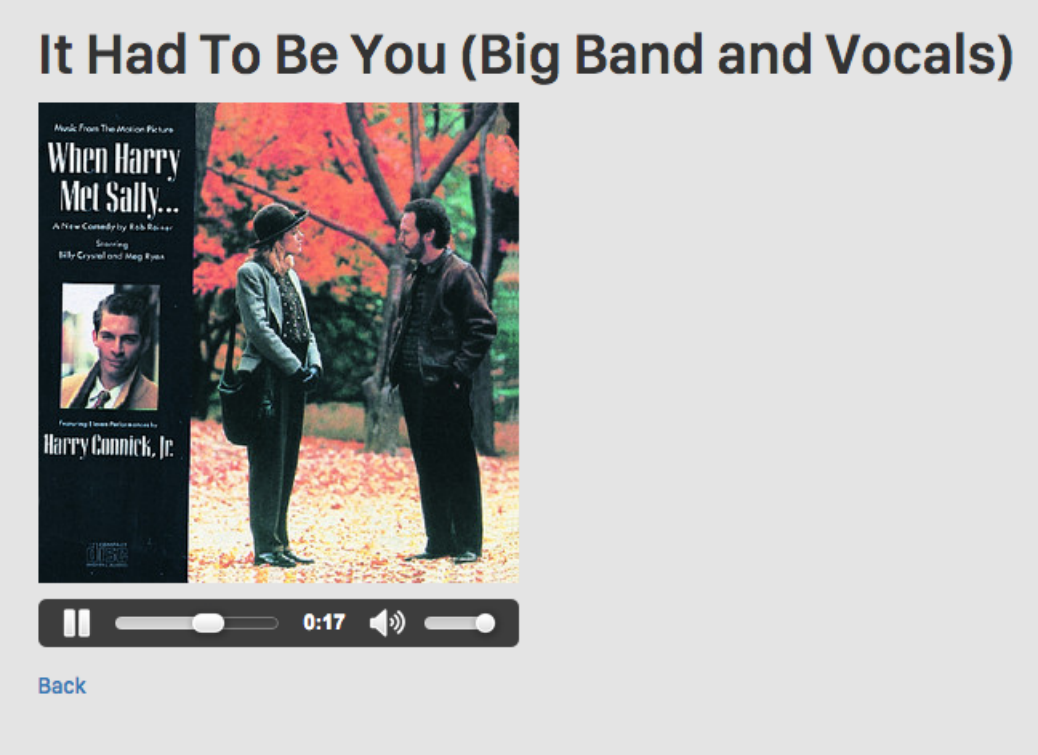
code/routes/music/src/app/track/track.component.ts

```
28    ngOnInit(): void {
29      this.spotify
30        .getTrack(this.id)
31        .subscribe((res: any) => this.renderTrack(res));
32    }
```

The other components work in a similar way and use `get*` methods from the `SpotifyService` to retrieve information about either an Artist or a Track based on their ID.

Wrapping up music search

Now we have a pretty functional music search and preview app. Try searching for a few of your favorite tunes and try it out!



It Had to Route You

Router Hooks

There are times that we may want to do some action when changing routes. A classical example of that is authentication. Let's say we have a **login** route and a **protected** route.

We want to only allow the app to go to the protected route if the correct username and password were provided on the login page.

In order to do that, we need to hook into the lifecycle of the router and ask to be notified when the protected route is being activated. We then can call an authentication service and ask whether or not the user provided the right credentials.

In order to check if a component can be activated we add a *guard class* to the key `canActivate` in our router configuration.

Let's revisit our initial application, adding login and password input fields and a new protected route that only works if we provide a certain username and password combination.

Sample Code The complete code for the examples in this section build on the first section and can be found in the `routes/routing` folder of the sample code. That folder contains a `README.md`, which gives instructions for building and running the project.

AuthService

Let's create a very simple and minimal implementation of a service, responsible for authentication and authorization of resources:

code/routes/routing/src/app/auth.service.ts

```
1  import { Injectable } from '@angular/core';
2
3  @Injectable()
4  export class AuthService {
5    login(user: string, password: string): boolean {
6      if (user === 'user' && password === 'password') {
7        localStorage.setItem('username', user);
8        return true;
9      }
10
11     return false;
12   }
```

The `login` method will return true if the provided user/password pair equals `'user'` and `'password'`, respectively. Also, when it is matched, it's going to use `localStorage` to save the username. This will also serve as a flag to indicate whether or not there is an active logged user.

