MIGRATING AN ANGULARJS APP TO ANGULAR

By Kim Maida

Auth0

Migrating an AngularJS App to Angular

Kim Maida, Auth0 Inc.

Version 1.3.0, 2017

Contents

Chapter 1

Introduction

Many AngularJS 1.x developers are interested in Angular 2+, but the major differences between versions 1 and 2+ are daunting when we have so many AngularJS 1 apps already in production or maintenance. Learn how to migrate a real-world AngularJS app to a fresh Angular 2+ build: what's the same, what's similar, and what's completely different. After this tutorial, you should be prepared to tackle your own migrations as well as new Angular 2+ projects. The final code for our Angular 2+ app can be cloned from the ng2-dinos GitHub repo.

Note: *The Branding Guidelines for Angular state that version 1.x should be referred to as "AngularJS", whereas all releases from version 2 and up are named "Angular". This migration article will continue to use "Angular 1" to refer to AngularJS (1.x) and "Angular 2" to refer to Angular (2 and up) in order to clearly differentiate the frameworks and reduce confusion.*

1.1 AngularJS 1 and Angular 2+

AngularJS 1.x has been a frontrunner among JavaScript frameworks over the past few years. There are thousands of production sites and apps built with Google's "superheroic MVW framework" and many more still in development. In mid-September 2016, Angular 2 was released after a lengthy period of betas and release candidates. Angular developers knew this was coming and that Angular 2 was a full rewrite and platform implementation, not an incremental update.

While Angular developers were and *are* eager to try Angular 2+, adoption can be challenging. Many of us have Angular 1 apps in development or maintenance and aren't in a position to migrate them to Angular 2 due to tight deadlines, budget constraints, client or management reluctance, etc. Angular 1 is still being maintained under the "AngularJS" moniker and Angular 1 apps are not about to go away.

Note: *Angular 2+ uses SemVer (Semantic Versioning). This means that unlike Angular 1, there will no longer be breaking changes in point releases. There will not be an Angular 3; instead, Angular 4 will be the next major release in order to correlate to version 4 of the Angular router.*

1.2 Migrate vs. Upgrade

Angular 2 is a powerful and attractive platform. Many developers will have their first opportunity to dig in when they tackle migrating an existing Angular 1 app to Angular 2. At this time, ***upgrading*** the original codebase is extremely difficult: Angular 2 is not an iteration of Angular 1. Moving between them is more straightforward when ***migrating*** to a fresh build that translates the same features on the new platform.

We'll walk through the process of migrating an Angular 1 app to Angular 2. Our Angular 1 project is relatively small but it represents a scalable, real-world Single Page Application. After following this tutorial, you should have a better understanding of how to get started with Angular 2 and how features from Angular 1 translate to Angular 2.

This tutorial assumes you are comfortable developing apps with AngularJS version 1.x. If you're looking to learn Angular 2 without an Angular 1 comparison, check out resources like Angular 2 Authentication and Getting Started with Angular 2.

Chapter 2

Angular 1 App "ng1-dinos"

Our Angular 1 app is called **ng1-dinos**. The code is available at the ng1-dinos GitHub repo. It has the following features:

- Routing (dinosaurs listing with individual detail pages)
- Filtering (search for dinosaurs by name)
- Calls an external Node API to get dinosaur data
- SCSS and Bootstrap CSS
- Custom off-canvas navigation
- Metadata factory to provide dynamic `<title>`s
- Gulp build
- Guided by the Angular 1 Style Guide
- Scalability

2.1 Dependencies

Follow the instructions on the following sites to install these dependencies:

- NodeJS with npm
- Gulp (install globally with `npm install -g gulp`)

We'll also need to clone **sample-nodeserver-dinos**. This local Node server will provide the external API for both our ng1-dinos and ng2-dinos apps. Follow the instructions in the sample-nodeserver-dinos README to get it installed and running on http://localhost:3001.

2.2 Install and Run "ng1-dinos"

1. Clone **ng1-dinos** from GitHub to a local directory of your choosing.

2. Run `npm install` from the root directory.
3. Run `gulp` to serve the application (runs locally on http://localhost:8000).

Once you have ng1-dinos and the Node API running, the app should look like this in the browser:

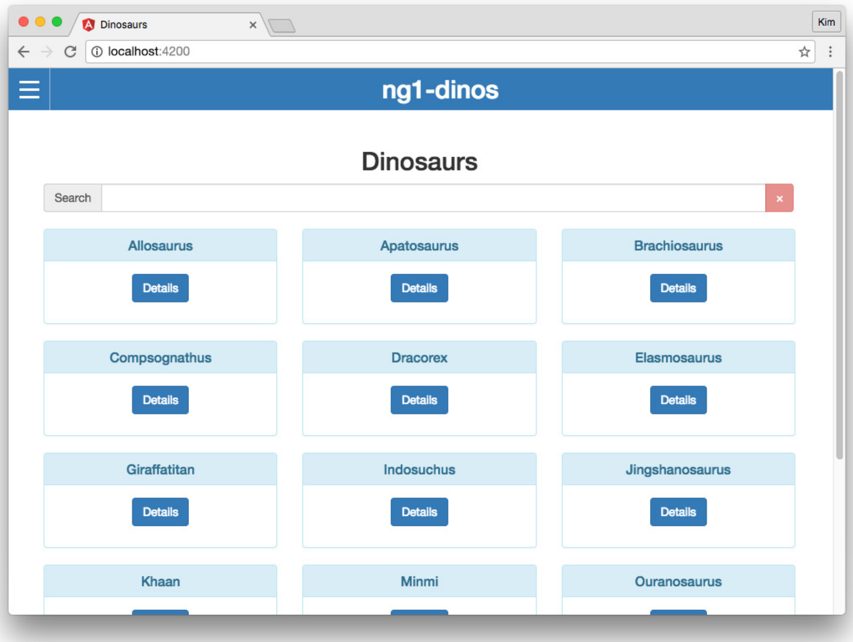


Figure 2.1: Angular 1 ng1-dinos home view

Important: Take some time to familiarize with the file structure, code, and features. We won't be making any *changes* to this application, but it's important to get comfortable with it because everything we do in our Angular 2 app will be a migration of ng1-dinos.

Chapter 3

Introducing Angular 2 App "ng2-dinos"

Our migrated Angular 2 application will be called **ng2-dinos**. The full source code for the completed app can be cloned from the ng2-dinos GitHub repo. This app will use the same Node API. From a user's perspective, we want ng2-dinos to be indistinguishable from ng1-dinos. Under the hood, we'll rewrite the app to take advantage of the powerful new features of Angular 2.

Angular 2 brings in several technologies that ng1-dinos does not take advantage of. Instead of a Gulp build, we'll use the Angular CLI to set up and serve ng2-dinos. We're going to write the app using TypeScript and ES6 which will be transpiled by the Angular CLI.

We'll follow the Angular 2 Style Guide for the most part, with a few minor exceptions regarding file structure. For this tutorial, we want to preserve as much of a correlation with ng1-dinos as we can. This will make it easier to follow the migration of features.

3.1 Dependencies

You should have NodeJS with npm installed already.

Next, install the Angular CLI globally with the following command:

```
$ npm install -g @angular/cli
```

3.2 Initialize ng2-dinos

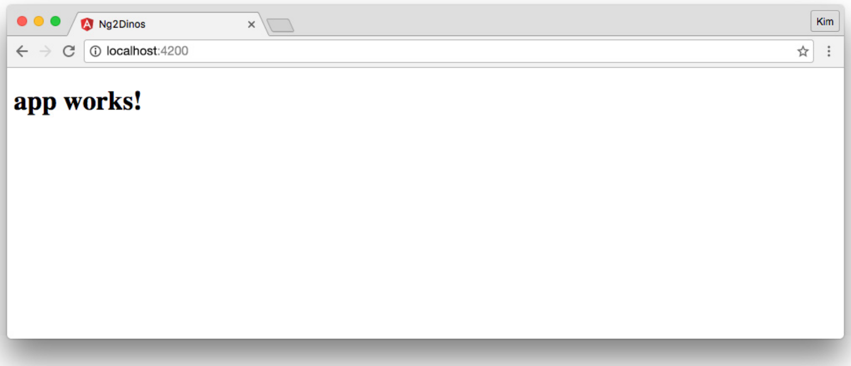
The first thing we'll do is initialize our new Angular 2 app and get it running. We'll use the Angular CLI to generate a new project with SCSS support using the following command:

```
$ ng new ng2-dinos --style=scss
```

Next we can serve the app by running the following command from the root directory of our new app:

```
$ ng serve
```

We should be able to view the site in the browser at http://localhost:4200. The app should look like this:



app works!

Figure 3.1: New Angular 2 app initialized

Take a look at the file structure for your new ng2-dinos app. You may notice there are test files and configuration, but **we won't cover testing in this tutorial**. If you'd like to learn more about testing Angular 2, check out Testing in the Angular docs and articles like Angular 2 Testing In Depth: Services and Three Ways to Test Angular 2 Components.

3.3 Linting and Style Guide

The Angular CLI provides code linting with TSLint and Codelyzer. TSLint provides TypeScript linting and Codelyzer provides TSLint rules that adhere to the Angular 2 Style Guide. We can view all of these linting rules at `ng2-dinos/tslint.json`. We can lint our project using the following command:

```
$ ng lint
```

This tutorial follows the Style Guide and adheres to the default rules in the TSLint config file. It's good to lint your project periodically to make sure your code is clean and free of linter errors.

Note: *The Angular CLI TSLint* `"eofline": true` *rule requires files to end with a* newline*. This is standard convention. If you want to avoid lots of newline errors when linting, make sure that your files include this.*

Chapter 4

Customizing Our Angular 2 Project for Migration

Now that we have a working starter project for our ng2-dinos app, we want to restructure it and add some libraries.

4.1 Bootstrap CSS

Let's start by adding the Bootstrap CSS CDN to the `ng2-dinos/src/index.html` file. We can also add a default `<title>` and some `<meta>` tags:

```
<!-- ng2-dinos/src/index.html -->

<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>ng2-dinos</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <meta name="author" content="Auth0">
  <meta name="description" content="Learn about some popular as well as obscure dinosaurs!"
  <base href="/">

  <!-- Bootstrap CDN stylesheet -->
  <link
    rel="stylesheet"
    href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
    integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
    crossorigin="anonymous">
```

```
</head>

<body>
  <app-root>Loading...</app-root>
</body>
</html>
```

Note: *The code for including the Bootstrap CSS can be found at Bootstrap CDN - Getting Started. We're using version 3.3.7 because it is latest stable at the time of writing. Please note that if you upgrade to version 4.x, there are major changes to be mindful of.*

4.2 Third Party Libraries

The only third party JavaScript we want is a custom build of Modernizr. We'll be doing quite a bit of copying and pasting from ng1-dinos since we're doing a migration, so it's best to keep your local ng1-dinos project handy.

Our minified, custom Modernizr build can be found at ng1-dinos `modernizr.min.js`. Create the necessary folder structure in ng2-dinos:

```
ng2-dinos
  |-src/
    |-assets/
      |-js/
        |-vendor/
          |-modernizr.min.js
```

Angular CLI uses Webpack to bundle local dependencies, so we *won't* add Modernizr to our ng2-dinos index file. Instead, we'll add a reference to the `.angular-cli.json` app's `scripts`:

```
// ng2-dinos/.angular-cli.json

{...
  "apps": [
    {...
      "scripts": [
        "assets/js/vendor/modernizr.min.js"
      ],
      ...
```

4.3 Global SCSS

We initialized our project with the `--styles=scss` flag so SCSS is supported and a global `styles.scss` file has already been generated. However, it's currently located at the root of the `ng2-dinos/src/` folder. To maintain a similar file structure with ng1-dinos, it needs to live in `ng2-dinos/src/assets/scss/` instead.

Create a `ng2-dinos/src/assets/scss/` folder and move the `ng2-dinos/src/styles.scss` file into it. Then update the `.angular-cli.json` app's `styles` reference:

```
// ng2-dinos/.angular-cli.json

{...
  "apps": [
    {...
      "styles": [
        "assets/scss/styles.scss"
      ],
      ...
```

Note: *When moving or adding new files, you'll need to stop and restart the Angular CLI server (`Ctrl+C`, `ng serve`) to avoid module build errors. Changes within files are watched and live reloaded, but reorganizing the file structure can break this.*

Now let's add some global SCSS from ng1-dinos. We'll copy the files and subdirectories from `ng1-dinos/src/assets/css/scss/core/` to `ng2-dinos/src/assets/scss/`.

Note: *If you paid close attention, you'll notice we've left off a folder in ng2-dinos. Our Angular 1 ng1-dinos app had a `css` folder with `scss` inside it. We don't need the `css` folder in ng2-dinos because of the Angular CLI Webpack bundling.*

When we're done, our ng2-dinos global styles file structure should look like this:

```
ng2-dinos
  |-src/
    |-assets/
      |-scss/
        |-partials/
          |-_layout.vars.scss
          |-_responsive.partial.scss
        |-_base.scss
        |-_layout.scss
        |-_presentation.scss
        |-styles.scss
```

Now we'll `@import` these SCSS files in the ng2-dinos global `styles.scss`:

```
/* ng2-dinos/src/assets/scss/styles.scss */

// partials
@import 'partials/layout.vars';
@import 'partials/responsive.partial';

// global styles
@import 'base';
@import 'presentation';
@import 'layout';
```

Restart the Angular CLI server and our app's background color should change to grey. This is a visual indicator that our new global styles are working. If we inspect the page, we should see the global `<body>` styles applied.

Finally, we'll clean up the `_base.scss` file. Angular 2 doesn't utilize `ng-cloak` so we'll remove the `ng-cloak` ruleset. Afterwards, this is what remains:

```
/* ng2-dinos/src/assets/scss/_base.scss */

/*--------------------
       BASICS
--------------------*/

/*-- Cursor --*/

a,
input[type=button],
input[type=submit],
button {
  cursor: pointer;
}

/*-- Forms --*/

input[type="text"],
input[type="number"],
input[type="password"],
input[type="date"],
select option,
textarea {
  font-size: 16px;  /* for iOS to prevent autozoom */
}
```

4.4 Update App File Structure

The Angular CLI creates all app files (modules, components, services, pipes, etc.) relative to `ng2-dinos/src/app/`. Note that the ng2-dinos app has a component (`app.component.ts|.html|.scss|.spec.ts`) in the root of this folder. This is our app's root component, but we want to move it into a subfolder to keep ng2-dinos organized, scalable, and correlated with ng1-dinos.

