Note that the `router-outlet` tag will be replaced with your child component's view.

Router links

After we configure our application routes, we'll be able to navigate through our application either by changing the browser URL or using the `RouterLink` directive to generate anchor tags pointing to a link inside our app. The `RouterLink` directive uses an array of link parameters, which the router will later resolve into a URL matching a component mapping. An example anchor with the `RouterLink` directive will look like this:

```
<a [routerLink]="['/about']">Some</a>
```

Summary

As we've progressed in this chapter, we've learned about TypeScript and Angular 2. We've now covered everything we need in order create an Angular application inside our MEAN application. So let's start by setting up our project.

The project setup

In order to use Angular in our project, we'll need to install both TypeScript and Angular. We'll need to use the TypeScript transpiler to convert our TypeScript files into valid ES5 or ES6 JavaScript files. Furthermore, since Angular is a frontend framework, installing it requires the inclusion of JavaScript files in the main page of your application. This can be done in various ways, and the easiest one would be to download the files you need and store them in the `public` folder. Another approach is to use Angular's CDN and load the files directly from the CDN server. While these two approaches are simple and easy to understand, they both have a strong flaw. Loading a single third-party JavaScript file is readable and direct, but what happens when you start adding more vendor libraries to your project? More importantly, how can you manage your dependencies' versions?

The answer to all of these questions is NPM! NPM will allow us to install all of our dependencies and run the TypeScript transpiler while we develop our application. In order to do that, you'll need to change your `package.json` file, as follows:

```
{
  "name": "MEAN",
  "version": "0.0.7",
  "scripts": {
    "tsc": "tsc",
    "tsc:w": "tsc -w",
    "app": "node server",
      "start": "concurrently \"npm run tsc:w\" \"npm run app\" ",
      "postinstall": "typings install"
    },
    "dependencies": {
      "@angular/common": "2.1.1",
      "@angular/compiler": "2.1.1",
      "@angular/core": "2.1.1",
      "@angular/forms": "2.1.1",
      "@angular/http": "2.1.1",
      "@angular/platform-browser": "2.1.1",
      "@angular/platform-browser-dynamic": "2.1.1",
      "@angular/router": "3.1.1",
      "body-parser": "1.15.2",
      "core-js": "2.4.1",
      "compression": "1.6.0",
      "connect-flash": "0.1.1",
      "ejs": "2.5.2",
      "express": "4.14.0",
      "express-session": "1.14.1",
      "method-override": "2.3.6",
      "mongoose": "4.6.5",
      "morgan": "1.7.0",
      "passport": "0.3.2",
      "passport-facebook": "2.1.1",
      "passport-google-oauth": "1.0.0",
      "passport-local": "1.0.0",
      "passport-twitter": "1.0.4",
      "reflect-metadata": "0.1.8",
      "rxjs": "5.0.0-beta.12",
      "systemjs": "0.19.39",
      "zone.js": "0.6.26"
    },
    "devDependencies": {
      "concurrently": "3.1.0",
      "traceur": "0.0.111",
      "typescript": "2.0.3",
      "typings": "1.4.0"
    }
  }
```

In our new `package.json` file, we did a few things; first, we added our project's Angular dependencies, including a few supportive libraries:

- **CoreJS**: This will provide us with some ES6 polyfills
- **ReflectMetadata**: This will provide us with some a metadata reflection polyfill
- **Rx.JS**: This is a Reactive framework that we'll use later
- **SystemJS**: This will help with loading our application modules
- **Zone.js**: This allows the creation of different execution context zones and is used by the Angular library
- **Concurrently**: This will allow us to run both the TypeScript transplier and our server concurrently
- **Typings**: This will help us with downloading predefined TypeScript definitions for our external libraries

At the top, we added a scripts property, where we defined different scripts we would like npm to run for us. For instance, we have a script that installs our typings for third-party libraries, another one that runs the TypeScript compiler called `tsc`, a script called `app` that we use to run our node server, and one called `start` to run both of these scripts together using the concurrency tool.

Next, we're going to configure the way we want the TypeScript compiler to run.