If you're not familiar, `localStorage` is an HTML5 provided key/value pair that allows you to persist information on the browser. The API is very simple, and basically allows the setting, retrieval and deletion of items. For more information, see the `Storage` interface documents on MDN[70]

The `logout` method just clears the `username` value:

code/routes/routing/src/app/auth.service.ts

```
14   logout(): any {
15     localStorage.removeItem('username');
16   }
```

And the final two methods:

- `getUser` returns the username or null
- `isLoggedIn` uses `getUser()` to return true if we have a user

Here's the code for those methods:

[70] https://developer.mozilla.org/en-US/docs/Web/API/Storage

code/routes/routing/src/app/auth.service.ts

```
18   getUser(): any {
19     return localStorage.getItem('username');
20   }
21
22   isLoggedIn(): boolean {
23     return this.getUser() !== null;
24   }
```

The last thing we do is export an `AUTH_PROVIDERS`, so it can be injected into our app:

code/routes/routing/src/app/auth.service.ts

```
27 export const AUTH_PROVIDERS: Array<any> = [
28   { provide: AuthService, useClass: AuthService }
29 ];
```

Now that we have the `AuthService` we can inject it in our components to log the user in, check for the currently logged in user, log the user out, etc.

In a little bit, we'll also use it in our router to protect the `ProtectedComponent`. But first, let's create the component that we use to log in.

`LoginComponent`

This component will either show a login form, for the case when there is no logged user, or display a little banner with user information along with a logout link.

The relevant code here is the `login` and `logout` methods:

code/routes/routing/src/app/login/login.component.ts

```
9  export class LoginComponent {
10   message: string;
11
12   constructor(public authService: AuthService) {
13     this.message = '';
14   }
15
16   login(username: string, password: string): boolean {
17     this.message = '';
18     if (!this.authService.login(username, password)) {
19       this.message = 'Incorrect credentials.';
```

```
20        setTimeout(function() {
21          this.message = '';
22        }.bind(this), 2500);
23      }
24      return false;
25    }
26
27    logout(): boolean {
28      this.authService.logout();
29      return false;
30    }
```

Once our service validates the credentials, we log the user in.

The component template has two snippets that are displayed based on whether the user is logged in or not.

The first is a login form, protected by `*ngIf="!authService.getUser()"`:

code/routes/routing/src/app/login/login.component.html

```
5   </div>
6
7   <form class="form-inline" *ngIf="!authService.getUser()">
8     <div class="form-group">
9       <label for="username">User: (type <em>user</em>)</label>
10      <input class="form-control" name="username" #username>
11    </div>
12
13    <div class="form-group">
14      <label for="password">Password: (type <em>password</em>)</label>
15      <input class="form-control" type="password" name="password" #password>
16    </div>
17
18    <a class="btn btn-default" (click)="login(username.value, password.value)">
19      Submit
```

And the information banner, containing the logout link, protected by the inverse - `*ngIf="authService.getUser()"`:

code/routes/routing/src/app/login/login.component.html

```
<div class="well" *ngIf="authService.getUser()">
  Logged in as <b>{{ authService.getUser() }}</b>
  <a href (click)="logout()">Log out</a>
</div>
```

There's another snippet of code that is displayed when we have an authentication error:

code/routes/routing/src/app/login/login.component.html

```
<div class="alert alert-danger" role="alert" *ngIf="message">
  {{ message }}
</div>
```

Now that we can handle the user login, let's create a resource that we are going to protect behind a user login.

`ProtectedComponent` and Route Guards

The `ProtectedComponent`

Before we can protect the component, it needs to exist. Our `ProtectedComponent` is straightforward:

code/routes/routing/src/app/protected/protected.component.ts

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-protected',
  templateUrl: './protected.component.html',
  styleUrls: ['./protected.component.css']
})
export class ProtectedComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }

}
```

And the template will show some protected content:

code/routes/routing/src/app/protected/protected.component.html

```
<h1>Protected</h1>
<p>
  Protected content
</p>
```

We want this component to only be accessible to logged in users. But how can we do that?

The answer is to use the router hook `canActivate` with a *guard class* that implements `CanActivate`.

The `LoggedInGuard`

We create a new file `logged-in.guard.ts`:

code/routes/routing/src/app/logged-in.guard.ts