Note: *Recall that this tutorial won't cover testing. The `.spec.ts` files have been largely removed from the sample ng2-dinos repo to make it simpler to view. The Angular CLI creates these files automatically when generating new architecture. Feel free to keep them in your project and write tests. For brevity,* ***the rest of the tutorial will no longer mention `.spec.ts` files.*** *If you're using them, just remember to include them whenever managing files.*

Let's move the `app.component[.html|.scss|.ts]` files to a new folder: `ng2-dinos/src/app/core/`. The `app` folder's file structure should now look like this:

```
ng2-dinos
  |-src/
    |-app/
      |-core/
        |-app.component[.html|.scss|.ts]
      |-app.module.ts
```

This breaks our build. We can fix it by updating the `ng2-dinos/src/app/app.module.ts` file. If you have a TypeScript extension enabled in your code editor or IDE, you should see syntax highlighting where TypeScript detects problems. We need to update the path to `app.component` like so:

```
// ng2-dinos/src/app/app.module.ts

...
import { AppComponent } from './core/app.component';
...
```

Note: *Always keep in mind that Angular 2 is very interconnected with regard to dependency imports. When we move files, we break references in other places. The CLI tells us where the problems are when we build. TypeScript code hinting in our editor can help too. To address the issue at its root, we can use additional `@NgModules` to manage dependencies; you can learn more by reading Use NgModule to Manage Dependencies in your Angular 2 Apps.*

That's it for setup! We can officially start migrating ng1-dinos to ng2-dinos.

Chapter 5

Angular 2 Root App Component

In the ng1-dinos Angular 1 app, `ng-app` was on the `<html>` element. This provided Angular control over the `<head>`, allowing us to dynamically update the `<title>` with a custom metadata factory. In Angular 2, our root app component is located inside the `<body>`. Angular 2 provides a service to manage page `<title>`s and we shouldn't use an `<html>`-level app root anymore.

As we saw above, the body of our Angular 2 **ng2-dinos** `index.html` file looks like this:

```
<!-- ng2-dinos/src/index.html -->

...
<body>
  <app-root>Loading...</app-root>
</body>
```

In comparison, the body of our Angular 1 **ng1-dinos** `index.html` file looks like this:

```
<!-- ng1-dinos/src/index.html -->

...
<body>
  <div class="layout-overflow">
    <div
      class="layout-canvas"
      nav-control
      ng-class="{'nav-open': nav.navOpen, 'nav-closed': !nav.navOpen}">
      <!-- HEADER -->
      <header
        id="header"
        class="header"
        ng-include="'app/header/header.tpl.html'"></header>

      <!-- CONTENT (Angular View) -->
      <div
        id="layout-view"
        class="layout-view"
        ng-view autoscroll="true"></div>

      <!-- FOOTER -->
      <footer
        id="footer"
        class="footer clearfix"
        ng-include="'app/footer/footer.tpl.html'"></footer>

    </div> <!-- /.layout-canvas -->
  </div> <!-- /.layout-overflow -->
  ...
</body>
```

The layout markup, header, content, and footer children will now move to the ng2-dinos root component `app.component` (`<app-root>`).

5.1 App Component Template

Let's stub out `app.component.html`:

```
<!-- ng2-dinos/src/app/core/app.component.html -->

<div class="layout-overflow">
  <div
    class="layout-canvas"
    [ngClass]="{'nav-open': navOpen, 'nav-closed': !navOpen}">

    <!-- HEADER -->

    <!-- CONTENT -->
    <div id="layout-view" class="layout-view">
        ...content goes here...
    </div>

    <!-- FOOTER -->
  </div> <!-- /.layout-canvas -->
</div> <!-- /.layout-overflow -->
```

5.2 App Component Styles

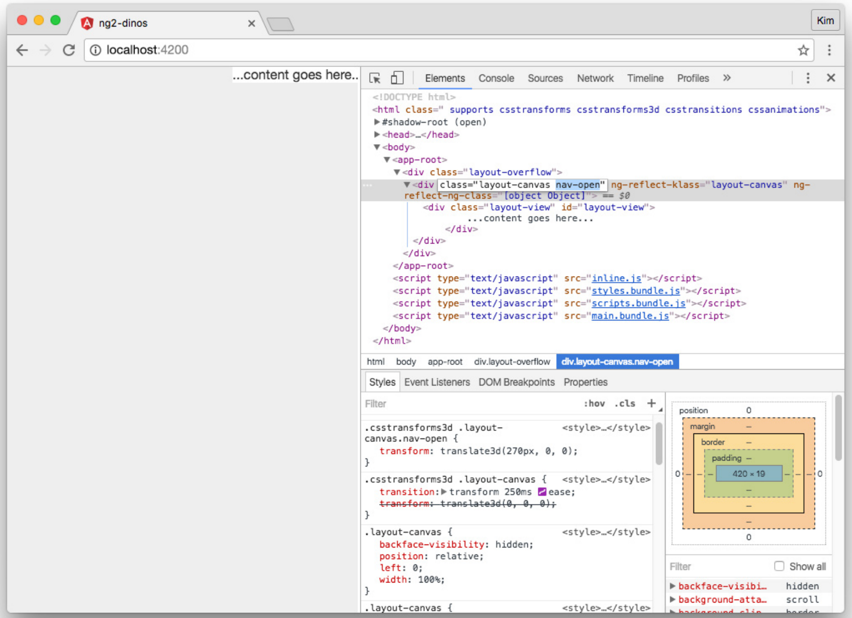
We already included global SCSS for the site layout and off-canvas nav functionality. Because the styles for the layout, header, and navigation interact with each other, we won't componetize the layout styles in this tutorial. We want to maintain a fairly direct migration path with ng1-dinos, but there will be room for refactoring after the app is migrated. We won't use the `app.component.scss` file so let's delete it.

5.3 App Component TypeScript

Now we'll add the `navOpen` boolean property we referenced for controlling the `.nav-open`/`.nav-closed` classes in the `app.component.html` above. We also need to remove the reference to `app.component.scss` since we deleted that file:

```
// ng2-dinos/src/app/core/app.component.ts

import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  navOpen: boolean;

  constructor() { }
}
```

If we restart the Angular CLI server now and inspect the DOM in the browser, we'll see a `.nav-closed` class on the `<div class="layout-canvas">` element. We can use the inspector to change `.nav-closed` to `.nav-open`. If we do this, we should see the page content slide to the right:



```
<!DOCTYPE html>
<html class=" supports csstransforms csstransforms3d csstransitions cssanimations">
 ▶#shadow-root (open)
 ▶<head>…</head>
 ▼<body>
   ▼<app-root>
     ▼<div class="layout-overflow">
       ▼<div class="layout-canvas nav-open" ng-reflect-klass="layout-canvas" ng-
        reflect-ng-class="[object Object]"> == $0
           <div class="layout-view" id="layout-view">
                 ...content goes here...
               </div>
         </div>
       </div>
     </app-root>
     <script type="text/javascript" src="inline.js"></script>
     <script type="text/javascript" src="styles.bundle.js"></script>
     <script type="text/javascript" src="scripts.bundle.js"></script>
     <script type="text/javascript" src="main.bundle.js"></script>
   </body>
 </html>
```

```
.csstransforms3d .layout-          <style>…</style>
canvas.nav-open {
   transform: translate3d(270px, 0, 0);
}
.csstransforms3d .layout-canvas {   <style>…</style>
   transition: ▶ transform 250ms ease;
   ~~transform: translate3d(0, 0, 0);~~
}
.layout-canvas {                   <style>…</style>
   backface-visibility: hidden;
   position: relative;
   left: 0;
   width: 100%;
}
.layout-canvas {                   <style>…</style>
```

```
▶ backface-visibi…     hidden
▶ background-atta…     scroll
```

Figure 5.1: Angular 2 ng2-dinos app with off-canvas navigation open

Now we're ready to create the header.

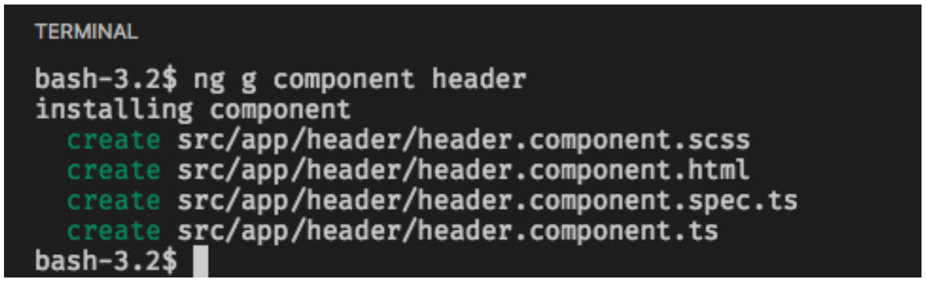
Chapter 6

Angular 2 Header Component

We can use the Angular CLI's `g` command (shortcut for `generate`) to generate new components for our app. Stop the server (`Ctrl+C`) and let's create a header component:

```
$ ng g component header
```

New components are created relative to the `ng2-dinos/src/app` root. The resulting output should resemble the following:



```
TERMINAL

bash-3.2$ ng g component header
installing component
  create src/app/header/header.component.scss
  create src/app/header/header.component.html
  create src/app/header/header.component.spec.ts
  create src/app/header/header.component.ts
bash-3.2$ 
```

Figure 6.1: Create an Angular 2 component with the Angular CLI

We can see from the terminal output that new files were created, but let's also look at the `app.module.ts` file so we're familiar with everything necessary for adding new components to an Angular 2 app.

`app.module.ts` is our app's primary `@NgModule`. It now looks like this:

```
// ng2-dinos/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 { AppComponent } from './core/app.component';
import { HeaderComponent } from './header/header.component';

@NgModule({
  declarations: [
    AppComponent,
    HeaderComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

As you can see, the `HeaderComponent` class is imported and has also been added to the `@NgModule`'s `declarations` array.

6.1 Add Header Element to App Component Template

If we open he `header.component.ts` file, we can see that the `@Component`'s `selector` is `app-header`. We generally want custom elements to be hyphenated as per the W3C spec for custom elements. This is also covered by the Angular 2 Style Guide. The Angular CLI generates new component selectors with a prefix. By default, this prefix is `app`. This way, we won't get conflicts with native elements when calling this component (since `<header>` already exists in the HTML5 spec).

Let's add `<app-header>` to our `app.component.html`:

```
<!-- ng2-dinos/src/app/core/app.component.html -->

...
    <!-- HEADER -->
    <app-header></app-header>
...
```

6.2 Header Component Template

Let's add our markup to the header component similar to ng1-dinos header.tpl.html. Open header.component.html and add HTML for the header, off-canvas toggle, and navigation menu:

```
<!-- ng2-dinos/src/app/header/header.component.html -->

<header id="header" class="header">
  <div class="header-page bg-primary">
    <a class="toggle-offcanvas bg-primary" (click)="toggleNav()"><span></span></a>
    <h1 class="header-page-siteTitle">
      <a href="/">ng2-dinos</a>
    </h1>
  </div>

  <nav id="nav" class="nav" role="navigation">
    <ul class="nav-list">
      <li>
        <a href>Dinosaurs</a>
      </li>
      <li>
        <a href>About</a>
      </li>
      <li>
        <a href="https://github.com/auth0-blog/sample-nodeserver-dinos">Dino API on GitHub</
      </li>
    </ul>
  </nav>
</header>
```

This is mostly standard markup. The only Angular 2 functionality so far is a (click) binding on the link to toggle the off-canvas menu. We'll add more Angular later once we have multiple views and routing in place.

6.3 Header Component Styles

First grab the Angular 1 ng1-dinos _nav.scss file and copy it into the ng2-dinos header component folder.

Now let's @import it and add SCSS to header.component.scss:

```
/* ng2-dinos/src/app/header/header.component.scss */

/*--------------------
       HEADER
--------------------*/

@import '../../assets/scss/partials/layout.vars';
@import 'nav';

.header-page {
  color: #fff;
  height: 50px;
  margin-bottom: 10px;
  position: relative;

  &-siteTitle {
    font-size: 30px;
    line-height: 50px;
    margin: 0;
    padding: 0 0 0 50px;
    position: absolute;
      top: 0;
    text-align: center;
    width: 100%;

    a {
      color: #fff;
      text-decoration: none;
    }
  }
}
```

We need to make one modification in the `_nav.scss` file. We'll change the `.nav-open &` selector to `:host-context(.nav-open) &` instead:

```
/* ng2-dinos/src/app/header/_nav.scss */

...
  :host-context(.nav-open) & {
    span {
      background: transparent;

      &:before,
      &:after { ...
```

This has to do with how Angular 2 encapsulates DOM node styles. If you've ever used native web components or Google Polymer, you should be familiar with shadow DOM encapsulation in components. Regardless, you may want to read about View Encapsulation in Angular 2.

In a nutshell, Angular 2's default encapsulation mode is `Emulated`. This means styles are scoped to their components with unique attributes that Angular 2 generates. Having component-isolated styles is often very useful—except for when we want to reach up the DOM tree and have our component styles affected by ancestors.

We don't need to change View Encapsulation in the header component class though. There is only one reference to an ancestor in `_nav.scss`. We can use special selectors like `:host-context()` to look up the cascade instead.

Now the component CSS can access the `.nav-open` class up the DOM tree from the header component.

Note: *Recall that the site layout and navigation functionality styles remained global rather than being componetized in `app.component.scss` (instead we deleted that file). We could have moved the sections of the global `_layout.scss` into different child components and replaced references to parent styles with `:host-context()`. We didn't do this because the goal of this tutorial is to demonstrate* ***as close to a 1:1 migration as possible*** *while covering many topics. When we're finished migrating the entire app, I encourage you to refactor where desirable! We'll highlight refactoring suggestions at the end of this tutorial.*

Chapter 7

Angular 2 Component Interaction

Let's make our header component functional. We need the header to communicate with the root app component to implement the off-canvas navigation.

7.1 Header Component TypeScript

Open the `header.component.ts` file. We'll implement component communication with inputs/outputs and events. Remember that we added a `click` event binding to our header HTML that looked like this:

```
<a class="toggle-offcanvas bg-primary" (click)="toggleNav()"><span></span></a>
```

Note: *`[]`, `()`, and `[()]` are "binding punctuation" and refer to the direction of data flow. `()` indicates a binding to an event. You can read more about binding syntax in the Angular 2 docs.*

There are a few things we need to do to make this event handler functional.