Configuring TypeScript

In order to configure the way TypeScript works, we'll need to add a new file called `tsconfig.json` to our application's root folder. In your new file, paste the following JSON:

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "system",
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "removeComments": false,
    "noImplicitAny": false
  },
  "exclude": [
    "node_modules",
        "typings/main",
        "typings/main.d.ts"
      ]
    }
```

In our `tsconfig.json` file, we configured the TypeScript compiler to:

- Compile our TypeScript code into ES5 code
- Compile our modules into a system module pattern
- Use Node for module resolution
- Generate source maps
- Include decorators and emit their metadata
- Keep comments
- Cancel the error for any implicit declarations
- Not include the `node_modules` folder and typings files

When we run our application, the TypeScript will use the `tsconfig.json` configuration file by default. Next, you'll need to add a new file called `typings.json` to your application's root folder. In your new file, paste the following JSON:

```
    {
      "globalDependencies": {
      "core-js": "registry:dt/core-js#0.0.0+20160914114559",
        "jasmine": "registry:dt/jasmine#2.5.0+20161025102649",
        "socket.io-client":
          "registry:dt/socket.io-client#1.4.4+20160317120654",
        "node": "registry:dt/node#6.0.0+20161102143327"
      }
    }
```

As you can see, we've added all third-party libraries we need in order for the TypeScript transpiler to compile our code properly. Once you're done, go ahead and install your new dependencies:

```
$ npm install
```

All the packages we need will be installed along with external type definitions we'll need in order to support the TypeScript compiling. Now that we have installed our new packages and configured our TypeScript implementation, it is time to set up Angular.

[It is recommended that you continue reading about Typings at the official documentation at https://github.com/typings/typings.]

Configuring Express

To start using Angular, you will need to include the new JavaScript library files in our main EJS view. So, we will use the app/views/index.ejs file as the main application page. However, NPM installed all of our dependencies in the node_module folder, which is not accessible to our client side. To solve this issue, we'll have to change our config/express.js file as follows:

```
const path = require('path'),
const config = require('./config'),
const express = require('express'),
const morgan = require('morgan'),
const compress = require('compression'),
const bodyParser = require('body-parser'),
const methodOverride = require('method-override'),
const session = require('express-session'),
const flash = require('connect-flash'),
const passport = require('passport');

module.exports = function() {
  const app = express();

  if (process.env.NODE_ENV === 'development') {
    app.use(morgan('dev'));
  } else if (process.env.NODE_ENV === 'production') {
    app.use(compress());
  }

  app.use(bodyParser.urlencoded({
    extended: true
  }));
  app.use(bodyParser.json());
  app.use(methodOverride());

  app.use(session({
    saveUninitialized: true,
    resave: true,
    secret: config.sessionSecret
  }));
```

```
    app.set('views', './app/views');
    app.set('view engine', 'ejs');

    app.use(flash());
    app.use(passport.initialize());
    app.use(passport.session());

    app.use('/', express.static(path.resolve('./public')));
    app.use('/lib', express.static(
      path.resolve('./node_modules')));

    require('../app/routes/users.server.routes.js')(app);
    require('../app/routes/index.server.routes.js')(app);

    return app;
  };
```

A major change here involves the creation of a `/lib` static route that directs to our `node_modules` folder. While we were here, we also switched the order of the routes users and index routes. This will come in handy when we start dealing with Angular's routing mechanism. In this regard, there is one more thing we have to do, and that is making sure our Express application always return the main application view when receiving routes that are not defined. This is for the case where the browser's initial request is made using a URL that was generated by the Angular Router and is not supported by our Express configuration. To do this, go back to the `app/routes/index.server.routes.js` file, and change it as follows:

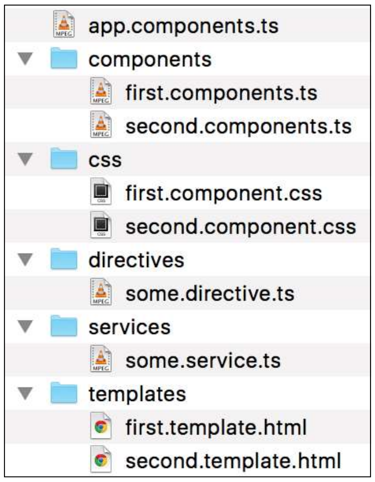
```
module.exports = function(app) {
  const index = require('../controllers/index.server.controller');