```
/* tslint:disble max-line-length */
import { Injectable } from '@angular/core';
import {
  CanActivate,
  ActivatedRouteSnapshot,
  RouterStateSnapshot
} from '@angular/router';
import { Observable } from 'rxjs/Observable';
import { AuthService } from './auth.service';

@Injectable()
export class LoggedInGuard implements CanActivate {
  constructor(private authService: AuthService) {}

  canActivate(
    next: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolea\
n {
      const isLoggedIn = this.authService.isLoggedIn();
      console.log('canActivate', isLoggedIn);
      return isLoggedIn;
  }
}
```

Angular CLI contains a generator for creating guards. So this file could be created with the command: `ng generate guard logged-in`

Our guard states that it implements the `CanActivate` interface. This is satisfied by implementing a method `canActive`.

We inject the `AuthService` into this class in the constructor and save it as a private variable `authService`.

In our `canActivate` function we check `this.authService` to see if the user `isLoggedIn`.

Configuring the Router

To configure the router to use this guard we need to do the following:

1. `import` the `LoggedInGuard`
2. Use the `LoggedInGuard` in a route configuration
3. Include `LoggedInGuard` in the list of providers (so that it can be injected)

We do all of these steps in our `app.ts`.

We import the `LoggedInGuard`:

code/routes/routing/src/app/app.module.ts

```
23 import { AUTH_PROVIDERS } from './auth.service';
24 import { LoggedInGuard } from './logged-in.guard';
```

We add `canActivate` with our guard to the `protected` route:

code/routes/routing/src/app/app.module.ts

```
26 const routes: Routes = [
27   // basic routes
28   { path: '', redirectTo: 'home', pathMatch: 'full' },
29   { path: 'home', component: HomeComponent },
30   { path: 'about', component: AboutComponent },
31   { path: 'contact', component: ContactComponent },
32   { path: 'contactus', redirectTo: 'contact' },
33
34   // authentication demo
35   { path: 'login', component: LoginComponent },
36   {
37     path: 'protected',
38     component: ProtectedComponent,
39     canActivate: [ LoggedInGuard ]
40   },
41
```

```
42    // nested
43    {
44      path: 'products',
45      component: ProductsComponent,
46      children: childRoutes
47    }
48  ];
```

We add `LoggedInGuard` to our list of providers:

code/routes/routing/src/app/app.module.ts

```
68    providers: [
69      // uncomment this for "hash-bang" routing
70      // { provide: LocationStrategy, useClass: HashLocationStrategy }
71      AUTH_PROVIDERS,
72      LoggedInGuard
73    ],
```

Logging in

We import the `LoginComponent`:

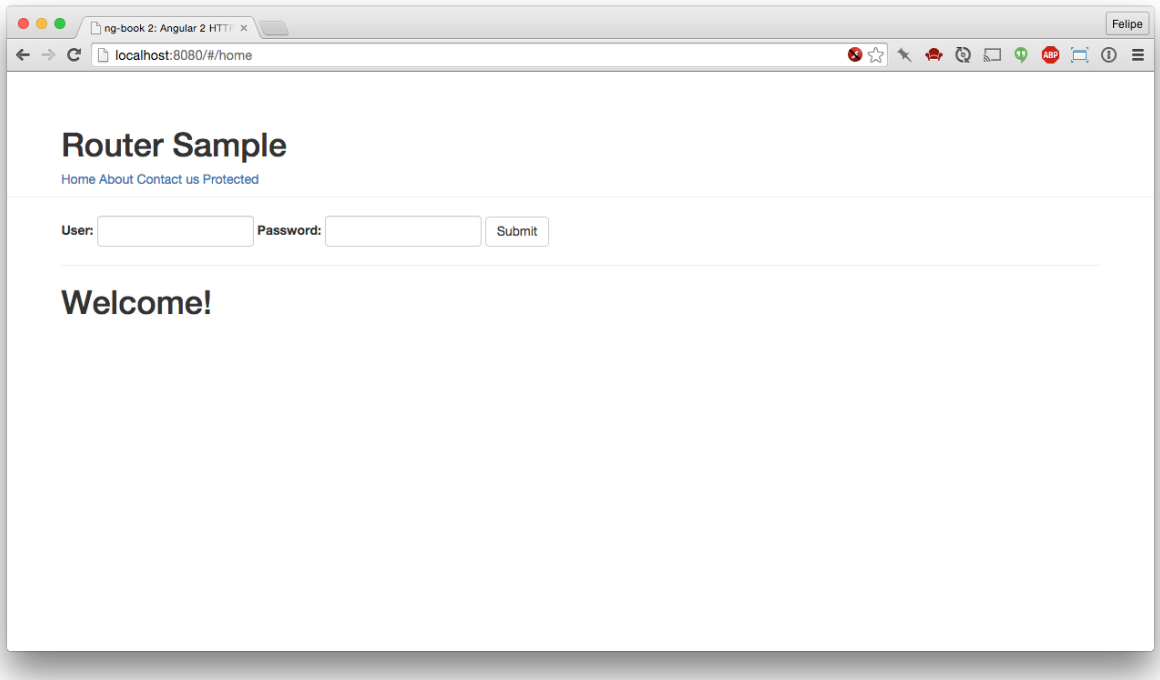
code/routes/routing/src/app/app.module.ts