```
// ng2-dinos/src/app/header/header.component.ts

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

@Component({
  selector: 'app-header',
  templateUrl: './header.component.html',
  styleUrls: ['./header.component.scss']
})
export class HeaderComponent implements OnInit {
  @Output() navToggled = new EventEmitter();
  navOpen = false;

  constructor() { }

  ngOnInit() {
  }

  toggleNav() {
    this.navOpen = !this.navOpen;
    this.navToggled.emit(this.navOpen);
  }

}
```

The header component is a child of the root app component. We need a way to notify the parent when the user clicks the hamburger to open or close the menu. We'll do this by emitting an event that the parent can bind to.

We'll import `Output` and `EventEmitter` from `@angular/core` and then create a new event emitter `@Output` decorator. We also need a way to track whether the navigation is open or closed, so we'll add a `navOpen` property that defaults to `false`.

Note: *Notice that we didn't declare a type annotation for `navOpen`. This is because we initialized the property with a value. The type is inferred from this value. Adding type annotations that can be inferred automatically will result in linting errors.*

Now we need to define the `click` event handler. We already named this function `toggleNav()` in our `header.component.html`. The function will toggle the `navOpen` boolean and emit the `navToggled` event with the current state of `navOpen`.

7.2 Header Communication with App Component

Next we need to listen for the `navToggled` event in the parent. Add the following declarative code to `app.component.html`:

```
<!-- ng2-dinos/src/app/core/app.component.html -->

...
    <!-- HEADER -->
    <app-header (navToggled)="navToggleHandler($event)"></app-header>
...
```

Now we'll create the `navToggleHandler($event)` in `app.component.ts`:

```
// ng2-dinos/src/app/core/app.component.ts

import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  navOpen: boolean;

  navToggleHandler(e: boolean) {
    this.navOpen = e;
  }
}
```

If we build now, we should be able to open and close the off-canvas navigation by clicking the hamburger icon. When open, the icon should animate into an X and the app should look like this:

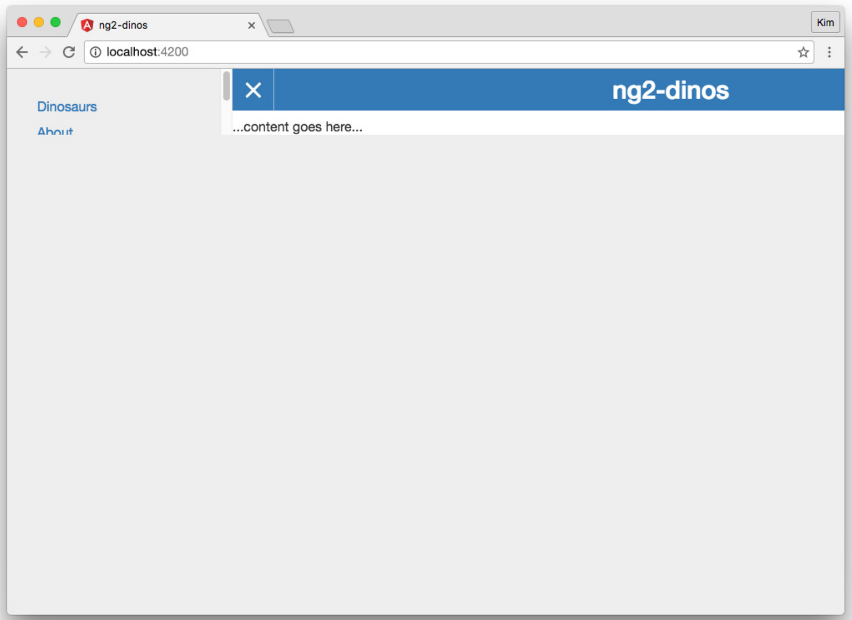


Figure 7.1: Angular 2 ng2-dinos app with off-canvas navigation

Everything is working correctly but this doesn't look very good. Let's fix it!

Chapter 8

Angular 2 Observables and DOM Properties

In ng1-dinos, all off-canvas nav functionality was handled by `navControl.dir.js`, including menu toggling and layout height. We've migrated the navigation functionality but we're still missing the layout height fix.

We want our minimum page height to be the height of the window no matter how tall the content is. This way, the off-canvas navigation will never look prematurely cut off. To address this, we'll use an RxJS observable and the `window.resize` event.

Note: *In ng1-dinos, we referenced the* `navControl` *directive's DOM* `$element` *and applied* `min-height` *styles with JS. We did this to avoid an additional watcher in Angular 1. However, Angular 2's change detection is vastly improved so we can shift our concerns over watchers to other things instead.*

Angular 2 strongly recommends *avoiding* direct DOM manipulation. There is an `ElementRef` class that provides access to the native element, but using it is not recommended and is usually avoidable. We'll use property data binding instead.

8.1 Add Observable to App Component TypeScript

Our `app.component.ts` will look like this:

```
// ng2-dinos/src/app/core/app.component.ts

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

```
declare var window: any;

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent implements OnInit {
  navOpen: boolean;
  minHeight: string;
  private initWinHeight: number = 0;

  ngOnInit() {
    Observable.fromEvent(window, 'resize')
      .debounceTime(200)
      .subscribe((event) => {
        this.resizeFn(event);
      });

    this.initWinHeight = window.innerHeight;
    this.resizeFn(null);
  }

  navToggleHandler(e: boolean) {
    this.navOpen = e;
  }

  private resizeFn(e) {
    let winHeight: number = e ? e.target.innerHeight : this.initWinHeight;
    this.minHeight = `${winHeight}px`;
  }
}
```

Let's talk about the code above.

First we'll import dependencies. We're going to use the `OnInit` lifecycle hook from `@angular/core` to manage the observable and implement initial layout height. Then we need `Observable` from the RxJS library which is packaged with Angular 2.

In order to avoid TypeScript `Name not found` errors, we'll declare the type for `window` to be `any`.

We're using an RxJS observable to subscribe to the `window.resize` event and execute a debounced function that sets a `min-height`. The `window.resize` event doesn't automatically fire on page load, so we need to trigger the handler manually in `ngOnInit()`.

Note: *This tutorial does* not *cover Reactive Programming (RP) and RxJS in depth. If RP and RxJS are new to you, please read Understanding Reactive Programming and RxJS, or for a more Angular 2-centric approach: Functional Reactive Programming for Angular 2 Developers - RxJs and Observables.*

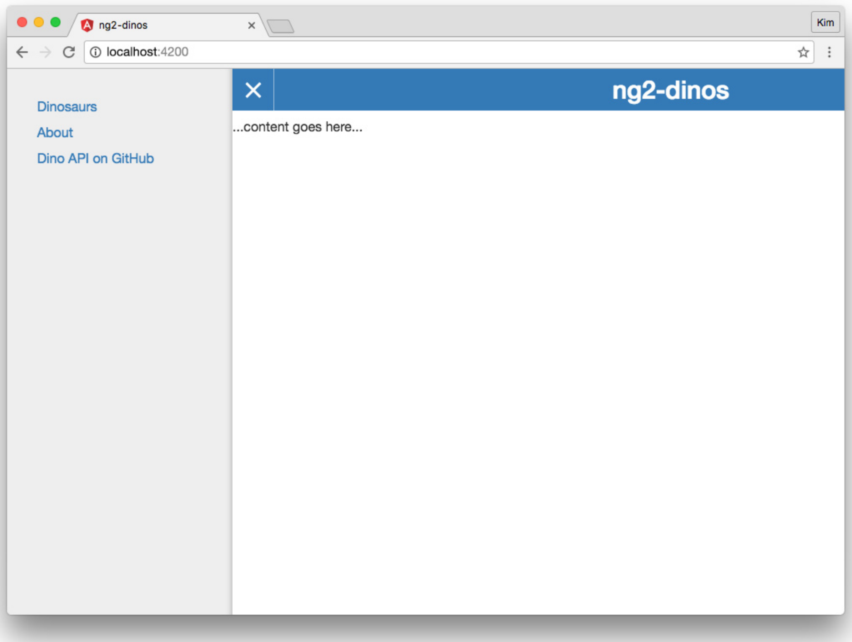
8.2 Add DOM Property to App Component Template

We can then bind `minHeight` to the `[style.min-height]` DOM property on the layout canvas element in `app.component.html`:

```
<!-- ng2-dinos/src/app/core/app.component.html -->

...
  <div
    class="layout-canvas"
    [ngClass]="{'nav-open': navOpen, 'nav-closed': !navOpen}"
    [style.min-height]="minHeight">
...
```

Note: *Angular 2 binds to* ***DOM properties****, not HTML attributes. This may seem counter-intuitive because we're declaratively adding things like* `[disabled]` *or* `[style.min-height]` *to our markup, but these refer to properties, not attributes. Please read Binding syntax: An overview to learn more.*

Now our app should be the height of the window even if the content is short. If the navigation grows longer than the content, the CSS we imported from ng1-dinos will ensure that it gets a scrollbar. With the menu open, our app should look like this in the browser:



...content goes here...

Figure 8.1: Angular 2 app with off-canvas navigation, final

Chapter 9

Angular 2 Footer Component

We have a header, so let's add the simple footer from ng1-dinos too. Run the `ng g` command to create a new component:

```
$ ng g component footer
```

9.1 Footer Component TypeScript

The `footer.component.ts` should be very simple. There's no dynamic functionality; we just need to create the component and display it. Let's simplify the `FooterComponent` class:

```
// ng2-dinos/src/app/footer/footer.component.ts

import { Component } from '@angular/core';

@Component({
  selector: 'app-footer',
  templateUrl: './footer.component.html',
  styleUrls: ['./footer.component.scss']
})
export class FooterComponent {}
```

9.2 Footer Component Template

We can copy the footer markup from ng1-dinos `footer.tpl.html` to our ng2-dinos `footer.component.html` file. We just need to update the link so that it references ng2-dinos instead of ng1-dinos:

```
<!-- ng2-dinos/src/app/footer/footer.component.html -->

<p>
  <small>MIT 2017 | <a href="https://github.com/auth0-blog/ng2-dinos">ng2-dinos @ GitHub</a
</p>
```

9.3 Footer Component Styles

The ng1-dinos footer SCSS comes from ng1-dinos `_footer.scss`. We need to add `@import`s so our Angular 2 component can access global layout variables and responsive mixins. We're also going to change `.footer` to the special `:host` selector since `.footer` no longer exists and we need to style the component's host element:

```
/* ng2-dinos/src/app/footer/footer.component.scss */

/*--------------------
        FOOTER
--------------------*/

@import '../../assets/scss/partials/layout.vars';
@import '../../assets/scss/partials/responsive.partial';

:host {
  padding: $padding-screen-small;
  text-align: center;

  @include mq($large) {
    padding: $padding-screen-large;
  }
}
```

9.4 Add Footer to App Component Template

Finally, we'll add the `<app-footer>` element to the `app.component.html`:

```
<!-- ng2-dinos/src/app/core/app.component.html -->
```

```
...
    <!-- FOOTER -->
    <app-footer></app-footer>
...
```

Restart `ng serve` and we should see the simple footer in our app.

Chapter 10

Migrating Angular 2 Pages

We've implemented some links in our navigation, but we don't have pages to display when the links are clicked. Let's create some components so we can implement routing.

10.1 Create Home, About, and 404 Page Components

In order to implement routing, the first thing we need is multiple pages. Let's quickly create home, about, and 404 components. These will be pages so create a subdirectory in the `ng2-dinos/src/app/` folder called `pages`. Stop the server and execute the following commands:

- Home page component: `$ ng g component pages/home`
- About page component: `$ ng g component pages/about`
- 404 page component: `$ ng g component pages/error404`

10.2 Add Title Provider to App Module

We want to update the document `<title>` tag for each page. Recall that `<title>` is outside the `<app-root>` element in the document `<head>`, but Angular 2 provides a useful service to set the title.

We want the `Title` service to be registered in the root injector so it's available to the entire application. Let's add it to our `app.module.ts`:

```
// ng2-dinos/src/app/app.module.ts
```

```
import { BrowserModule, Title } from '@angular/platform-browser';
...

@NgModule({
  ...
  providers: [
    Title
  ]
})
export class AppModule { }
```

To learn more about this, read the Angular 2 docs on Dependency Injection.

10.3 Add Title to Page Components

The page components should each display a heading and update the `<title>` with the `Title` service we provided in the step above. Let's implement this in each of our new page components. We *don't* have to provide `Title` at the component level (`@Component({ providers: [Title]...)` because we're providing it at an application level in `app.module.ts` (above).

Open the `home.component.ts` file and make the following changes:

```
// ng2-dinos/src/app/pages/home/home.component.ts

import { Component, OnInit } from '@angular/core';
import { Title } from '@angular/platform-browser';

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

  constructor(private titleService: Title) { }

  ngOnInit() {
    this.titleService.setTitle(this.pageName);
  }

}
```

First we'll import the `Title` class from `@angular/platform-browser`. In our `HomeComponent` class, we'll create a `pageName` property and set it to "Dinosaurs".

Then we'll add the `private titleService: Title` to our constructor function. In our `ngOnInit()` function, we'll set the title to the `pageName`. You can consult the Angular 2 docs to learn more about the Title service.

Now let's do the same for the about and 404 components: `about.component.ts` and `error404.component.ts`. We'll also *delete* the `about.component.scss` and `error404.component.scss` files and any references to them. The about and 404 components will be plain pages with some static copy. We can use Bootstrap classes to style both and don't need componetized SCSS.