  app.get('/*', index.render);
};
```

Now, that we have configured TypeScript and Express, it is time to set up Angular, but before we do that, let's talk a bit about our application structure.

Restructuring the application

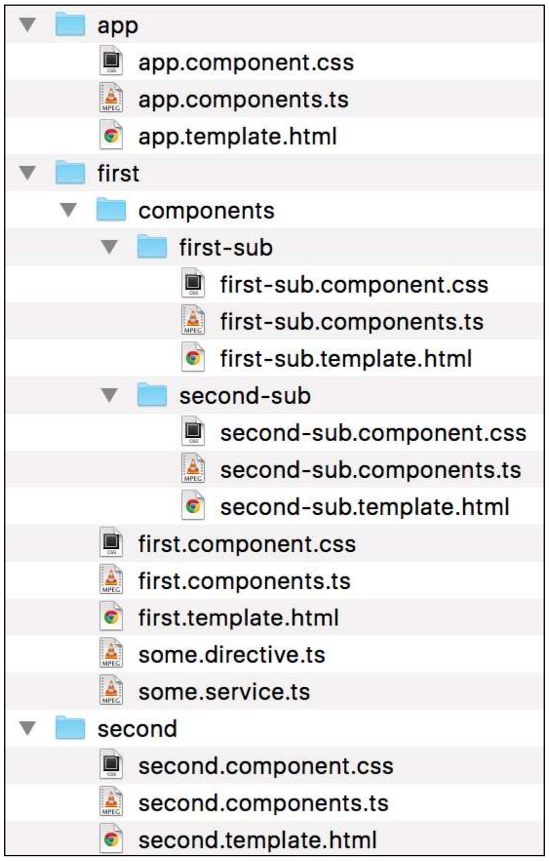
As you might remember from *Chapter 3*, *Building an Express Web Application*, your application's structure depends on the complexity of your application. We previously decided to use the horizontal approach for the entire MEAN application; however, as we stated earlier, MEAN applications can be constructed in various ways, and an Angular application structure is a different topic, which is often discussed by the community and the Angular development team. There are many doctrines for different purposes, some of which are a bit more complicated, while others offer a simpler approach. In this section, we'll introduce a recommended structure. With the move from Angular 1 to Angular 2, this discussion is now even more complicated. For us, the easiest approach would be to start by using the `public` folder of our Express application as the root folder for the Angular application so that every file is available statically.

There are several options to structure your application according to its complexity. A simple application can have a horizontal structure where entities are arranged in folders according to their type, and a main application file is placed at the root folder of the application. An example application structure of this kind can be seen in the following screenshot:



```
app.components.ts
▼ components
    first.components.ts
    second.components.ts
▼ css
    first.component.css
    second.component.css
▼ directives
    some.directive.ts
▼ services
    some.service.ts
▼ templates
    first.template.html
    second.template.html
```

As you can see, this is a very comfortable solution for small applications with a few entities. However, your application might be more complex with several different features and many more entities. This structure cannot handle an application of this sort since it obfuscates the behavior of each application file, will have a bloated folder with too many files, and will generally be very difficult to maintain. For this purpose, there is a different approach to organizing your files in a vertical manner. A vertical structure positions every file according to its functional context, so different types of entities can be sorted together according to their role in a feature or a section. This is similar to the vertical approach we introduced in *Chapter 3*, *Building an Express Web Application*. However, the difference is that only Angular's logical units will have a standalone module folder structure, usually with a component and a template files. An example of an Angular application vertical structure can be seen in the following screenshot:



```
app
    app.component.css
    app.components.ts
    app.template.html
first
    components
        first-sub
            first-sub.component.css
            first-sub.components.ts
            first-sub.template.html
        second-sub
            second-sub.component.css
            second-sub.components.ts
            second-sub.template.html
    first.component.css
    first.components.ts
    first.template.html
    some.directive.ts
    some.service.ts
second
    second.component.css
    second.components.ts
    second.template.html
```

As you can see, each module has its own folder structure, which allows you to encapsulate each component. We're also using the file naming convention that we introduced in *Chapter 3*, *Building an Express Web Application*.

Now that you know the basic best practices of naming and structuring your application, let's continue and create the application module.

Creating the application module

To begin, clear the contents of the `public` folder and create the folder named `app` inside it. Inside your new folder, create a file named `app.module.ts`. In your file, add the following code:

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

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

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

As you can see, we basically just created a simple module that declares the application component and uses it for bootstrapping. Next we'll need to create the application component.

Creating the application component

Inside your `public/app` folder, create a new file named `app.component.ts`. In your file, add the following code:

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

@Component({
  selector: 'mean-app',
  template: '<h1>Hello World</h1>',
})
export class AppComponent {}
```

As you can see, we basically just created the simplest component. Next we'll learn how to bootstrap our `AppModule` class.

