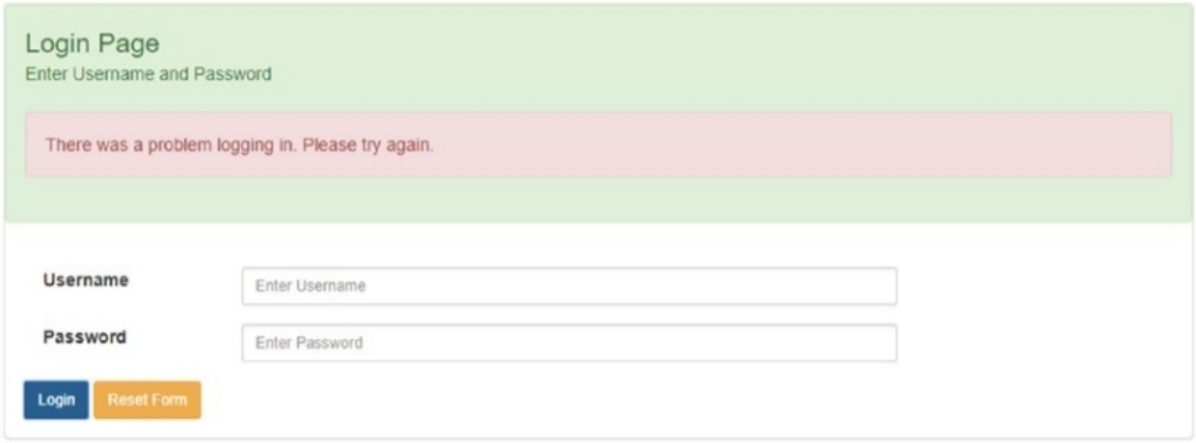

Figure 5-6. *Login page with error message*

As per the routes defined in an application, such as `app.js`, you have defined a login form that needs to be linked with `loginController`, so let's update your `controller.js` file.

Updating controller.js for the Login and Logout Authentication Process

To support the login and logout authentication process, you need to add some more features. You need to add three new controllers: `homeController`, `loginController`, and `logoutController`. You add these to the existing UserRegistrationSystem application in the `src/main/resources/static/js/controller.js` file, as shown in Listing 5-8.

Listing 5-8. src/main/resources/static/js/controller.js

```
app.controller('homeController', function($rootScope, $scope,
                $http, $location, $route){

         if ($rootScope.authenticated) {
                $location.path("/");
                $scope.loginerror = false;
       } else {
                $location.path("/login");
                $scope.loginerror = true;
       }
});

app.controller('loginController', function($rootScope, $scope,
                $http, $location, $route){
        $scope.credentials = {};

        $scope.resetForm = function() {
                $scope.credentials = null;
        }
```

```
        var authenticate = function(credentials, callback) {
                var headers = $scope.credentials ? {
                        authorization : "Basic "
                                        + btoa($scope.credentials.username + ":"
                                                        + $scope.credentials.password)
                        } : {};

                $http.get('user', {
                        headers : headers
                }).then(function(response) {
                        if (response.data.name) {
                                $rootScope.authenticated = true;
                        } else {
                                $rootScope.authenticated = false;
                        }
                        callback && callback();
                }, function() {
                        $rootScope.authenticated = false;
                        callback && callback();
                });
        }

        authenticate();

        $scope.loginUser = function() {
              authenticate($scope.credentials, function() {
                  if ($rootScope.authenticated) {
                    $location.path("/");
                    $scope.loginerror = false;
                  } else {
                    $location.path("/login");
                    $scope.loginerror = true;
                  }
              });
        };
});

app.controller('logoutController', function($rootScope, $scope,
                $http, $location, $route){
        $http.post('logout', {}).finally(function() {
            $rootScope.authenticated = false;
            $location.path("/");
        });
});
```

As shown in Listing 5-8, you created new controllers named `homeController`, `loginController`, and `logoutController`. The `homeController` controller checks whether `authenticated` inside `$rootScope` is true and then sets `$location.path` with `/` and `loginerror` in `$scopeto` to false, or it sets `$location.path` with `/login` and `loginerror` in `$scope` to `true`.

The `loginController` controller will be executed when the login page loads. This controller starts with initializing the credentials object, and then it defines the functions: the `resetForm()` method that resets the

input box for the username and password with a null value, the `authenticate()` function (a local helper function) that loads a `user` resource from the back end, and the function `loginUser()` that you need in the form.

The local helper function `authenticate()` is called when the controller is loaded to see whether the user is actually already authenticated, and you need this function just to make a remote call because the actual authentication is done by the server. This function also sets an application-wide flag called `authenticated` that you used in the `index.html` page to show/hide elements and control which parts of the page are rendered. You achieved this application-wide flag using `$rootScope` because it's convenient and easy to follow, and you need to share the authenticated flag between different controllers.