Now open the about component `about.component.ts`:

```
// ng2-dinos/src/app/pages/about/about.component.ts

import { Component, OnInit } from '@angular/core';
import { Title } from '@angular/platform-browser';

@Component({
  selector: 'app-about',
  templateUrl: './about.component.html'
})
export class AboutComponent implements OnInit {
  pageName = 'About';

  constructor(private titleService: Title) { }

  ngOnInit() {
    this.titleService.setTitle(this.pageName);
  }

}
```

Finally we'll update the error404 component `error404.component.ts`:

```
// ng2-dinos/src/app/pages/error404/error404.component.ts

import { Component, OnInit } from '@angular/core';
import { Title } from '@angular/platform-browser';

@Component({
  selector: 'app-error404',
  templateUrl: './error404.component.html'
})
export class Error404Component implements OnInit {
  pageName = '404 Page Not Found';

  constructor(private titleService: Title) { }
  ngOnInit() {
    this.titleService.setTitle(this.pageName);
  }

}
```

10.4 Home Component Template

Now we have a document title but we also want to display `pageName` in a heading in our HTML. Let's write some basic markup.

In the `home.component.html` file, add an `<article>` and a heading with an interpolated binding to display `pageName`.

```
<!-- ng2-dinos/src/app/pages/home/home.component.html -->

<article id="content-wrapper" class="content-wrapper">
  <h2 class="content-heading">{{pageName}}</h2>

</article>
```

We'll add a lot more to this component later.

10.5 About Component Template

Let's add some basic information about our app in the `about.component.html` template:

```
<!-- ng2-dinos/src/app/pages/about/about.component.html -->

<article id="content-wrapper" class="content-wrapper lead">
  <h2 class="content-heading">{{pageName}}</h2>

  <p><strong>ng2-dinos</strong> is a sample application built with Angular 2 with the follow

  <ul>
    <li>Routing</li>
    <li>Dynamic <code>&lt;title&gt;</code> metadata</li>
    <li>External <code>GET</code> API</li>
    <li>Custom off-canvas navigation</li>
    <li>Filtering by predicate</li>
    <li>Bootstrap</li>
    <li>SCSS</li>
    <li>Angular CLI (Webpack) build</li>
  </ul>

  <p>Download the code for this app from the <a ng-href="http://github.com/auth0-blog/ng2-di
</article>
```

10.6 404 Component Template

This component will show when the route the user attempts to access does not exist. We'll apply a couple of Bootstrap classes in the `error404.component.html` template:

```
<!-- ng2-dinos/src/app/pages/error404/error404.component.html -->

<article id="content-wrapper" class="content-wrapper">
  <h2 class="content-heading text-danger">{{pageName}}</h2>

  <p class="lead">The page you are attempting to access does not exist.</p>
</article>
```

Note: *Our Angular 1 ng1-dinos app had classes like `.home-wrapper` and `.about-wrapper` on the article elements but Angular 2's view encapsulation negates the need for this!*

Chapter 11

Routing in Angular 2

Routing is an essential feature of our ng1-dinos app. For ng2-dinos, we're going to create a new `@NgModule` to handle routing. This gives us more flexibility to expand routing later, if needed, without bloating the `app.module.ts`.

11.1 Create a Routing Module

Because of how the CLI generates multiple files per component in its own subdirectory, sometimes it's more straightforward to create a new feature manually. Regardless, we should know how to do this. Let's create a routing module in the `ng2-dinos/src/app/core/` folder. We'll name this file `app-routing.module.ts`:

```
// ng2-dinos/src/app/core/app-routing.module.ts

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

import { HomeComponent } from '../pages/home/home.component';
import { AboutComponent } from '../pages/about/about.component';
import { Error404Component } from '../pages/error404/error404.component';

@NgModule({
  imports: [
    RouterModule.forRoot([
      {
        path: '',
        component: HomeComponent
      },
      {
        path: 'about',
        component: AboutComponent
      },
      {
        path: '**',
        component: Error404Component
      }
    ])
  ],
  exports: [
    RouterModule
  ]
})
export class AppRoutingModule {}
```

At its heart, this doesn't look much different from the Angular 1 route config at ng1-dinos `app.config.js`. We declare a path and a component that should display when routed to that path. We need to import the `RouterModule` as well as any components we want to use. The wildcard path `**` should be the last one.

Note: *You can read more about routing in the Angular 2 docs. At time of writing, the docs are the most reliable source of information on the Angular 2 router. When searching for blog articles or Stack Overflow answers, be mindful of publish dates and versioning: the Angular 2 router was one of the last pieces to reach completion and has undergone rewrites and breaking changes throughout the beta and release candidate phases.*

Let's take a quick break to verify our `ng2-dinos/src/app` file structure:

```
ng2-dinos
  |-src/
    |-app/
      |-core/
        |-app.component[.html|.scss|.ts]
        |-app-routing.module.ts
      |-header/
        |-_nav.scss
        |-header.component[.html|.scss|.ts]
      |-footer/
        |-footer.component[.html|.scss|.ts]
      |-pages/
        |-about/
          |-about.component[.html|.ts]
        |-error404/
          |-error404.component[.html|.ts]
        |-home/
          |-home.component[.html|.scss|.ts]
      |-app.module.ts
```

11.2 Import Routing Module in App Module

We have a new module to handle routing but it isn't being imported anywhere in our app right now. We need to add it to our `app.module.ts`:

```
// ng2-dinos/src/app/app.module.ts

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

@NgModule({
  declarations: [
    ...
  ],
  imports: [
    ...,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Import the `AppRoutingModule` class and then add it to the `imports` array.

11.3 Display Routed Components

Routing is now configured! Now we just need to display our routed components in the view. In Angular 1, this was done with the `ng-view` directive. In Angular 2, we'll add the `<router-outlet>` element where we want our page components to display in our `app.component.html` template:

```
<!-- ng2-dinos/src/app/core/app.component.html -->

...
    <!-- CONTENT -->
    <div id="layout-view" class="layout-view">
        <router-outlet></router-outlet>
    </div>
...
```

If we serve and view the app in the browser, we should see the home component when we visit http://localhost:4200.

11.4 Route Navigation

Right now, we don't have any live links to our routes. We still need to make some updates to the `header.component.html` to enable route navigation and active link highlighting.

Our Angular 1 ng1-dinos `Header.ctrl.js` had to utilize a custom `navIsActive(path)` function to compare the URL path with the link `href` to apply an `active` class in the navigation markup. The Angular 2 router can do this for us!

Open the `header.component.html` file and let's make some changes to the first two links in the menu:

```html
<!-- ng2-dinos/src/app/header/header.component.html -->

...
    <ul class="nav-list">
      <li>
        <a
          routerLink="/"
          routerLinkActive="active"
          [routerLinkActiveOptions]="{ exact: true }">Dinosaurs</a>
      </li>
      <li>
        <a routerLink="/about" routerLinkActive="active">About</a>
      </li>
      ...
```

In Angular 1, we used the `ng-href` directive. In Angular 2, we'll use the routerLink directive instead. We can also add `routerLinkActive="[active-class-name]"` and Angular 2 will automatically apply our desired class to the link when that route is active.

Note: *The caveat is that this needs an additional option when dealing with the root URL. The routerLinkActive directive returns a match if the `routerLink` is contained in the URL tree. This means that `routerLink="/"` is also matched by all other routes with a / in them. To enable exact matching, we need to add `[routerLinkActiveOptions]="{ exact: true }"` to our root link.*

Now we should be able to click the links in the off-canvas menu and be routed appropriately with proper active link classes. Try it out.

11.5 Router Events

You probably noticed that there's still one thing missing that ng1-dinos had: automatic navigation closing on route change. We definitely don't want to manually close the off-canvas menu every time we switch pages.

In ng1-dinos, we used `$scope.$on('$locationChangeStart', ...)` in the `navControl` directive to bind a handler and close the menu. Something similar exists in Angular 2, so let's implement it!

11.6 Auto-close Menu in Header Component

We'll do this in our `header.component.ts` file where we emitted the event earlier to notify the app component parent. This way we can ensure that both components know about the change and the nav states don't get out of sync:

```
// ng2-dinos/src/app/header/header.component.ts

...
import { Router, NavigationStart } from '@angular/router';

@Component({
  selector: 'app-header',
  templateUrl: './header.component.html',
  styleUrls: ['./header.component.scss']
})
export class HeaderComponent implements OnInit {
  ...
  constructor(private router: Router) { }

  ngOnInit() {
    this.router.events
      .filter(event => event instanceof NavigationStart && this.navOpen)
      .subscribe(event => this.toggleNav());
  }
  ...
```

We need to import `Router` and `NavigationStart` from `@angular/router`. Next we need to make `private router: Router` available in our constructor function.

Router.events is an observable of route events. We'll filter for when the event is an instance of `NavigationStart` and the navigation is open. We'll then subscribe to it to set `navOpen` to `false`.

Now when we click on links in the menu the correct component displays and the navigation closes. Our app homepage now looks like this:

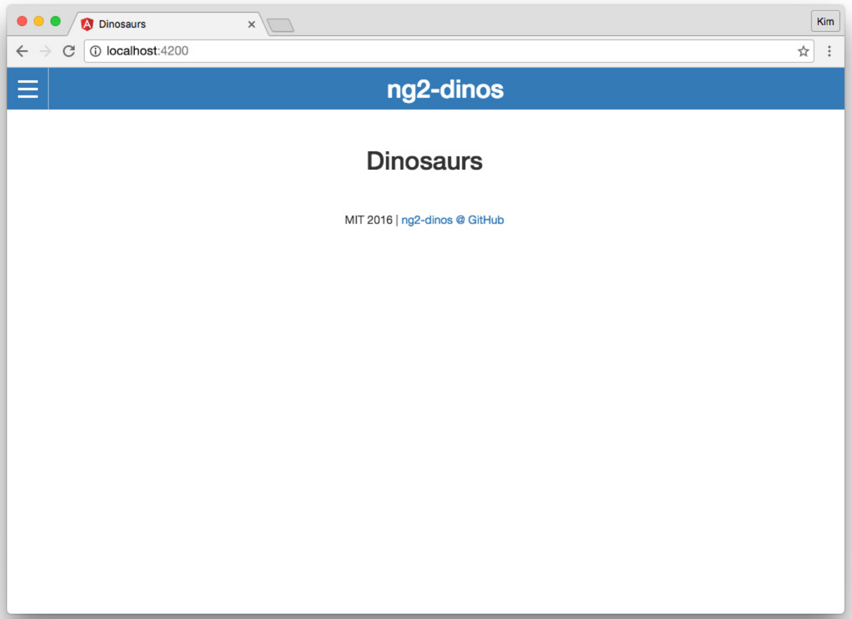


Figure 11.1: Angular 2 ng2-dinos app with basic routing

Chapter 12

Calling an API in Angular 2

Now our architecture and navigation is in place! We've arrived at the business logic portion of our app. Angular 1 ng1-dinos used a service to call the API: `Dinos.service.js`.

We're going to author a service for this in our Angular 2 migration too. Let's start by creating the file. Use the following Angular CLI command to create a service boilerplate:

```
$ ng g service core/dinos
```

When we run this command, note the warning output informing us that the service was generated but not provided. We'll provide it at the component level this time instead of application-wide like we did with the `Title` service. This means we *won't* put `DinosService` in the `app.module.ts`.

The purpose of `DinosService` is to call the API and get dinosaur information. To do this, we'll use HTTP observables. We also need to create TypeScript *models* for our fetched data.

12.1 Dinosaur API Data Model

Let's create a model for the data we're going to retrieve for the main listing of dinosaurs. In order to do this, we need to know the format of the API response. We can determine this simply by making an API request in the browser (and consulting the sample-nodeserver-dinos API README).

The API route we want to use is http://localhost:3001/api/dinosaurs. Assuming you have the API running locally, let's access this route in the browser and look at the response:

Note: *You may want to install/enable a JSON formatting browser extension to view the response.*

```
// http://localhost:3001/api/dinosaurs

[
  {
    "id": 1,
    "name": "Allosaurus"
  },
  {
    "id": 2,
    "name": "Apatosaurus"
  },
  {
    "id": 3,
    "name": "Brachiosaurus"
  },
  ...
]
```

We can see that the response is an array of dinosaur objects. Each dinosaur has an `id` and a `name`. We can see the `id` is a number and the `name` is a string. Now we can create a model.

We'll have more than one model, so let's create a folder for models to keep our app scalable: `ng2-dinos/src/app/core/models/`. In this folder, we'll make our model file: `dino.model.ts`.