Bootstrapping the application module

To bootstrap your application module, go to your `app` folder and create a new file named `bootstrap.ts`. In your file, add the following code:

```
import { platformBrowserDynamic } from
  '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

Basically, this code is using the browser platform module to bootstrap the application module for browsers. Once we have these configured, it's time to learn how to load our bootstrap code using the SystemJS module loader.

Starting your Angular application

To use SystemJS as our module loader, we'll create a new file named `systemjs.config.js` inside our `public` folder. In your new file, paste the following code:

```
(function(global) {
  var packages = {
    app: {
      main: './bootstrap.js',
      defaultExtension: 'js'
    }
  };

  var map = {
    '@angular': 'lib/@angular',
    'rxjs': 'lib/rxjs'
  };

  var ngPackageNames = [
    'common',
    'compiler',
    'core',
    'forms',
    'http',
    'router',
      'platform-browser',
      'platform-browser-dynamic',
    ];

    ngPackageNames.forEach(function(pkgName) {
      packages['@angular/' + pkgName] = {
        main: '/bundles/' + pkgName + '.umd.js',
        defaultExtension: 'js' };
    });

    System.config({
      defaultJSExtensions: true,
      transpiler: null,
      packages: packages,
      map: map
    });
  })(this);
```

In this file, we're telling SystemJS about our application package and from where to load the Angular and Rx modules. We then describe the main file for each package of Angular; in this case, we ask it to load the UMD file of each package. We then use the `System.config` method to configure SystemJS. Finally, we revisit our `app/views/index.ejs` file and change it, as follows:

```
<!DOCTYPE html>
<html>
<head>
  <title><%= title %></title>
  <base href="/">
</head>
<body>
  <mean-app>
    <h1>Loading...</h1>
  </mean-app>

  <script src="lib/core-js/client/shim.min.js"></script>
  <script src="lib/zone.js/dist/zone.js"></script>
  <script src="lib/reflect-metadata/Reflect.js"></script>
  <script src="lib/systemjs/dist/system.js"></script>

  <script src="systemjs.config.js"></script>
  <script>
    System.import('app').catch(function(err){ console.error(err); });
  </script>
</body>
</html>
```

As you can see, we're loading our module files directly from the `node_modules` package folder and include our SystemJS configuration file. The last script tells SystemJS to load the application package we defined in the configuration file.

> To learn more about SystemJS, it is recommended that you visit the official documentation at `https://github.com/systemjs/systemjs`.

Now all you have left to do is run your application by invoking the following command in your command line:

```
$ npm start
```

When your application is running, use your browser and open your application URL at `http://localhost:3000`. You should see a header tag saying `Hello World` being rendered. Congratulations! You've created your first Angular 2 module and component and successfully bootstrapped your application. Next, we'll refactor the authentication part of our application and create a new authentication module.

Managing authentication

Managing an Angular application authentication is a complex issue. The problem is that while the server holds the information about the authenticated user, the Angular application is not aware of that information. One solution is to use a service and ask the server about the authentication status; however, this solution is flawed since all the Angular components will have to wait for the response to return, causing inconsistencies and development overhead. This can be solved using an advanced Angular router object; however, a simpler solution would be to make the Express application render the `user` object directly in the EJS view and then use an Angular service to serve the object.

Rendering the user object

To render the authenticated `user` object, you'll have to make several changes. Let's begin by changing the `app/controllers/index.server.controller.js` file, as follows:

```
exports.render = function(req, res) {
  const user = (!req.user) ? null : {
    _id: req.user.id,
    firstName: req.user.firstName,
    lastName: req.user.lastName
  };
```

```
    res.render('index', {
      title: 'Hello World',
      user: JSON.stringify(user)
    });
  };
```

Next, go to your `app/views/index.ejs` file and make the following changes:

```
  <!DOCTYPE html>
  <html>
  <head>
    <title><%= title %></title>
    <base href="/">
  </head>
  <body>
    <mean-app>
      <h1>Loading...</h1>
    </mean-app>

    <script type="text/javascript">
      window.user = <%- user || 'null' %>;
    </script>