```
19  import { LoginComponent } from './login/login.component';
```

And then to access it we have:

1. a route that links to the `LoginComponent`
2. a new link to the protected route

Now when we open the application on the browser, we can see the new login form and the new protected link:

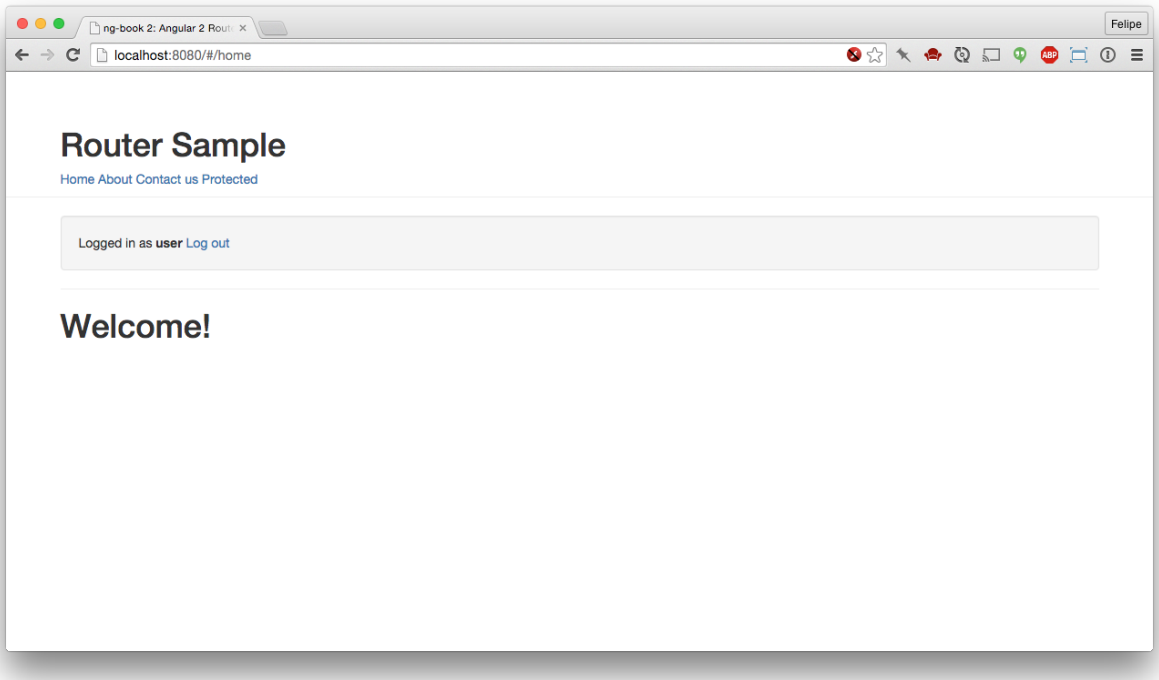


Auth App - Initial Page

If you click the Protected link, you'll see nothing happens. The same happens if you try to manually visit http://localhost:4200/protected[71].

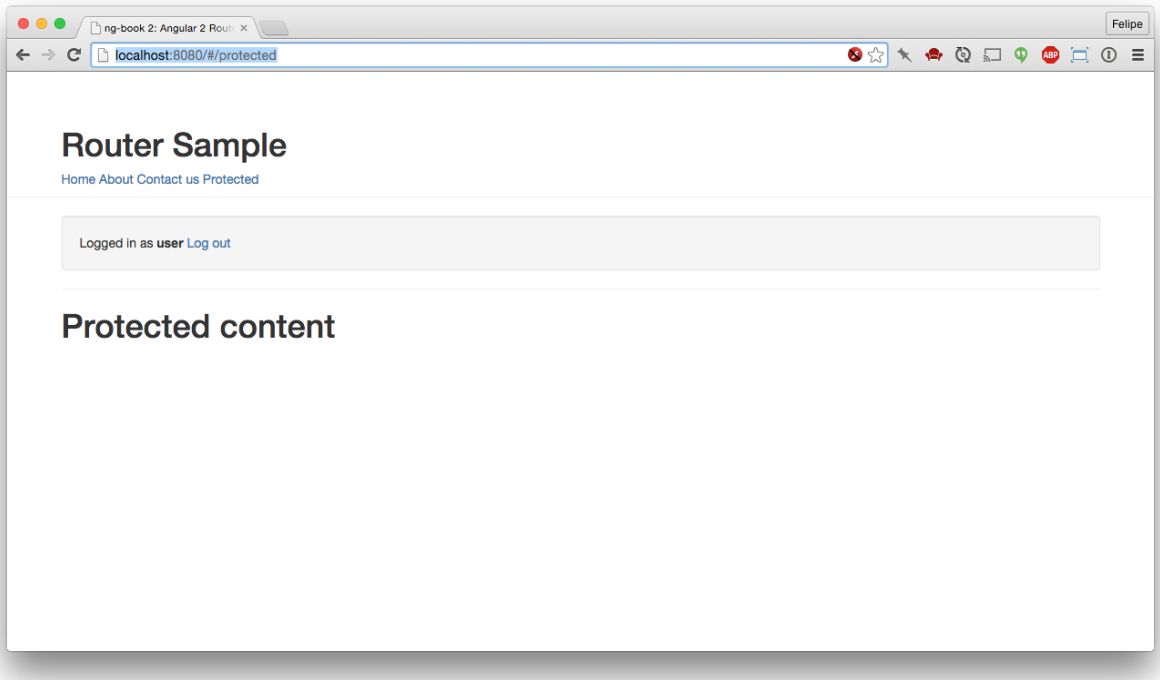
Now enter the string `user` for the user and `password` for the password on the form and click **Submit**. You'll see that we now get the current user displayed on a banner:

[71] http://localhost:4200/protected



Auth App - Logged In

And, sure enough, if we click the Protected link, it gets redirected and the component is rendered:



Auth App - Protected Area

A Note on Security: It's important to know how client-side route protection is working before you rely too heavily on it for security. That is, you should consider client-side route protection a form of *user-experience* and not one of security.

Ultimately all of the javascript in your app that gets served to the client can be inspected, whether the user is logged in or not.

So if you have sensitive data that needs to be protected, you must protect it with **server-side authentication**. That is, require an API key (or auth token) from the user which is validated by the server on every request for data.

Writing a full-stack authentication system is beyond the scope of this book. The important thing to know is that protecting routes on the client-side don't necessarily keep anyone from viewing the javascript pages behind those routes.

Nested Routes

Nested routes is the concept of containing routes within other routes. With nested routes we're able to encapsulate the functionality of parent routes and have that functionality apply to the child routes.

Let's say we have a website with one area to allow users to know our team, called **Who we are?** and another one for our **Products**.

We could think that the perfect route for **Who we are?** would be `/about` and for products `/products`.

And we're happily displaying all our team and all our products when visiting this areas.

What happens when the website grows and we now need to display individual information about each person in our team and also for each product we sell?

In order to support scenarios like these, the router allows the user to define nested routes.

To do that, you can have multiple, nested `router-outlet`. So each area of our application can have their own child components, that also have their own `router-outlets`.

Let's work on an example to clear things up.

In this example, we'll have a products section where the user will be able to view two highlighted products by visiting a nice URL. For all the other products, the routes will use the product ID.

Configuring Routes

We will start by describing the `products` route on the `app.module.ts` file:

code/routes/routing/src/app/app.module.ts