```
export class Dino {
  constructor(
    public id: number,
    public name: string
  ) { }
}
```

12.2 Add HTTP Client Module to App Module

Now we have the "shape" of a dinosaur defined. Let's work on getting the data from the API.

First we need to import the `HttpClientModule` in our `app.module.ts`:

```
// ng2-dinos/src/app/app.module.ts

...
import { HttpClientModule } from '@angular/common/http';
```

```
...
@NgModule({
  ...,
  imports: [
    ...,
    HttpClientModule
  ],
  ...
```

Import `HttpClientModule` from `@angular/common/http` and then add it to the NgModule's `imports` array.

12.3 Get API Data with Dinos Service

Now we have the "shape" of a dinosaur defined. Let's work on getting the data from the API in our `dinos.service.ts`:

```
// ng2-dinos/src/app/core/dinos.service.ts

import { Injectable } from '@angular/core';
import { HttpClient, HttpErrorResponse } from '@angular/common/http';
import { Observable } from 'rxjs/Rx';
import 'rxjs/add/operator/catch';

import { Dino } from './models/dino.model';

@Injectable()
export class DinosService {
  private baseUrl = 'http://localhost:3001/api/';

  constructor(private http: HttpClient) { }

  getAllDinos$(): Observable<Dino[]> {
    return this.http
      .get(`${this.baseUrl}dinosaurs`)
      .catch(this.handleError);
  }

  private handleError(err: HttpErrorResponse | any) {
    let errorMsg = err.message || 'Unable to retrieve data';
    return Observable.throw(errorMsg);
  }

}
```

This is pretty straightforward and it doesn't look that much different from our ng1-dinos Dinos service. Aside from Angular 2 format, the primary difference is that we're returning typed observables instead of promises (and we haven't added the API call to get a single dinosaur's details by `id` yet—we'll do that later).

Starting from the top: we import our dependencies. Services are *injectable*. The CLI adds the `Injectable` class for us. We also need `HttpClient` and `HttpErrorResponse` from `@angular/common/http`, `Observable` from RxJS, and the `catch` operator. Finally we need our `Dino` model.

Note: *RxJS observables are preferable over promises. Angular 2's `http.get` returns an observable but we* could *convert it to a promise with `.toPromise()` if we had to (but we won't in this tutorial).*

We set our private API `baseUrl` property and make `private http: Http` available in the constructor function.

Then we define our `getAllDinos$()` function. The `$` at the end of the function name indicates that an observable is returned and we can subscribe to it. The `getAllDinos$(): Observable<Dino[]>` type annotation declares that we expect an array of items matching the `Dino` model we created previously.

Finally we manage successes and errors. The `map` operator processes the result from the observable. In our case, we're returning the response as JSON. We'll use the `catch` operator to handle failed API responses and generate an observable that terminates with an error.

Note: *In the Angular 1 ng1-dinos Dinos service, the success function checks for an object because some server configurations (such as NGINX) will return a successful XHR response with an HTML error page in the case of an API failure. The front-end promise incorrectly resolves this as the appropriate data. We do not need to do this check in Angular 2 ng2-dinos because we have TypeScript ensuring that the shape of the data matches our `Dino` model. Pay attention to your data though: if you have a response that occasionally changes shape, you'll need to address that in the model so you don't receive errors. You can read more about TypeScript functions and optional parameters here.*

12.4 Provide the Dinos Service in App Module

We want the dinos service to be a singleton. Unlike Angular 1, Angular 2 services can be singletons *or* have multiple instances depending on how they're provided. To create a global singleton, we'll provide the service in the `app.module.ts`:

```
// ng2-dinos/src/app/app.module.ts

...
```

```
import { DinosService } from './core/dinos.service';

@NgModule({
  ...,
  providers: [
    ...,
    DinosService
  ],
  ...
```

We import the `DinosService` and then add it to the `providers` array. It's now available for use in our components.

12.5 Use the Dinos Service in Home Component

Now we have a service that fetches data from the API. We'll use this service in our home component to display a list of dinosaurs. Open the `home.component.ts` file:

```
// ng2-dinos/src/app/pages/home/home.component.ts

...
import { DinosService } from '../../core/dinos.service';
import { Dino } from '../../core/models/dino.model';

@Component({
  ...
})
export class HomeComponent implements OnInit {
  dinos: Dino[];
  error: boolean;
  pageName = 'Dinosaurs';

  constructor(
    private titleService: Title,
    private dinosService: DinosService) { }

  getDinos() {
    this.dinosService
      .getAllDinos$()
      .subscribe(
        res => {
          this.dinos = res;
        },
        err => {
```

```
        this.error = true;
      }
    );
  }

  ngOnInit() {
    this.titleService.setTitle(this.pageName);
    this.getDinos();
  }

}
```

As always, we import our dependencies. We need our new `DinosService` and `Dino` model.

Then we'll implement the functionality to use this service. We'll declare that the `dinos` property should be of type `Dino[]` (an array of items matching the `Dino` model). We'll also create an `error` boolean property. We'll add the `private dinosService: DinosService` to the constructor parameters.

We can then write the `getDinos()` method to subscribe to the `getAllDinos$()` observable and assign the response to the `dinos` property. In the function for error handling, we'll set the `error` property to true.

Finally, we'll call the `getDinos()` method in our `ngOnInit()` function.

12.6 Display a List of Dinosaurs

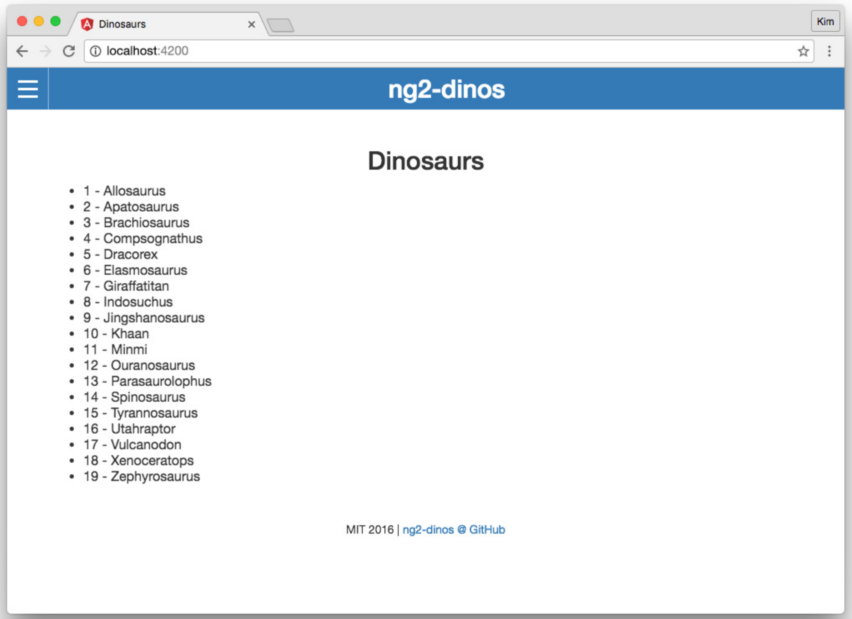
We now have dinosaur data available, we just need to render it in the `home.component.html` template. We'll start by displaying it in a simple unordered list. We also want to show an error if something goes wrong retrieving data from the API:

```
<!-- ng2-dinos/src/app/pages/home/home.component.html -->

...
  <!-- Dinosaurs -->
  <ul *ngIf="dinos">
    <li *ngFor="let dino of dinos">{{dino.id}} - {{dino.name}}</li>
  </ul>

  <!-- Error -->
  <p *ngIf="error" class="alert alert-danger">
    <strong>Rawr!</strong> There was an error retrieving dinosaur data.
  </p>
...
```

The `ng-repeat` of Angular 1 has been replaced by the `ngFor` repeater directive.

Note: *The `*` asterisk before `ngIf` and `ngFor` is syntactic sugar that allows us to skip wrapping subtrees in `<template>` tags. You can read more about `*` and `<template>` in the docs.*

We now have a list of all the dinosaurs returned from the API. Our app homepage looks like this in the browser:



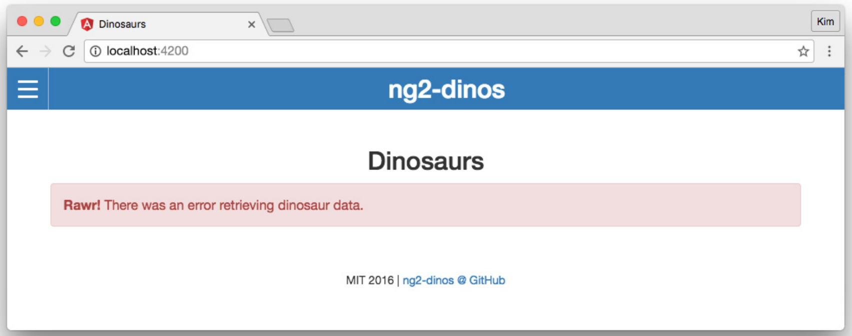
Dinosaurs

- 1 - Allosaurus
- 2 - Apatosaurus
- 3 - Brachiosaurus
- 4 - Compsognathus
- 5 - Dracorex
- 6 - Elasmosaurus
- 7 - Giraffatitan
- 8 - Indosuchus
- 9 - Jingshanosaurus
- 10 - Khaan
- 11 - Minmi
- 12 - Ouranosaurus
- 13 - Parasaurolophus
- 14 - Spinosaurus
- 15 - Tyrannosaurus
- 16 - Utahraptor
- 17 - Vulcanodon
- 18 - Xenoceratops
- 19 - Zephyrosaurus

MIT 2016 | ng2-dinos @ GitHub

Figure 12.1: Angular 2 ng2-dinos app showing list with API data

We can also test the error state by stopping the local Node dinos server and then reloading our Angular 2 app. We should see this:



Dinosaurs

Rawr! There was an error retrieving dinosaur data.

MIT 2016 | ng2-dinos @ GitHub

Figure 12.2: Angular 2 ng2-dinos app showing API data retrieval error

Chapter 13

Display Dino Cards

Our Angular 1 ng1-dinos app repeats a `dinoCard.dir.js` directive with a template that displays each dinosaur's name and detail link in a card styled with Bootstrap. The implementation in ng2-dinos will be similar.

We'll start by generating the new dino card component in the same folder as our home component:

```
$ ng g component pages/home/dino-card
```

13.1 Dino Card Component TypeScript

The dino card won't have to do much processing, but we want to use the `@Input` decorator to give it dinosaur data. Let's set this up in the `dino-card.component.ts`:

```
// ng2-dinos/src/app/pages/home/dino-card/dino-card.component.ts

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

import { Dino } from '../../../core/models/dino.model';

@Component({
  selector: 'app-dino-card',
  templateUrl: './dino-card.component.html'
})
export class DinoCardComponent {
  @Input() dino: Dino;
}
```

We need to import `Input` from `@angular/core`. We also need our trusty `Dino` model. Then we'll declare our `@Input() dino: Dino` typed property. We don't need to add anything to the constructor so the `constructor() { }` function can be deleted. We also aren't using the `OnInit` lifecycle hook so we can remove it from imports, the exported class, and the `ngOnInit()` function. Keep in mind that if we expand functionality at some future date, we may need to replace things we've cleaned up for brevity.

13.2 Dino Card Component Template

Let's create the template for the dino card component. This file will be very similar to the ng1-dinos dino card template:

```html
<!-- ng2-dinos/src/app/pages/home/dino-card/dino-card.component.html -->

<div class="dinoCard panel panel-info">
  <div class="panel-heading">
    <h3 class="panel-title text-center">{{dino.name}}</h3>
  </div>
  <div class="panel-body">
    <p class="text-center">
      <a class="btn btn-primary" href>Details</a>
    </p>
  </div>
</div>
```

Notice that the Details button doesn't go anywhere yet. We'll hook this up when we add the dinosaur detail component and routing.

13.3 Display Dino Card in Home Component

Now let's replace the unordered list with our new dino card component in `home.component.html`:

```html
<!-- ng2-dinos/src/app/pages/home/home.component.html -->

...
  <!-- Dinosaurs -->
  <section *ngIf="dinos" class="row">
    <div class="col-xs-12 col-sm-4" *ngFor="let dino of dinos">
      <app-dino-card [dino]="dino"></app-dino-card>
    </div>
  </section>
...
```

We'll add some Bootstrap classes so our cards display nicely in a grid. Then we'll implement the `<app-dino-card>` element in our repeater. We'll pass `dino` data to it with property binding.

Our ng2-dinos homepage now looks like this:

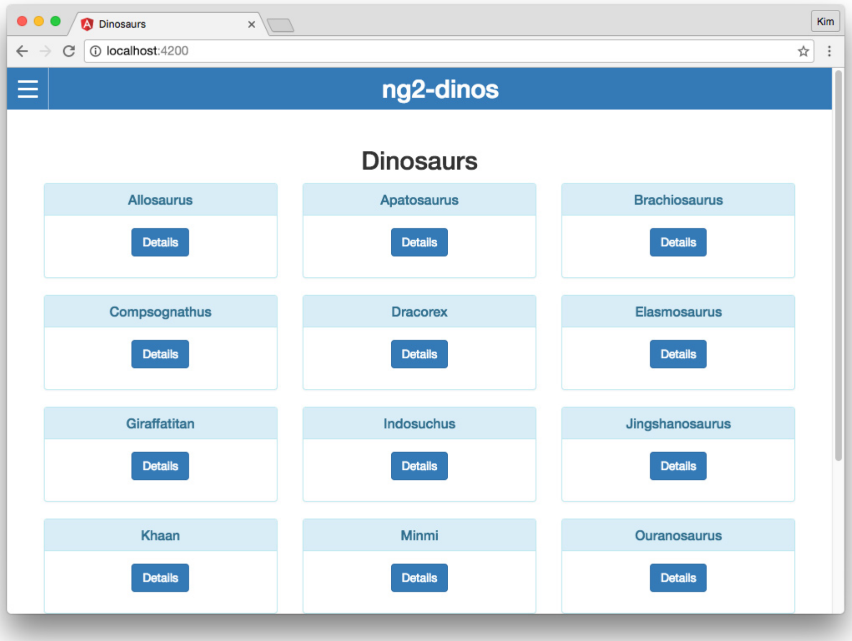


Figure 13.1: Angular 2 ng2-dinos app showing child component cards with API data

Our migration is coming together. The Angular 2 app is finally starting to look more like ng1-dinos!

Chapter 14

Migrating Angular 1 Filtering to Angular 2

You may have heard about Angular 2 pipes. Pipes transform displayed values within a template. In Angular 1, we used the pipe character (`|`) to do similar things with filters. However, filters are *gone* in Angular 2.

14.1 No Filter or OrderBy Pipes

In our Angular 1 ng1-dinos app, we could filter our dinosaurs repeater by binding an `ng-model="query"` to an input and then using `item in array | filter: query` on the repeater. This is no longer built-in in Angular 2. The Angular 2 team recommends *against* replicating this functionality with a custom filtering pipe due to concerns over performance and minification.

Instead, we'll create a *service* that performs filtering. You may already be familiar with filtering this way on Angular 1 apps with large amounts of data where performance becomes an issue. Angular 1 apps can slow to a crawl if care isn't taken with how filtering is handled. If you've ever had to search hundreds or thousands of items or implemented faceted search, you should be familiar with the pitfalls and workarounds.

Note: *How is a filtering service different from a custom pipe? Filtering lists is very expensive. With a service, we can control when and how often the filtering logic is executed. You can read more in the "No* `FilterPipe` *or* `OrderByPipe`*" section of the Pipes docs (at the very bottom).*

14.2 Create a Filter Service

Let's create a service for filtering:

```
$ ng g service core/filter
```

We want our filter service to provide a `search()` method that accepts an array and a query string. It should check objects in the array for strings that contain the query and return a new array of all objects with a match. Let's implement this in `filter.service.ts`:

```
// ng2-dinos/src/app/core/filter.service.ts

import { Injectable } from '@angular/core';

@Injectable()
export class FilterService {
  search(array: any[], query: string) {
    const lQuery = query.toLowerCase();

    if (!query) {
      return array;
    } else if (array) {
      const filteredArray = array.filter(item => {
        for (const key in item) {
          if ((typeof item[key] === 'string') && (item[key].toLowerCase().indexOf(lQuery) !=
            return true;
          }
        }
      });
      return filteredArray;
    }
  }