    <script src="lib/core-js/client/shim.min.js"></script>
    <script src="lib/zone.js/dist/zone.js"></script>
    <script src="lib/reflect-metadata/Reflect.js"></script>
    <script src="lib/systemjs/dist/system.js"></script>

    <script src="systemjs.config.js"></script>

    <script>
      System.import('app').catch(function(err){
       console.error(err); });
    </script>
  </body>
  </html>
```

This will render the user object as a JSON representation right in your main view application. When the Angular application bootstraps, the authentication state will already be available. If the user is authenticated, the `user` object will become available; otherwise, the `user` object will be Null.

Modifying the users' server controller

To support our authentication refactoring, we'll need to make sure our user's server controller is able to process the Angular service requests. To do that, you'll need to change the code in your `app/controllers/users.server.controller.js` file to look like this:

```
const User = require('mongoose').model('User'),
  passport = require('passport');

const getErrorMessage = function(err) {
  const message = '';

  if (err.code) {
    switch (err.code) {
      case 11000:
      case 11001:
      message = 'Username already exists';
      break;
      default:
      message = 'Something went wrong';
    }
  } else {
    for (let errName in err.errors) {
      if (err.errors[errName].message) message =
        err.errors[errName].message;
    }
  }

  return message;
};

exports.signin = function(req, res, next) {
  passport.authenticate('local', function(err, user, info) {
    if (err || !user) {
      res.status(400).send(info);
    } else {
      // Remove sensitive data before login
      user.password = undefined;
      user.salt = undefined;

      req.login(user, function(err) {
        if (err) {
          res.status(400).send(err);
        } else {
          res.json(user);
        }
      });
    }
  })(req, res, next);
};

exports.signup = function(req, res) {
  const user = new User(req.body);
  user.provider = 'local';

  user.save((err) => {
    if (err) {
      return res.status(400).send({
        message: getErrorMessage(err)
      });
    } else {
      // Remove sensitive data before login
      user.password = undefined;
      user.salt = undefined;

      req.login(user, function(err) {
        if (err) {
          res.status(400).send(err);
        } else {
          res.json(user);
        }
      });
    }
  });
};

exports.signout = function(req, res) {
  req.logout();
  res.redirect('/');
};

exports.saveOAuthUserProfile = function(req, profile, done) {
  User.findOne({
    provider: profile.provider,
    providerId: profile.providerId
  }, function(err, user) {
    if (err) {
      return done(err);
      } else {
        if (!user) {
          const possibleUsername = profile.username ||
          ((profile.email) ? profile.email.split('@')[0] : '');

          User.findUniqueUsername(possibleUsername, null,
          function(availableUsername) {
            profile.username = availableUsername;

            user = new User(profile);

            user.save((err) => {
              if (err) {
                const message =
                  _this.getErrorMessage(err);

                req.flash('error', message);
                return res.redirect('/signup');
              }

              return done(err, user);
            });
          });
        } else {
          return done(err, user);
        }
      }
    });
  };
```

We basically just encapsulated the authentication logic inside two methods that can accept and respond with a JSON object. Now let's go ahead and change the `app/routes/users.server.routes.js` directory as follows:

```
const users =
  require('../../app/controllers/users.server.controller'),
  passport = require('passport');

module.exports = function(app) {
  app.route('/api/auth/signup').post(users.signup);
  app.route('/api/auth/signin').post(users.signin);
  app.route('/api/auth/signout').get(users.signout);

  app.get('/api/oauth/facebook',
    passport.authenticate('facebook', {
      failureRedirect: '/signin'
    }));
    app.get('/api/oauth/facebook/callback',
      passport.authenticate('facebook', {
      failureRedirect: '/signin',
      successRedirect: '/'
    }));

    app.get('/api/oauth/twitter', passport.authenticate('twitter',
    {
       failureRedirect: '/signin'
    }));
    app.get('/api/oauth/twitter/callback',
      passport.authenticate('twitter', {
      failureRedirect: '/signin',
      successRedirect: '/'
    }));

    app.get('/api/oauth/google', passport.authenticate('google', {
      failureRedirect: '/signin',
      scope: [
        'https://www.googleapis.com/auth/userinfo.profile',
        'https://www.googleapis.com/auth/userinfo.email'
      ],
    }));
    app.get('/api/oauth/google/callback',
      passport.authenticate('google', {
      failureRedirect: '/signin',
      successRedirect: '/'
    }));