```
26 const routes: Routes = [
27   // basic routes
28   { path: '', redirectTo: 'home', pathMatch: 'full' },
29   { path: 'home', component: HomeComponent },
30   { path: 'about', component: AboutComponent },
31   { path: 'contact', component: ContactComponent },
32   { path: 'contactus', redirectTo: 'contact' },
33
34   // authentication demo
35   { path: 'login', component: LoginComponent },
36   {
37     path: 'protected',
38     component: ProtectedComponent,
39     canActivate: [ LoggedInGuard ]
40   },
41
42   // nested
43   {
44     path: 'products',
45     component: ProductsComponent,
46     children: childRoutes
47   }
48 ];
```

Notice that `products` has a `children` parameter. Where does this come from? We've defined the `childRoutes` **in a new module**: the `ProductsModule`. Let's take a look:

`ProductsModule`

The `ProductsModule` will have its own route configuration:

code/routes/routing/src/app/products/products.module.ts

```
15 export const routes: Routes = [
16   { path: '', redirectTo: 'main', pathMatch: 'full' },
17   { path: 'main', component: MainComponent },
18   { path: 'more-info', component: MoreInfoComponent },
19   { path: ':id', component: ProductComponent },
20 ];
```

Notice here that we have an empty `path` on the first object. We do this so that when we visit `/products`, we'll be redirected to the `main` route.

The other route we need to look at is `:id`. In this case, when the user visits something *that doesn't match any other route*, it will fallback to this route. Everything that is passed after `/` will be extracted to a parameter of the route, called `id`.

Now on the component template, we'll have a link to each of those static child routes:

code/routes/routing/src/app/products/products.component.html

```
3 <div class="navLinks">
4   <a [routerLink]="['./main']">Main</a> |
5   <a [routerLink]="['./more-info']">More Info</a> |
```

You can see that the route links are all in the format `['./main']`, with a preceding `./`. This indicates that you want to navigate the Main route *relative to the current route context.*

You could also declare the routes with the `['products', 'main']` notation. The downside is that by doing it this way, the child route is aware of the parent route and if you were to move this component around or reuse it, you would have to rewrite your route links.

After the links, we'll add an input where the user will be able to enter a product id, along with a button to navigate to it, and lastly add our `router-outlet`:

code/routes/routing/src/app/products/products.component.html

```
<h2>Products</h2>

<div class="navLinks">
  <a [routerLink]="['./main']">Main</a> |
  <a [routerLink]="['./more-info']">More Info</a> |
  Enter id: <input #id size="6">
  <button (click)="goToProduct(id.value)">Go</button>
</div>

<div class="products-area">
  <router-outlet></router-outlet>
</div>
```

Let's look at the `ProductsComponent` definition:

code/routes/routing/src/app/products/products.component.ts

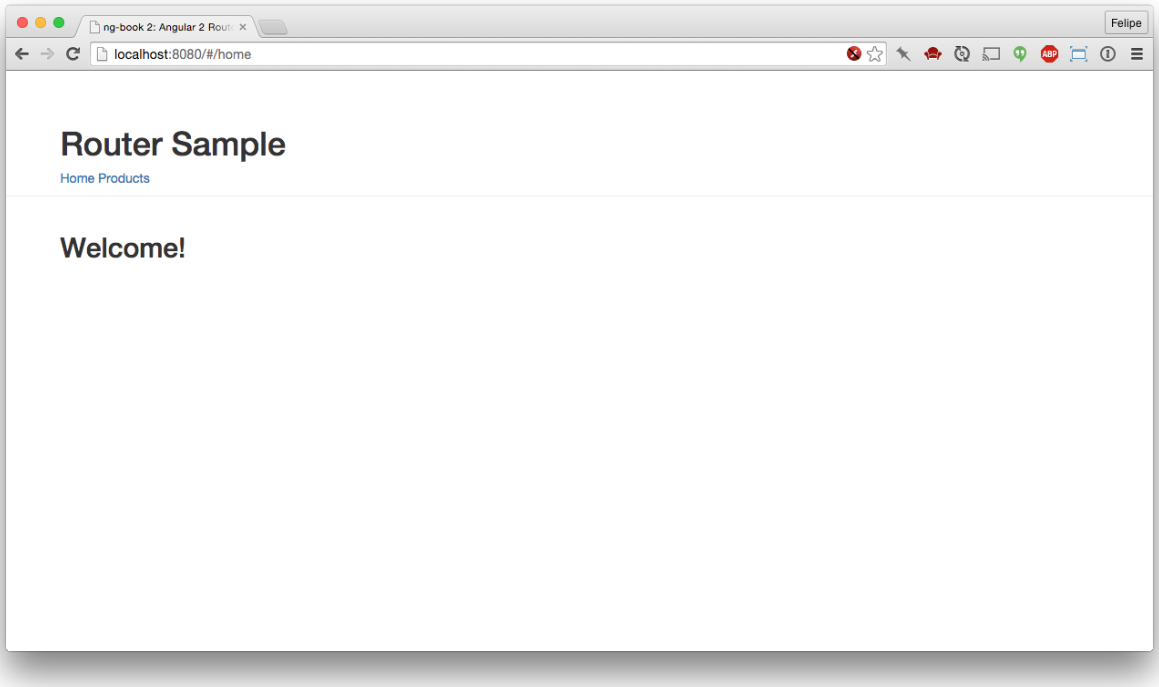
```
import { Component } from '@angular/core';
import {
  ActivatedRoute,
  Router
} from '@angular/router';