}
```

We want search to be case-insensitive so we'll convert the query and values to lowercase when checking for matches. If the method is called with a falsey query, we'll return the original array instead of trying to check for matches. For our ng2-dinos search, we're only going to check string values in the objects. If you need a more robust search (ie., one that also checks dates, numbers, etc.) you'll want to handle that specifically. This is one of the benefits of implementing filters this way over the old Angular 1 filter: we have more fine-grained control.

14.3 Use Angular 2 Filter Service to Search

Now that we have a way to filter by query, let's implement this in our home component.

14.3.1 Filter in Home Component TypeScript

Open the `home.component.ts` file:

```
// ng2-dinos/src/app/pages/home/home.component.ts

...
import { FilterService } from '../../core/filter.service';

@Component({
  ...
  providers: [DinosService, FilterService]
})
export class HomeComponent implements OnInit {
  dinos: Dino[];
  filteredDinos: Dino[];
  error: boolean;
  pageName = 'Dinosaurs';
  query = '';

  constructor(..., private filterService: FilterService) { }

  getDinos() {
    this.dinosService.getAllDinos$()
      .subscribe(
        res => {
          this.dinos = res;
          this.filteredDinos = res;
        },
        err => {
          this.error = true;
        }
      );
  }

  ngOnInit() {
    this.titleService.setTitle(this.pageName);
    this.getDinos();
  }
  filterDinos() {
    this.filteredDinos = this.filterService.search(this.dinos, this.query);
  }

  resetQuery() {
    this.query = '';
    this.filteredDinos = this.dinos;
  }

  get noSearchResults() {
    return this.dinos && !this.filteredDinos.length && this.query && !this.error;
  }

}
```

We need to import and then provide our `FilterService`. Next we'll set its parameter in the constructor function. Now we can use it in our home component.

Note: *By providing the filter service in the component instead of `app.module.ts`, we're creating an instance unique to this component. We're doing this here because there is only one place we're filtering. If you add filters to additional components in the future, consider using a global singleton if there's no compelling reason to create multiple instances.*

We're going to create a property called `filteredDinos` alongside our `dinos` property. The filtered collection should also have the `Dino[]` type. When we successfully retrieve data from the API, we'll set `filteredDinos` as well as `dinos`. At this point it is the full collection.

Next we need a method for the template to use to filter the dinosaur list. We'll call this method `filterDinos()`. Inside this function, we'll pass the `query` and our full `dinos` collection to the `FilterService` method we created and set its results: `this.filteredDinos = this.filterService.search(this.dinos, this.query)`.

Our ng1-dinos app has a way to instantly clear the search with a button. We want the same feature in ng2-dinos, so let's create a `resetQuery()` method. This method sets the `query` to an empty string and then sets `filteredDinos` to the original, unfiltered `dinos` array. The reason we have to manually reset the array is because we're going to declaratively run `filterDinos()` on `keyup` in the query input field. This won't be triggered when the user clicks the button to clear the query.

Finally, we need a method that returns an expression informing the template that no search results match the query. If there is a `dinos` array, the `filteredDinos` array is empty, there is a query, and (as a catch-all), there is no API error, then we can conclude the user's search has produced no results. In our ng1-dinos app, we used this expression in the `ng-if` in the view. Angular 2 recommends shifting logic of this type into the component.

14.3.2 Filter in Home Component Template

You can reference the Angular 1 ng1-dinos `Home.view.html` to check out the markup for searching. We're going to copy and then modify it for ng2-dinos `home.component.html`:

```
<!-- ng2-dinos/src/app/pages/home/home.component.html -->

...
  <!-- Search dinosaurs -->
  <section *ngIf="dinos" class="home-search input-group">
    <label class="input-group-addon" for="search">Search</label>

    <input
      id="search"
      type="text"
      class="form-control"
      [(ngModel)]="query"
      (keyup)="filterDinos()" />

    <span class="input-group-btn">
      <button
        class="btn btn-danger"
        (click)="resetQuery()"
        [disabled]="!query">&times;</button>
    </span>
  </section>

  <!-- Dinosaurs -->
  <section *ngIf="dinos" class="row">
    <div class="col-xs-12 col-sm-4" *ngFor="let dino of filteredDinos">
      <app-dino-card [dino]="dino"></app-dino-card>
    </div>
  </section>

  <!-- No search results -->
  <p *ngIf="noSearchResults" class="alert alert-warning">
    No information available on a dinosaur called <em class="text-danger">{{query}}</em>, s
  </p>
...
```

We want to use two-way binding with `ngModel` to bind the `query` to the search input. On the `keyup` event, we'll run our `filterDinos()` function. This will update the `filteredDinos` array. We also have a button to clear the search query. On `click`, we'll execute `resetQuery()`. If there's no query, we can disable the button.

Note: *`ngModel` now requires the `FormsModule` from `@angular/forms`. The Angular CLI creates new projects with this dependency in `app.module.ts` automatically but it's important to know why and how we utilize it in our app.*

In order for our filtering to work in the template, we need to update the `*ngFor` repeater to use the `filteredDinos` array instead of the `dinos` array.

We also want to show a message if a user searches and there are no matching results. This message should show if the `noSearchResults` getter returns `true`.

14.3.3 Filter in Home Component Styles

If we view our app, you may notice we could use a bit of styling to put some space between the search and the dinosaur list. Open the `home.component.scss` file and add:

```
/* ng2-dinos/src/app/pages/home/home.component.scss */

/*--------------------
         HOME
--------------------*/

.home-search {
  margin-bottom: 20px;
}
```

We should now be able to search for dinosaurs by name:

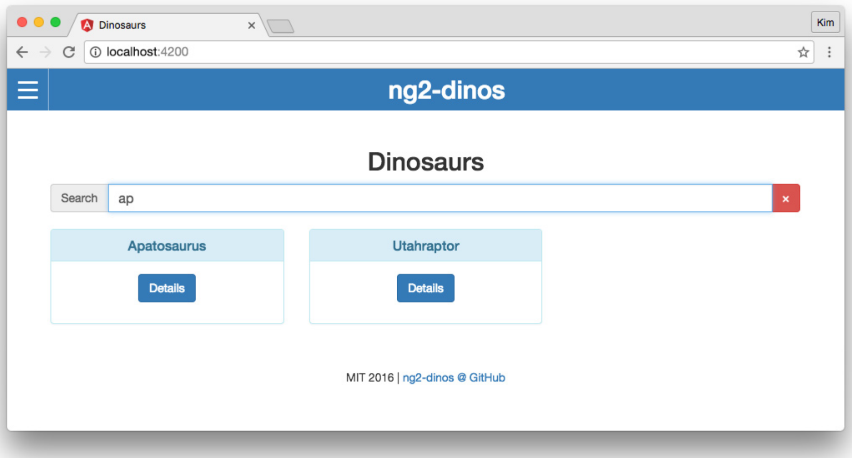
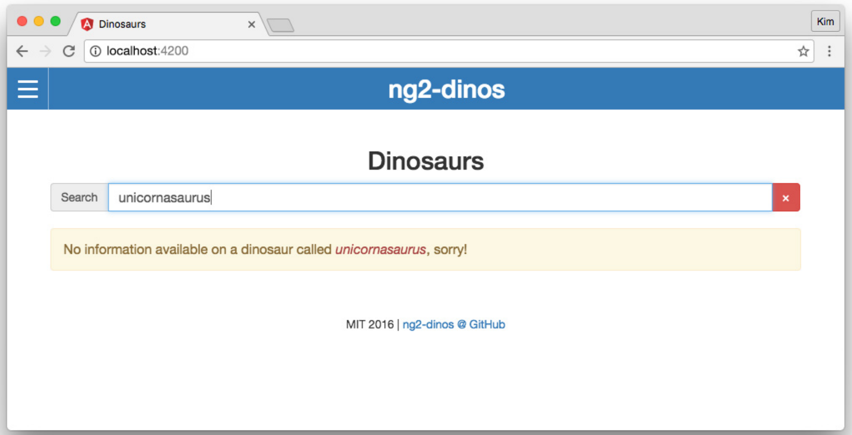


Figure 14.1: Angular 2 ng2-dinos app with search filtering

If the search doesn't return any results, we should see a message:



No information available on a dinosaur called *unicornasaurus*, sorry!

Figure 14.2: Angular 2 ng2-dinos app with search filtering showing no results found

Chapter 15

Migrating Detail Component to Angular 2

Our Angular 1 ng1-dinos app shows a dinosaur's details when we click on one in the homepage listing. We'll implement this in our Angular 2 app now.

Let's create a new detail component:

```
$ ng g component pages/detail
```

15.1 Routing with Parameters

Let's make our detail component accessible in the application. We want to show the detail page with a dinosaur ID, like this: `http://localhost:4200/dinosaur/5`. Open the `app-routing.module.ts` file:

```
// ng2-dinos/src/app/core/app-routing.module.ts

...
import { DetailComponent } from '../pages/detail/detail.component';

...
    RouterModule.forRoot([
      ...
      {
        path: 'dinosaur/:id',
        component: DetailComponent
      },
      {
        path: '**',
        component: Error404Component
      }
    ])
...
```

We'll import our new detail component and then add a route with an `:id` parameter. This route should be placed above the `**` wildcard route.

15.2 Linking to Routes with Parameters

Now we need to link each dinosaur with its detail page. Open `dino-card.component.html`:

```
<!-- ng2-dinos/src/app/pages/home/dino-card/dino-card.component.html -->

...
    <p class="text-center">
      <a class="btn btn-primary" [routerLink]="['/dinosaur', dino.id]">Details</a>
    </p>
...
```

We'll use the `routerLink` directive with the Details button and bind an array of the URL segments: `[routerLink]="['/dinosaur', dino.id]"`. Now we should be able to click on dinosaur Details in the homepage and see our detail component.

Chapter 16

Calling the API for Data by ID

Our detail component needs to make API calls to retrieve dinosaur data by ID. Let's implement this functionality using a new model and a new observable in the `Dinos` service.

16.1 Create a Dino Details Model

The Dinos Node API supports a route that accepts an ID and returns detailed dinosaur information. Let's create a model for this. Make sure the local Node API is running and we'll test out the route by accessing it in the browser: http://localhost:3001/api/dinosaur/1. The response looks like this:

```
// http://localhost:3001/api/dinosaur/1

{
  "id": 1,
  "name": "Allosaurus",
  "pronunciation": "AL-oh-sore-us",
  "meaningOfName": "other lizard",
  "diet": "carnivorous",
  "length": "12m",
  "period": "Late Jurassic",
  "mya": "156-144",
  "info": "Allosaurus was an apex predator in the Late Jurassic in North America."
}
```

Let's supply a model for this data shape. Create a new file in the `models` directory we created earlier and name it `dino-detail.model.ts`:

```
// ng2-dinos/src/app/core/models/dino-detail.model.ts

export class DinoDetail {
  constructor(
    public id: number,
    public name: string,
    public pronunciation: string,
    public meaningOfName: string,
    public diet: string,
    public length: string,
    public period: string,
    public mya: string,
    public info: string
  ) { }
}
```

16.2 Add HTTP Observable to Get Dinosaur by ID

Next we'll add the HTTP observable to call the API and retrieve the dinosaur data by ID. Let's open our `dinos.service.ts` file and add a new method:

```
// ng2-dinos/src/app/core/dinos.service.ts

...
import { DinoDetail } from './models/dino-detail.model';

...
  getDino$(id: number): Observable<DinoDetail> {
    return this.http
      .get(`${this.baseUrl}dinosaur/${id}`)
      .catch(this.handleError);
  }
...
```

We'll import the `DinoDetail` model we just created. Then we'll create an HTTP observable that accepts an `id: number` as a parameter. The observable has a type annotation of `Observable<DinoDetail>`. The ID parameter is passed to the `GET` request. The handlers we set up in earlier are then used for successes and errors. The `catch` operator will generate an observable that terminates with an error.

16.3 Using API Data in Detail Component

Now we're ready to get and display individual dinosaur information in our detail component.

16.3.1 Detail Component TypeScript

Let's update the `detail.component.ts` file:

```typescript
// ng2-dinos/src/app/pages/detail/detail.component.ts

import { Component, OnInit } from '@angular/core';
import { Title } from '@angular/platform-browser';
import { ActivatedRoute, Params } from '@angular/router';

import { DinosService } from '../../core/dinos.service';
import { DinoDetail } from '../../core/models/dino-detail.model';

@Component({
  selector: 'app-detail',
  templateUrl: './detail.component.html',
  styleUrls: ['./detail.component.scss']
})
export class DetailComponent implements OnInit {
  dino: DinoDetail;
  error: boolean;

  constructor(
    private titleService: Title,
    private dinosService: DinosService,
    private route: ActivatedRoute) { }

  getDino() {
    this.route.params.forEach((params: Params) => {
      let id = +params['id'];   // convert string to number

      this.dinosService.getDino$(id)
        .subscribe(
          res => {
            this.dino = res;
            this.titleService.setTitle(this.dino.name);
          },
          err => {
            this.error = true;
          }
      );
    });
  }

  ngOnInit() {
    this.getDino();
  }

}
```

Most of this should look familiar from implementing our home component.

Let's start by importing our dependencies. We need the `Title` service. We'll also need `ActivatedRoute` and `Params` from `@angular/router` in order to retrieve the route ID parameter to use to get the appropriate dinosaur data from the API. Finally, we'll also need the `DinosService` and `DinoDetail` model.

We'll create a couple of properties: `dino` will utilize the `DinoDetail` model type and `error` is a boolean, like in our `home.component.ts`. Then we'll add dependencies to the constructor function so we can use them.

The `getDino()` method iterates over the available route parameters. We'll convert the `id` string to a number and then pass it to the `getDino$(id)` observable. We'll subscribe to the observable and assign the JSON response to the `dino` property. We'll also set the page title as the dinosaur's `name`. If there's an error retrieving data, we'll simply set the `error` property to `true`.

Finally, we'll call the `getDino()` method in the `ngOnInit()` lifecycle hook.

16.3.2 Detail Component Template

Now we're ready to display the dinosaur detail information in our detail component template. Open the `detail.component.html` file:

```
<!-- ng2-dinos/src/app/pages/detail/detail.component.html -->

<article id="content-wrapper" class="content-wrapper">

  <section *ngIf="dino" id="detail-content-dinosaur" class="panel panel-default">
    <div class="panel-heading">
      <h2 class="text-center">{{dino.name}}</h2>
    </div>

    <ul class="list-group">
      <li class="list-group-item">
        <h4 class="list-group-item-heading">Pronunciation:</h4>
        <p class="list-group-item-text">
          <em>{{dino.pronunciation}}</em>
        </p>
      </li>
      <li class="list-group-item">
        <h4 class="list-group-item-heading">Name Means:</h4>
        <p class="list-group-item-text">{{dino.meaningOfName}}</p>
      </li>
      <li class="list-group-item">
        <h4 class="list-group-item-heading">Length:</h4>
        <p class="list-group-item-text">{{dino.length}}</p>
      </li>
      <li class="list-group-item">
        <h4 class="list-group-item-heading">Diet:</h4>
        <p class="list-group-item-text">{{dino.diet}}</p>
      </li>
      <li class="list-group-item">
        <h4 class="list-group-item-heading">Lived:</h4>
        <p class="list-group-item-text">
          {{dino.period}}<br>
          <em>({{dino.mya}} million years ago)</em>
        </p>
      </li>
    </ul>

    <div class="panel-body">
      <p class="lead" [innerHTML]="dino.info"></p>
    </div>

    <div class="panel-footer">
      <a routerLink="/">&larr; All Dinosaurs</a>
    </div>
  </section>

  <!-- Error -->
  <p *ngIf="error" class="alert alert-danger">
    <strong>Rawr!</strong> There was an error retrieving data for the dinosaur you requeste
  </p>

</article>
```

Like with the other page components we migrated, we don't need a `.detail-wrapper` class in the template. In Angular 1 ng1-dinos we used these classes to "componetize" globally-declared CSS. Angular 2 encapsulates styles by component so we don't need specific wrapper classes anymore.

We'll use Bootstrap to style most of our dinosaur details. Most of our data can be displayed simply using interpolation with double-curly braces. The exception is the `info` paragraph. Our API sometimes returns HTML markup in this string.

In Angular 1 ng-dinos, we used `ng-bind-html` to render markup in bindings. In Angular 2, we need to bind to the `innerHTML` DOM property like so:

```
<p class="lead" [innerHTML]="dino.info"></p>
```

We'll add a link back to the homepage and then finally, show an error message if there was a problem retrieving data from the API.

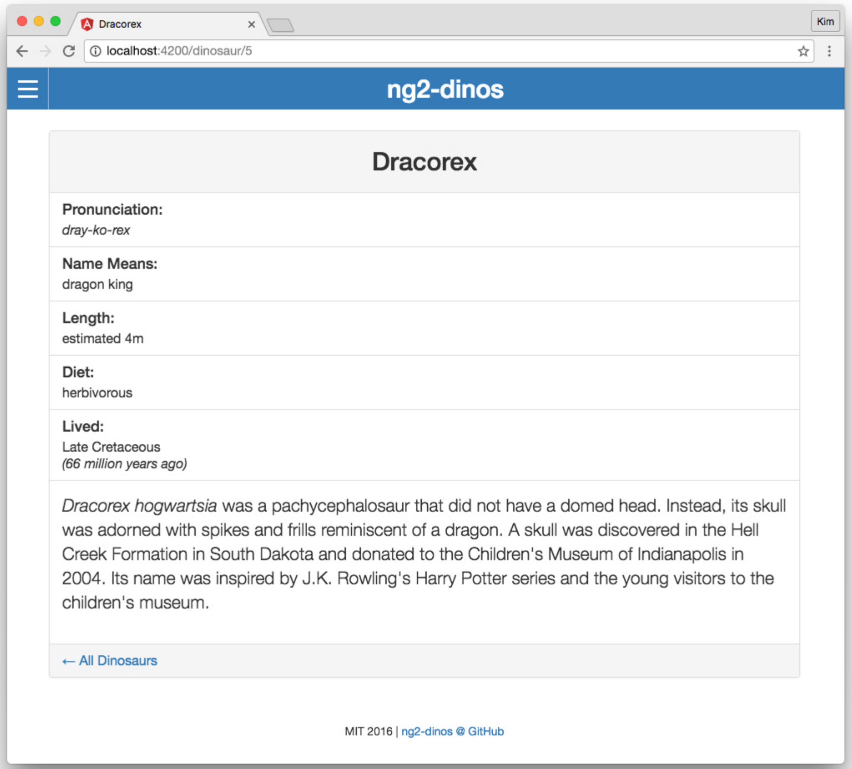
16.3.3 Detail Component Styles

We'll just make one small tweak in the SCSS for our detail component to reduce the amount of extra space above the dinosaur name heading. In the Angular 1 app, the detail page styles were here: ng1-dinos `_detail.scss`.

Our Angular 2 ng2-dinos detail component styles should look like this:

```
/* ng2-dinos/src/app/pages/detail/detail.component.scss */

/*--------------------
       DETAIL
--------------------*/

.panel-heading h2 {
  margin-top: 10px;
}
```

Now we have our detail component! When dinosaur details are clicked on the homepage, the detail pages should look something like this:



Dracorex

Pronunciation:
dray-ko-rex

Name Means:
dragon king

Length:
estimated 4m

Diet:
herbivorous

Lived:
Late Cretaceous
(66 million years ago)

Dracorex hogwartsia was a pachycephalosaur that did not have a domed head. Instead, its skull was adorned with spikes and frills reminiscent of a dragon. A skull was discovered in the Hell Creek Formation in South Dakota and donated to the Children's Museum of Indianapolis in 2004. Its name was inspired by J.K. Rowling's Harry Potter series and the young visitors to the children's museum.

← All Dinosaurs

MIT 2016 | ng2-dinos @ GitHub

Figure 16.1: Angular 2 ng2-dinos dinosaur detail route with parameters

Browse your app to make sure this is working as expected.

Chapter 17

Loading State for API Calls

Our Angular 1 to Angular 2 migration is almost complete! The last piece is a simple loading state that needs to be shown while API calls are resolving. Because we're running our app and API locally, communication between the two is almost instantaneous. In another environment this may not be the case. We'll implement a small loading state to show while data is being retrieved. This will show in the home and detail components.

In Angular 1 ng1-dinos, this loading state was a simple directive at `loading.dir.js`. In Angular 2, we'll create a very similar loading component.

17.1 Loading Image Asset

The first thing we need is the image asset for the loading state. This can be downloaded from the Angular 1 ng1-dinos app here: `raptor-loading.gif`. We'll place this image in our Angular 2 ng2-dinos app in an equivalent location: `ng2-dinos/src/assets/images/`.

17.2 Loading Component TypeScript

The loading component will be one flat file, so we'll add some flags to the CLI to generate it:

```
$ ng g component core/ui/loading --it --is --flat
```

The `--it` flag is shorthand for `inline-template`. The `--is` is shorthand for `inline-styles`, and `--flat` indicates a containing folder should not be generated.

> **Note:** You can also add `--no-spec` when generating CLI files if you don't want test files.

Open the new `loading.component.ts`:

```
// ng2-dinos/src/app/core/ui/loading.component.ts

import { Component } from '@angular/core';

@Component({
  selector: 'app-loading',
  template: '<img class="loading" src="/assets/images/raptor-loading.gif">',
  styles: [`
    .loading {
      display: block;
      margin: 30px auto;
    }
  `]
})
export class LoadingComponent { }
```

It's possible to keep everything we need in the component without external template or style files. Instead of using `templateUrl` we'll use `template`. The template consists of an image tag with our `raptor-loading.gif` file. Instead of `styleUrls`, we can use a `styles` array and add CSS rulesets right in the component. We'll use an ES6 template string literal (in backticks) to maintain readability.

17.3 Add Loading Component to App Module

In order to use our new component in our app, we need to add it to our `app.module.ts`:

```
// ng2-dinos/src/app/app.module.ts

...
import { LoadingComponent } from './core/ui/loading.component';
...

@NgModule({
  declarations: [
    ...
    LoadingComponent,
    ...
  ],
  ...
})
export class AppModule { }
```

We'll import the `LoadingComponent` class and then add it to the `declarations` array. Now we can use the `<app-loading>` element in other components.

17.4 Add Loading Component to Home Component

The Angular 1 ng1-dinos app shows the loading directive in the home and detail views.

17.4.1 Implement Loading Functionality in Home Component TypeScript

In `home.component.ts`, let's add the functionality we need to conditionally add our new loading component:

```
// ng2-dinos/src/app/pages/home/home.component.ts

...

export class HomeComponent implements OnInit {
  ...
  loading: boolean;

  constructor(...) { }

  getDinos() {
    this.dinosService.getAllDinos$()
      .subscribe(
        res => {
          ...
          this.loading = false;
        },
        err => {
          ...
          this.loading = false;
        }
      );
  }

  ngOnInit() {
    ...
    this.loading = true;
    this.getDinos();
  }

  ...

  get isLoaded() {
    return this.loading === false;
  }

}
```

We'll add a boolean `loading` property to track loading state. Loading should be turned off when the API responds either with a success or a failure; we don't want to get stuck in an infinite loading state. We'll add `this.loading = false` in both the `onNext` and `onError` subscription functions.

Note: *This differs from our implemention in the Angular 1 app: ng1-dinos used the promise method `.finally()`. When subscribing to observables, the `onCompleted` function is only executed upon graceful termination of the observable sequence. Unlike `finally()` with promises, it will not run if an exception occurs.*

To initiate the loading state, we'll set the `loading` property to `true` in the `ngOnInit()` lifecycle hook.

Finally, we need a getter method `get isLoaded()` to tell the template when loading has completed. Angular 1 ng1-dinos implemented this expression in the template, but the Angular 2 docs recommend moving this kind of logic to the component.

17.4.2 Implement Loading Functionality in Home Component Template

Now we need to implement our loading component and some template logic in the home markup `home.component.html`:

```
<!-- ng2-dinos/src/app/pages/home/home.component.html -->

<article id="content-wrapper" class="content-wrapper">
  <h2 class="content-heading">{{pageName}}</h2>

  <app-loading *ngIf="loading"></app-loading>

  <div *ngIf="isLoaded">
```

```
    <!-- Search dinosaurs -->
    ...

    <!-- Dinosaurs -->
    ...

    <!-- No search results -->
    ...

    <!-- Error -->
    ...
  </div>

</article>
```

We'll add and remove the loading component with `<app-loading *ngIf="loading">`. We'll also add a container around the rest of the page content and only stamp it if the `isLoaded` getter is true.

When our app home component is loading, it now looks like this:

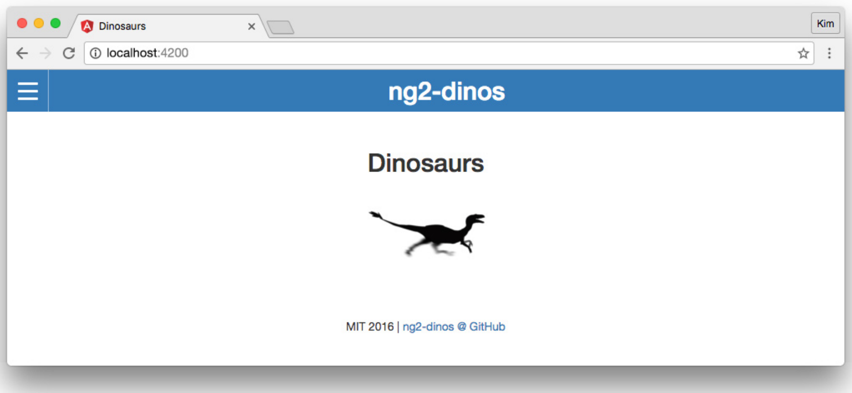


Figure 17.1: Angular 2 ng2-dinos app loading state for API calls

The animated gif shows a running raptor until loading is completed.

17.5 Add Loading Component to Detail Component

Now we'll make similar changes to the detail component to add the loading state.

17.5.1 Implement Loading Functionality in Detail Component TypeScript

Let's open our `detail.component.ts` file:

```
// ng2-dinos/src/app/pages/detail/detail.component.ts

...

export class DetailComponent implements OnInit {
  ...
  loading: boolean;

  constructor(...) { }

  getDino() {
    ...
      this.dinosService.getDino$(id)
        .subscribe(
          res => {
            ...
            this.loading = false;
          },
          err => {
            ...
            this.loading = false;
          }
        );
    ...
  }

  ngOnInit() {
    this.loading = true;
    this.getDino();
  }

  get isLoaded() {
    return this.loading === false;
  }
}
```

We'll make the same changes to our detail component as the home component. We want to add a boolean `loading` property that is `true` on initialization and `false` `onNext` and `onError`. A `get isLoaded()` getter compares the loading state to check if it's been set to `false` and will be used to stamp content in the template.

17.5.2 Implement Loading Functionality in Detail Component Template

Open `detail.component.html`:

```html
<!-- ng2-dinos/src/app/pages/detail/detail.component.html -->

<article id="content-wrapper" class="content-wrapper">

  <app-loading *ngIf="loading"></app-loading>

  <div *ngIf="isLoaded">

    <!-- Dinosaur details -->
    ...