  };
```

Note how we removed the routes that we used to render our authentication views. More importantly, look at the way in which we added an `/api` prefix for all the routes. It is a very good practice to keep all your routes under one prefix, since we want the Angular router to be able to have routes that do not interfere with our server routes. Now that we have our server side ready, it's time to create our Angular authentication module.

Creating the authentication module

Now that we're done with laying the ground for our Angular application, we can move forward and refactor our authentication logic into a cohesive authentication module. To do that, we'll begin by creating a new folder inside our `public/app` folder, called `authentication`. In our new folder, create a file named `authentication.module.ts` with the following code:

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

import { AuthenticationRoutes } from './authentication.routes';
import { AuthenticationComponent } from './authentication.component';
import { SigninComponent } from './signin/signin.component';
import { SignupComponent } from './signup/signup.component';

@NgModule({
  imports: [
    FormsModule,
    RouterModule.forChild(AuthenticationRoutes),
  ],
  declarations: [
    AuthenticationComponent,
    SigninComponent,
    SignupComponent,
  ]
})
export class AuthenticationModule {}
```

Our module consists of three components:

- An authentication component
- A signup component
- A signin component

We also included an authentication routing configuration and the Angular's Forms module to support our signin and signup forms. Let's begin by implementing the base authentication component.

Creating the authentication component

We'll begin by creating our authentication component hierarchy. Then, we will convert our server signin and signup views into Angular templates, add the authentication functionality to `AuthenticationService`, and refactor our server logic. Let's start by creating a file named `authentication.component.ts` inside our `public/app/authentication` folder. In the new file, paste the following code:

```
import { Component } from '@angular/core';
import { SigninComponent } from './signin/signin.component';
import { SignupComponent } from './signup/signup.component';

@Component({
  selector: 'authentication',
  templateUrl: 'app/authentication/authentication.template.html',
})
export class AuthenticationComponent { }
```

In this code, we implement our new authentication component. We begin by importing the authentication service and a signup and signin component, all of which we haven't created yet. Another thing to notice is that this time, we used an external template file for our component. We'll continue by creating a routing configuration for our authentication module.

Configuring the authentication routes

To do that, create a new file named `authentication.routes.ts` inside our `public/app/authentication` folder. In the new file, paste the following code:

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

import { AuthenticationComponent } from './authentication.component';
import { SigninComponent } from './signin/signin.component';
import { SignupComponent } from './signup/signup.component';

export const AuthenticationRoutes: Routes = [{
  path: 'authentication',
  component: AuthenticationComponent,
  children: [
    { path: 'signin', component: SigninComponent },
    { path: 'signup', component: SignupComponent },
  ],
}];
```

As you can see, we create a new `Routes` instance with a parent route of `authentication` and two child routes for the `signin` and `signup` components. We'll continue by creating the template file named `authentication.template.html` inside our component's folder. In the new file, paste the following code:

```
<div>
  <a href="/api/oauth/google">Sign in with Google</a>
  <a href="/api/oauth/facebook">Sign in with Facebook</a>
  <a href="/api/oauth/twitter">Sign in with Twitter</a>
  <router-outlet></router-outlet>
</div>
```

Note how we used the `RouterOutlet` directive inside our code. This is where our subcomponents will be rendered. We'll continue with creating these subcomponents.

Creating the signin component

To implement the `signin` component, create a new folder named `signin` inside your `public/app/authentication` folder. Inside your new folder, create a new file named `signin.component.ts` with the following code:

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

import { AuthenticationService } from '../authentication.service';

@Component({
  selector: 'signin',
  templateUrl: 'app/authentication/signin/signin.template.html'
})
export class SigninComponent {
  errorMessage: string;
  credentials: any = {};

  constructor (private _authenticationService: AuthenticationService,
private _router: Router) {    }

  signin() {
    this._authenticationService.signin(
      this.credentials).subscribe(result  =>
      this._router.navigate(['/']),
      error =>  this.errorMessage = error );
  }
}
```

Note how our `signin` component uses the authentication service in order to perform a `signin` action. Don't worry; we'll implement this in the next section. Next, you'll need to create a file named `signin.template.html` in the same folder as your component. In your new file, add the following code:

```
<form (ngSubmit)="signin()">
  <div>
    <label>Username:</label>
    <input type="text" [(ngModel)]="credentials.username"
      name="username">
  </div>
  <div>
    <label>Password:</label>
    <input type="password" [(ngModel)]="credentials.password"
      name="password">
  </div>
  <div>
    <input type="submit" value="Sign In">
  </div>
  <span>{{errorMessage}}</span>
</form>
```

We've just created a new component to handle our authentication signin operation! The signup component will look quite similar.