@Component({
  selector: 'app-products',
  templateUrl: './products.component.html',
  styleUrls: ['./products.component.css']
})
export class ProductsComponent {
  constructor(private router: Router, private route: ActivatedRoute) {
  }

  goToProduct(id: string): void {
    this.router.navigate(['./', id], {relativeTo: this.route});
  }
}
```

First on the constructor we're declaring an instance variable for the `Router`, since we're going to use that instance to navigate to the product by id.

When we want to go to a particular product we use the `goToProduct` method. In `goToProduct` we call the router's `navigate` method and providing the route name and an object with route parameters. In our case we're just passing the `id`.

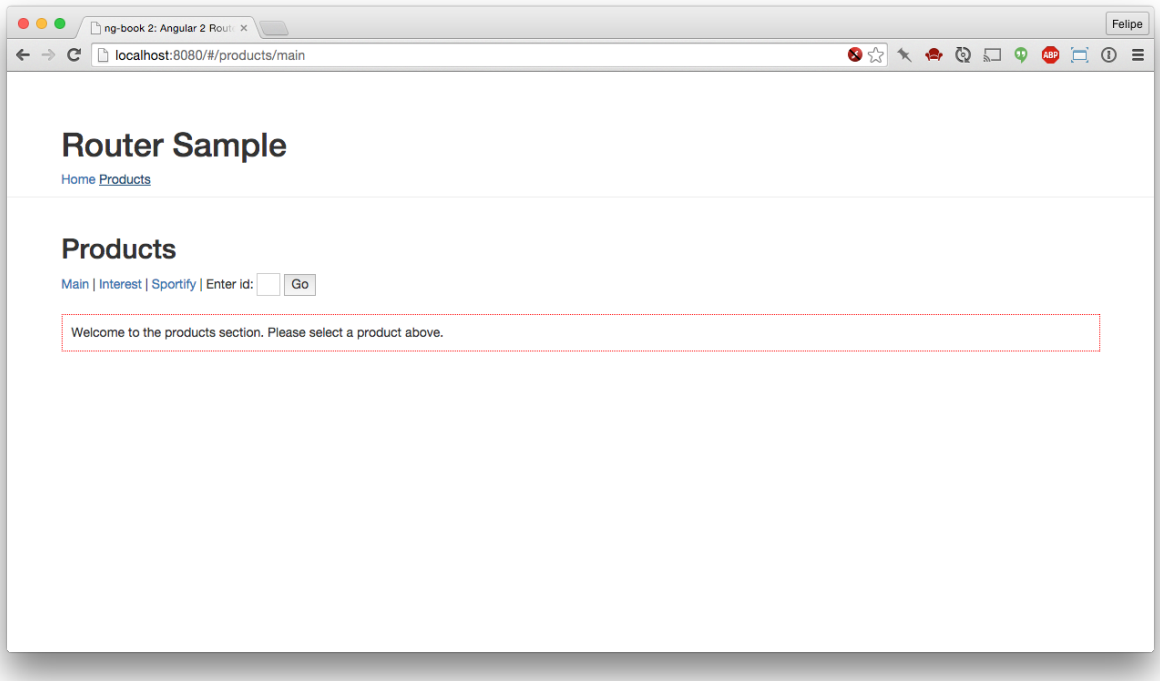
Notice that we use the relative `./` path in the `navigate` function. In order to use this we also pass the `relativeTo` object to the options, which tells the router what that route is relative to.

Now, if we run the application we will see the main page:



Nested Routes App

If you click on the Products link, you'll be redirected to `/products/main` that will render as follows:

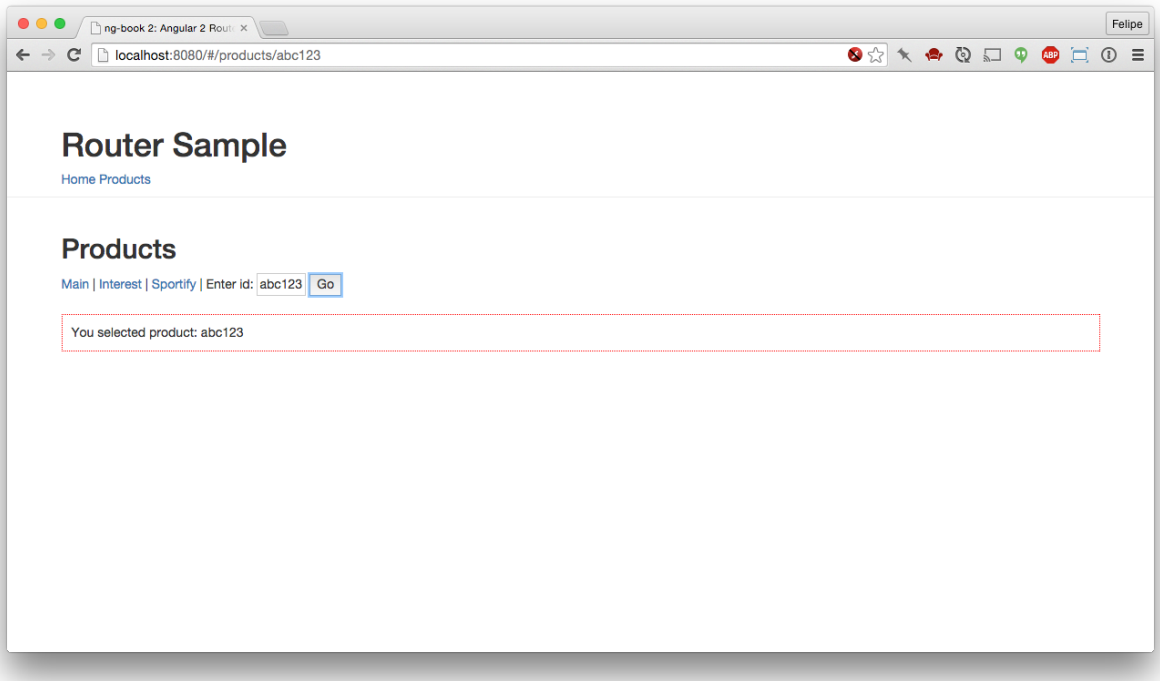


Nested Routes App - Products Section

Everything below that thin grey line is being rendered using the main application's `router-outlet`.

And the contents of the dotted red line is being rendered inside the `ProductComponent`'s `router-outlet`. That's how you indicate how the parent and child routes will be rendered.

When we visit one of the product links, or if we enter an ID on the textbox and click Go, the new content is rendered inside the `ProductComponent`'s outlet:



Nested Routes App - Product By Id

It's also worth noting that the Angular router is smart enough to prioritize concrete routes first (like `/products/sportify`) over the parameterized ones (like `/products/123`). This way `/products/sportify` will never be handled by the more generic, catch-all route `/products/:id`.

Redirecting and linking nested routes

Just to recap, if we want to go to a route named `MyRoute` on your top-level routing context, you use `['myRoute']`. This will only work if you're in that same top-level context.

If you are on a child component, and you try to link or redirect to `['myRoute']`, it will try to find a sibling route, and error out. In this case, you need to use `['/myRoute']` with a leading slash.

In a similar way, if we are on the top-level context and we want to link or redirect to a child route, we have to need to use multiple elements on the route definition array.

Let's say we want to visit the `Show` route, which is a child of the `Product` route. In this case, we use `['product', 'show']` as the route definition.

Summary

As we can see, the new Angular router is very powerful and flexible. Now go out and route your apps!