    <!-- Error -->
    ...
  </div>

</article>
```

Let's add the `<app-loading>` element and a wrapper to hide the content while loading is in progress. Now the loading gif should show while we retrieve API data for a dinosaur's detail information.

17.5.3 Remove "Loading..." Text from Index HTML

Finally, we're going to remove the `Loading...` text from our `index.html` file's `<app-root>` element. This is the last thing we'll do to make our Angular 2 migration feature-match our Angular 1 ng1-dinos app:

```html
<!-- ng2-dinos/src/index.html -->

...
<body>
  <app-root></app-root>
```

```
</body>
...
```

Chapter 18

Completed Migration

The migration of our Angular 1 ng1-dinos app to Angular 2 ng2-dinos is now complete! If you have both apps running, they should be functionally equivalent from a user's perspective. Please explore the two apps in the browser to make sure that our migration was successful.

18.1 Refactoring Suggestions

As mentioned before, this is a migration tutorial so one of our goals is to maintain close to 1:1 correlation with ng1-dinos while still implementing Angular 2 best practices. However, there are refactoring opportunities that we shouldn't ignore. Here are my refactoring suggestions from our migration tutorial:

- Consider componetizing more global SCSS, breaking files like `_layout.scss` up into respective `*.component.scss` files and utilizing selectors like `:host` and `:host-context()`.
- You may want to consider using additional `@NgModule`s to manage dependencies. Modules can make dependency management easier. Read the Angular Modules docs and [Use @NgModule to Manage Dependencies in your Angular 2 Apps](https://auth0.com/blog/angular-2-ngmodules/) to learn more.
- You could potentially abstract the template API error markup into its own component. The error message is currently different between the home and detail page components, but you could use data binding to pass a custom string into the component each time it's utilized. This might help with scalability if additional API calls will be made in new components in the future.

Chapter 19

Bonus: Authenticate an Angular App and Node API with Auth0

We can protect our applications and APIs so that only authenticated users can access them. Let's explore how to do this with an Angular application and a Node API using Auth0. You can clone this sample app and API from the angular-auth0-aside repo on GitHub.

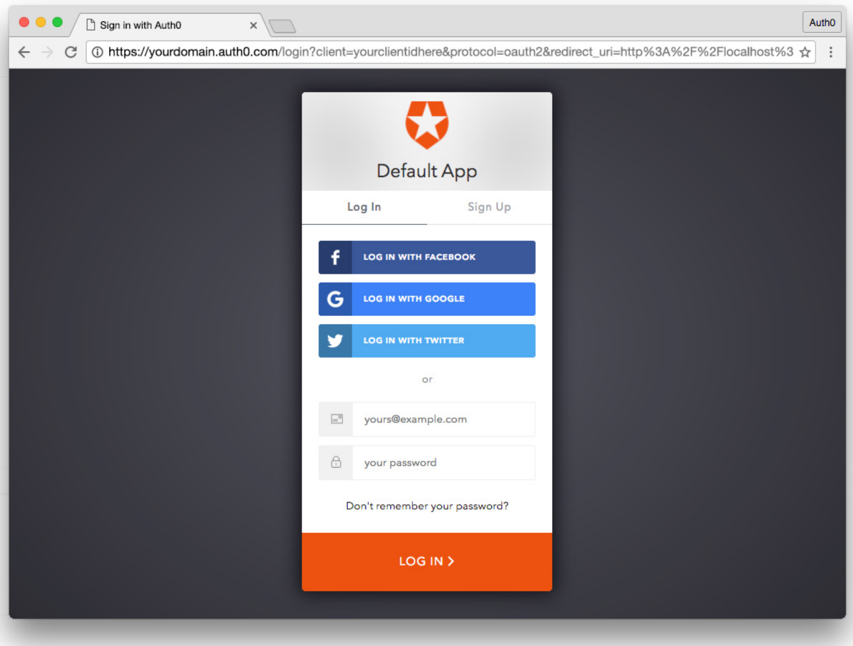


Figure 19.1: Auth0 hosted login screen

19.0.1 Features

The sample Angular application and API has the following features:

- Angular application generated with Angular CLI and served at http://localhost:4200
- Authentication with auth0.js using a hosted Lock instance
- Node server protected API route `http://localhost:3001/api/dragons` returns JSON data for authenticated `GET` requests
- Angular app fetches data from API once user is authenticated with Auth0
- Profile page requires authentication for access using route guards
- Authentication service uses a subject to propagate authentication status events to the entire app
- User profile is fetched on authentication and stored in authentication service
- Access token, ID token, profile, and token expiration are stored in local storage and removed upon logout

19.0.2 Sign Up for Auth0

You'll need an Auth0 account to manage authentication. You can sign up for a free account here. Next, set up an Auth0 client app and API so Auth0 can interface with an Angular app and Node API.

19.0.3 Set Up a Client App

1. Go to your **Auth0 Dashboard** and click the "create a new client" button.
2. Name your new app and select "Single Page Web Applications".
3. In the **Settings** for your new Auth0 client app, add `http://localhost:4200/callback` to the **Allowed Callback URLs** and `http://localhost:4200` to the **Allowed Origins (CORS)**.
4. Scroll down to the bottom of the **Settings** section and click "Show Advanced Settings". Choose the **OAuth** tab and set the **JsonWebToken Signature Algorithm** to `RS256`.
5. If you'd like, you can set up some social connections. You can then enable them for your app in the **Client** options under the **Connections** tab. The example shown in the screenshot above utilizes username/password database, Facebook, Google, and Twitter. For production, make sure you set up your own social keys and do not leave social connections set to use Auth0 dev keys.

19.0.4 Set Up an API

1. Go to **APIs** in your Auth0 dashboard and click on the "Create API" button. Enter a name for the API. Set the **Identifier** to your API endpoint URL. In this example, this is `http://localhost:3001/api/`. The **Signing Algorithm** should be `RS256`.
2. You can consult the Node.js example under the **Quick Start** tab in your new API's settings. We'll implement our Node API in this fashion, using Express, express-jwt, and jwks-rsa.

We're now ready to implement Auth0 authentication on both our Angular client and Node backend API.

19.0.5 Dependencies and Setup

The Angular app utilizes the Angular CLI. Make sure you have the CLI installed globally:

```
$ npm install -g @angular/cli
```

Once you've cloned the project, install the Node dependencies for both the Angular app and the Node server by running the following commands in the root of your project folder:

```
$ npm install
$ cd server
$ npm install
```

The Node API is located in the `/server` folder at the root of our sample application.

Open the `server.js` file:

```
// server/server.js
...
// @TODO: change [CLIENT_DOMAIN] to your Auth0 domain name.
// @TODO: change [AUTH0_API_AUDIENCE] to your Auth0 API audience.
var CLIENT_DOMAIN = '[CLIENT_DOMAIN]'; // e.g., youraccount.auth0.com
var AUTH0_AUDIENCE = '[AUTH0_API_AUDIENCE]'; // http://localhost:3001/api in this example

var jwtCheck = jwt({
    secret: jwks.expressJwtSecret({
      cache: true,
      rateLimit: true,
      jwksRequestsPerMinute: 5,
      jwksUri: `https://${CLIENT_DOMAIN}/.well-known/jwks.json`
    }),
    aud: AUTH0_AUDIENCE,
    issuer: `https://${CLIENT_DOMAIN}/`,
    algorithm: 'RS256'
});
...
//--- GET protected dragons route
app.get('/api/dragons', jwtCheck, function (req, res) {
  res.json(dragonsJson);
});
...
```

Change the `CLIENT_DOMAIN` variable to your Auth0 client domain. The `/api/dragons` route will be protected with express-jwt and jwks-rsa.

> **Note:** To learn more about RS256 and JSON Web Key Set, read Navigating RS256 and JWKS.

Our API is now protected, so let's make sure that our Angular application can also interface with Auth0. To do this, we'll activate the `src/app/auth/auth0-variables.ts.example` file by deleting the `.example` from the file extension. Then open the file and change the `[CLIENT_ID]` and `[CLIENT_DOMAIN]` strings to your Auth0 information:

```
// src/app/auth/auth0-variables.ts
...
export const AUTH_CONFIG: AuthConfig = {
  CLIENT_ID: '[CLIENT_ID]',
  CLIENT_DOMAIN: '[CLIENT_DOMAIN]',
  ...
```

Our app and API are now set up. They can be served by running `ng serve` from the root folder and `node server.js` from the `/server` folder.

With the Node API and Angular app running, let's take a look at how authentication is implemented.

19.0.6 Authentication Service

Authentication logic on the front end is handled with an `AuthService` authentication service: `src/app/auth/auth.service.ts` file.

```
import { Injectable } from '@angular/core';
import { Router } from '@angular/router';
import { BehaviorSubject } from 'rxjs/BehaviorSubject';
import * as auth0 from 'auth0-js';
import { AUTH_CONFIG } from './auth0-variables';
import { UserProfile } from './profile.model';

@Injectable()
export class AuthService {
  // Create Auth0 web auth instance
  // @TODO: Update AUTH_CONFIG and remove .example extension in src/app/auth/auth0-variables
  auth0 = new auth0.WebAuth({
    clientID: AUTH_CONFIG.CLIENT_ID,
    domain: AUTH_CONFIG.CLIENT_DOMAIN,
    responseType: 'token id_token',
    redirectUri: AUTH_CONFIG.REDIRECT,
    audience: AUTH_CONFIG.AUDIENCE,
    scope: AUTH_CONFIG.SCOPE
  });
  userProfile: UserProfile;

  // Create a stream of logged in status to communicate throughout app
  loggedIn: boolean;
  loggedIn$ = new BehaviorSubject<boolean>(this.loggedIn);

  constructor(private router: Router) {
    // If authenticated, set local profile property and update login status subject
    if (this.authenticated) {
```

```
    this.userProfile = JSON.parse(localStorage.getItem('profile'));
    this.setLoggedIn(true);
  }
}

setLoggedIn(value: boolean) {
  // Update login status subject
  this.loggedIn$.next(value);
  this.loggedIn = value;
}

login() {
  // Auth0 authorize request
  this.auth0.authorize();
}

handleAuth() {
  // When Auth0 hash parsed, get profile
  this.auth0.parseHash((err, authResult) => {
    if (authResult && authResult.accessToken && authResult.idToken) {
      window.location.hash = '';
      this._getProfile(authResult);
      this.router.navigate(['/']);
    } else if (err) {
      this.router.navigate(['/']);
      console.error(`Error: ${err.error}`);
    }
  });
}

private _getProfile(authResult) {
  // Use access token to retrieve user's profile and set session
  this.auth0.client.userInfo(authResult.accessToken, (err, profile) => {
    this._setSession(authResult, profile);
  });
}

private _setSession(authResult, profile) {
  // Save session data and update login status subject
  localStorage.setItem('access_token', authResult.accessToken);
  localStorage.setItem('id_token', authResult.idToken);
  localStorage.setItem('profile', JSON.stringify(profile));
  localStorage.setItem('expires_at', authResult.expiresAt);
  this.userProfile = profile;
  this.setLoggedIn(true);
}
```

```
  logout() {
    // Remove tokens and profile and update login status subject
    localStorage.removeItem('access_token');
    localStorage.removeItem('id_token');
    localStorage.removeItem('profile');
    localStorage.removeItem('expires_at');
    this.userProfile = undefined;
    this.setLoggedIn(false);
  }

  get authenticated(): boolean {
    // Check if current time is past access token's expiration
    const expiresAt = JSON.parse(localStorage.getItem('expires_at'));
    return Date.now() < expiresAt;
  }

}
```

This service uses the config variables from `auth0-variables.ts` to instantiate an `auth0.js` WebAuth instance.

An RxJS `BehaviorSubject` is used to provide a stream of authentication status events that you can subscribe to anywhere in the app.

The `login()` method authorizes the authentication request with Auth0 using your config variables. An Auth0 hosted Lock instance will be shown to the user and they can then log in.

> **Note:** If it's the user's first visit to our app *and* our callback is on `localhost`, they'll also be presented with a consent screen where they can grant access to our API. A first party client on a non-localhost domain would be highly trusted, so the consent dialog would not be presented in this case. You can modify this by editing your Auth0 Dashboard API **Settings**. Look for the "Allow Skipping User Consent" toggle.

We'll receive an `id_token`, `access_token`, and `expires_at` in the hash from Auth0 when returning to our app. The `handleAuth()` method uses Auth0's `parseHash()` method callback to get the user's profile (`_getProfile()`) and set the session (`_setSession()`) by saving the tokens, profile, and token expiration to local storage and updating the `loggedIn$` subject so that any subscribed components in the app are informed that the user is now authenticated.

> **Note:** The profile takes the shape of `profile.model.ts` from the OpenID standard claims.

The `handleAuth()` method can then be called in the `app.component.ts` constructor like so:

```
// src/app/app.component.ts
import { AuthService } from './auth/auth.service';
...
  constructor(private auth: AuthService) {
    // Check for authentication and handle if hash present
    auth.handleAuth();
  }
...
```

Finally, we have a `logout()` method that clears data from local storage and updates the `loggedIn$` subject. We also have an `authenticated` accessor to return current authentication status.

Once `AuthService` is provided in `app.module.ts`, its methods and properties can be used anywhere in our app, such as the home component.

The callback component is where the app is redirected after authentication. This component simply shows a loading message until hash parsing is completed and the Angular app redirects back to the home page.

19.0.7 Making Authenticated API Requests

In order to make authenticated HTTP requests, we need to add a `Authorization` header with the access token in our `api.service.ts` file.

```
// src/app/api.service.ts
...
  getDragons$(): Observable<any[]> {
    return this.http
      .get(`${this.baseUrl}dragons`, {
        headers: new HttpHeaders().set(
          'Authorization', `Bearer ${localStorage.getItem('access_token')}`
        )
      })
      .catch(this._handleError);
  }
...
```

19.0.8 Final Touches: Route Guard and Profile Page

A profile page component can show an authenticated user's profile information. However, we only want this component to be accessible if the user is logged in.

With an authenticated API request and login/logout implemented, the final touch is to protect our profile route from unauthorized access. The `auth.guard.ts` route guard can check authentication and activate routes conditionally. The guard is implemented on specific routes of our choosing in the `app-routing.module.ts` file like so:

```
// src/app/app-routing.module.ts
...
import { AuthGuard } from './auth/auth.guard';
...
      {
        path: 'profile',
        component: ProfileComponent,
        canActivate: [
          AuthGuard
        ]
      },
...
```

19.0.9 More Resources

That's it! We have an authenticated Node API and Angular application with login, logout, profile information, and protected routes. To learn more, check out the following resources:

- Why You Should Always Use Access Tokens to Secure an API
- Navigating RS256 and JWKS
- Access Token
- Verify Access Tokens
- Call APIs from Client-side Web Apps
- How to implement the Implicit Grant
- Auth0.js v8 Documentation
- OpenID Standard Claims

Chapter 20

Conclusion

Our ng2-dinos app is complete! Make sure you've run `ng lint` and corrected any issues. With clean code, we shouldn't have any errors. We've successfully migrated our AngularJS 1.x ng1-dinos application to Angular 2+! We've even covered adding authentication with Auth0 so we can authorize routes or make secure API calls in the future.

Thanks for following along. Hopefully you're now ready to dive into Angular migrations as well as new Angular 2 projects with confidence!