Creating the signup component

To implement the signup component, create a new folder named `signup` inside your `public/app/authentication` folder. Inside your new folder, create a new file named `signup.component.ts` with the following code:

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

import { AuthenticationService } from '../authentication.service';

@Component({
  selector: 'signup',
  templateUrl: 'app/authentication/signup/signup.template.html'
})
export class SignupComponent {
  errorMessage: string;
  user: any = {};

  constructor (private _authenticationService:
        AuthenticationService,
        private _router: Router) {}

      signup() {
        this._authenticationService.signup(this.user)
        .subscribe(result  => this._router.navigate(['/']),
        error =>  this.errorMessage = error);
      }
    }
```

Note how our signup component uses the authentication service in order to perform a `signup` action. Next, you'll need to create a file named `signup.template.html` in the same folder as your component. In your new file, add the following code:

```
<form (ngSubmit)="signup()">
  <div>
  <label>First Name:</label>
    <input type="text" [(ngModel)]="user.firstName"
      name="firstName">
  </div>
  <div>
    <label>Last Name:</label>
    <input type="text" [(ngModel)]="user.lastName"
      name="lastName">
  </div>
  <div>
    <label>Email:</label>
    <input type="text" [(ngModel)]="user.email" name="email">
  </div>
  <div>
    <label>Username:</label>
    <input type="text" [(ngModel)]="user.username"
      name="username">
  </div>
  <div>
    <label>Password:</label>
    <input type="password" [(ngModel)]="user.password"
      name="password">
  </div>
  <div>
    <input type="submit" value="Sign up" />
  </div>
  <span>{{errorMessage}}</span>
</form>
```

Now that we have our authentication components in place, let's go back and handle the authentication service.

Creating the authentication service

In order to support our new components, we would need to create an authentication service to provide them with the needed functionality. To do that, create a new file named `authentication.service.ts` inside your `public/app/authentication` folder. In your new file, paste the following code:

```
import 'rxjs/Rx';
import { Injectable } from '@angular/core';
import { Http, Response, Headers, RequestOptions } from
  '@angular/http';
import { Observable } from 'rxjs/Observable';

@Injectable()
export class AuthenticationService {
  public user = window['user'];

  private _signinURL = 'api/auth/signin';
  private _signupURL = 'api/auth/signup';

  constructor (private http: Http) {

  }
  isLoggedIn(): boolean {
    return (!!this.user);
  }

  signin(credentials: any): Observable<any> {
    let body = JSON.stringify(credentials);
    let headers = new Headers({ 'Content-Type':
      'application/json' });
    let options = new RequestOptions({ headers: headers });

    return this.http.post(this._signinURL, body, options)
    .map(res => this.user = res.json())
    .catch(this.handleError)
  }

  signup(user: any): Observable<any> {
    let body = JSON.stringify(user);
    let headers = new Headers({ 'Content-Type':
      'application/json' });
      let options = new RequestOptions({ headers: headers });

      return this.http.post(this._signupURL, body, options)
      .map(res => this.user = res.json())
      .catch(this.handleError)
    }

    private handleError(error: Response) {
      console.error(error);
      return Observable.throw(error.json().message ||
        'Server error');
    }
  }
```

Note how we decorated the `AuthenticationService` class with an `@Injectable` decorator. While that's not needed in this case, it is a good practice to decorate your services that way. The reason is that if you'd like to inject a service with another service, you'll need to use this decorator, so for the sake of uniformity, it is better to stay safe and decorate all your services. Another thing to note is the way we get our user object from the window object.

We also added three methods to our service: one that handles signin, another that handles signup, and a last one for error handling. Inside our methods, we use the HTTP module provided by Angular to call our server endpoints. In the next chapter, we'll elaborate further on this module, but in the meantime, all you need to know is that we just used it to send POST a request to our server. To finish up the Angular part, our application will need to modify our application module and add a simple home component.

Creating the home module

To extend our simple example, we'll need to have a home component that will provide the view for our base root and will present different information for the logged-in and logged-out users. To do that, create a folder named `home` inside your `public/app` folder. Then, create a file inside this folder called `home.module.ts`, which contains the following code:

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

import { HomeRoutes } from './home.routes';
import { HomeComponent } from './home.component';
```

```
@NgModule({
  imports: [
    CommonModule,
    RouterModule.forChild(HomeRoutes),
  ],
  declarations: [
    HomeComponent,
  ]
})
export class HomeModule {}
```

As you may have probably noticed, our module is only importing a new home component and the routing configuration. Let's continue by creating our home component.

Creating the home component

Next, we'll create our home component. To do that, go to your `public/app/home` folder and create a new file called `home.component.ts` containing the following code:

```
import { Component } from '@angular/core';
import { AuthenticationService } from '../authentication/
authentication.service';

@Component({
  selector: 'home',
  templateUrl: './app/home/home.template.html'
})
export class HomeComponent {
  user: any;

  constructor (private _authenticationService:
    AuthenticationService) {
    this.user = _authenticationService.user;
  }
}
```

As you can see, this is just a simple component, which has the authentication service injected and which is used to provide the component with the user object. Next, we'll need to create our home component template. To do that, go to your `public/app/home` folder and create a file named `home.template.html` with the following code inside it:

```
<div *ngIf="user">
  <h1>Hello {{user.firstName}}</h1>
  <a href="/api/auth/signout">Signout</a>
</div>

<div *ngIf="!user">
  <a [routerLink]="['/authentication/signup']">Signup</a>
  <a [routerLink]="['/authentication/signin']">Signin</a>
</div>
```

This template's code nicely demonstrates a few of the topics we previously discussed. Note the use of the `ngIf` and `routerLink` directives we talked about earlier in this chapter.

Configuring the home routes

To finish with our module, we'll need to create a routing configuration for our home component. To do that, create a new file named `home.routes.ts` inside your `public/app/home` folder. In your new file, paste the following code:

```
import { Routes } from '@angular/router';
import { HomeComponent } from './home.component';

export const HomeRoutes: Routes = [{
  path: '',
  component: HomeComponent,
}];
```

As you can see, this is just a simple component routing. To complete our implementation, we'll need to modify our application module a bit.

Refactoring the application module

To include our authentication and home component modules, we'll need to change our `app.module.ts` file as follows:

```
import { NgModule }       from '@angular/core';
import { BrowserModule }  from '@angular/platform-browser';
import { RouterModule }   from '@angular/router';
import { HttpModule } from '@angular/http';

import { AppComponent }       from './app.component';
import { AppRoutes }       from './app.routes';

import { HomeModule } from './home/home.module';
import { AuthenticationService } from './authentication/
authentication.service';
import { AuthenticationModule } from './authentication/authentication.
module';

@NgModule({
  imports: [
    BrowserModule,
    HttpModule,
    AuthenticationModule,
    HomeModule,
    RouterModule.forRoot(AppRoutes),
  ],
  declarations: [
    AppComponent
  ],
  providers: [
    AuthenticationService
  ],
  bootstrap: [AppComponent]
})
   export class AppModule { }
```

As you can see, this is quite a big change to our application module. First, we imported the HTTP module and our new home and authentication modules along with our new Application routing configuration. We injected the authentication service in the `providers` property so that it is available for all of our submodules. The last thing we have to do is implement our application routing configuration.

Configuring the application routes

To configure our application routes, we'll need to create a new file named `app.routes.ts` inside the `public/app` folder. In the new file, paste the following code:

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

export const AppRoutes: Routes = [{
  path: '**',
  redirectTo: '/',
}];
```

As you can see, our application consists of a very simple, single configuration, which redirects any unknown routing requests to our home component.

That is it. Your application is ready for use! All you need to do is to run it by invoking the following command in your command line:

```
$ npm start
```

When your application is running, use your browser and open your application URL at `http://localhost:3000`. You should see two links for signing up and signing in. Use them and see what happens. Try to refresh your application and see how it keeps its state and route.

Summary

In this chapter, you learned about the basic principles of TypeScript. You went through Angular's building blocks and learned how they fit in the architecture of an Angular 2 application. You also learned how to use NPM to install frontend libraries and how to structure and bootstrap your application. You discovered Angular's entities and how they work together. You also used Angular's Router to configure your application routing scheme. Near the end of this chapter, we made use of all of this in order to refactor our authentication module. In the next chapter, you'll connect everything you learned so far to create your first MEAN CRUD module.