This `authenticate()` method makes a `GET` call to a relative resource called `/user`. While calling from the `loginUser()` function, the `authenticate()` function adds the Base64-encoded credentials in the request headers, so on the server it does an authentication and accepts a cookie in return. The `loginUser()` function also sets a local `$scope.loginerror` flag accordingly when it gets the result of the authentication that is being used to control the display of the error message in the login page.

If the user is authenticated, then you show a Logout button on each web page. Clicking the Logout button will let `logoutController` to be executed. The `logoutController` controller sends an HTTP `POST` to `/logout`, which you do not need to implement on the server because it is added for you already by Spring Security. To add more control over the default behavior of the logout process provided by Spring Security, you could use the `HttpSecurity` callback's `inSpringSecurityConfiguration` in `Memory.java` to, for instance, execute some business logic after logout.

Updating the Back-End Code

You also need to update your back-end code to support the login and logout authentication process in your UserRegistrationSystem application.

Creating a New RESTful Endpoint to Get the Currently Authenticated User

As you saw in Listing 5-8, the `authenticate()` function makes a `GET` request to the resource `/user` to get the currently authenticated user. So, to service the `authenticate()` function, you have to add a new REST endpoint in your UserRegistrationSystem application. Let's create the `ServiceAuthenticate.java` class inside the package `com.apress.ravi.Rest` under the `src/main/java` folder, as shown in Listing 5-9.

Listing 5-9. com.apress.ravi.Rest.ServiceAuthenticate.java

```
package com.apress.ravi.Rest;

import java.security.Principal;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class ServiceAuthenticate {

        @RequestMapping("/user")
        public Principal user(Principal user) {
                return user;
        }
}
```

If the `/user` resource is reachable, then it will return the authenticated user.

Updating the Spring Security Configuration to Handle Login Requests

You also need to update your existing Spring Security configuration file com.apress.ravi.Security.SpringSecurityConfiguration_InMemory.java, as shown in Listing 5-10.

Listing 5-10. SpringSecurityConfiguration_InMemory.java

```
package com.apress.ravi.Security;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.autoconfigure.security.SecurityProperties;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.annotation.Order;
import org.springframework.security.config.annotation.authentication.builders.
AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.
WebSecurityConfigurerAdapter;
import org.springframework.security.web.csrf.CookieCsrfTokenRepository;

@Configuration
@Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)
public class SpringSecurityConfiguration_InMemory
                extends WebSecurityConfigurerAdapter {

        @Autowired
        protected void configureGlobal(AuthenticationManagerBuilder auth)
                        throws Exception {
                auth.inMemoryAuthentication().
                        withUser("user").password("password")
                                .roles("USER");
                auth.inMemoryAuthentication()
                        .withUser("admin").password("password")
                                .roles("USER", "ADMIN");
        }

        @Override
        protected void configure(HttpSecurity http) throws Exception {

                http
                        .httpBasic()
                                .realmName("User Registration System")
                        .and()
                        .authorizeRequests()
                                .antMatchers("/login/login.html", "/template/home.html",
                                "/").permitAll()
                                .anyRequest().authenticated()
                                .and()
                                .csrf()
.csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse());
        }
}
```

In Listing 5-10, you updated your existing `configure` method to handle the login request. You allowed anonymous access to the static resources such as `/login/login.html`, `/template/home.html`, and `/`, because these HTML resources need to be available to anonymous users.

Spring Security provides a special `CsrfTokenRepository` to send a cookie. When you start with a clean browser (Ctrl+F5 or incognito in Chrome), the first request to the server has no cookies, but the server sends back `Set-Cookie` for `JSESSIONID` (the regular `HttpSession`) and `X-XSRF-TOKEN`, which are the CRSF cookies that you set up in Listing 5-10. Subsequent requests will have these cookies, which are important: the Spring Security application doesn't work without them, and they are providing some really basic security features (authentication and CSRF protection). When you log out, the values of the cookies change. Spring Security expects the token sent to it in a header called `X-CSRF`. From the initial request that loads the home page, the value of the CSRF token was available in the `HttpRequest` attributes on the server side. Angular has built-in support for CSRF (which it calls XSRF) based on cookies. Angular wants the cookie name to be `XSRF-TOKEN`, and the best part of Spring Security is that it provides it as a request attribute by default.

Running the Application

Let's restart your UserRegistrationSystem application to test these features. Open a browser and visit `http://localhost:8080/`, and your application will redirect to the link `http://localhost:8080/#/login` to provide a login page, as shown in Figure 5-7.

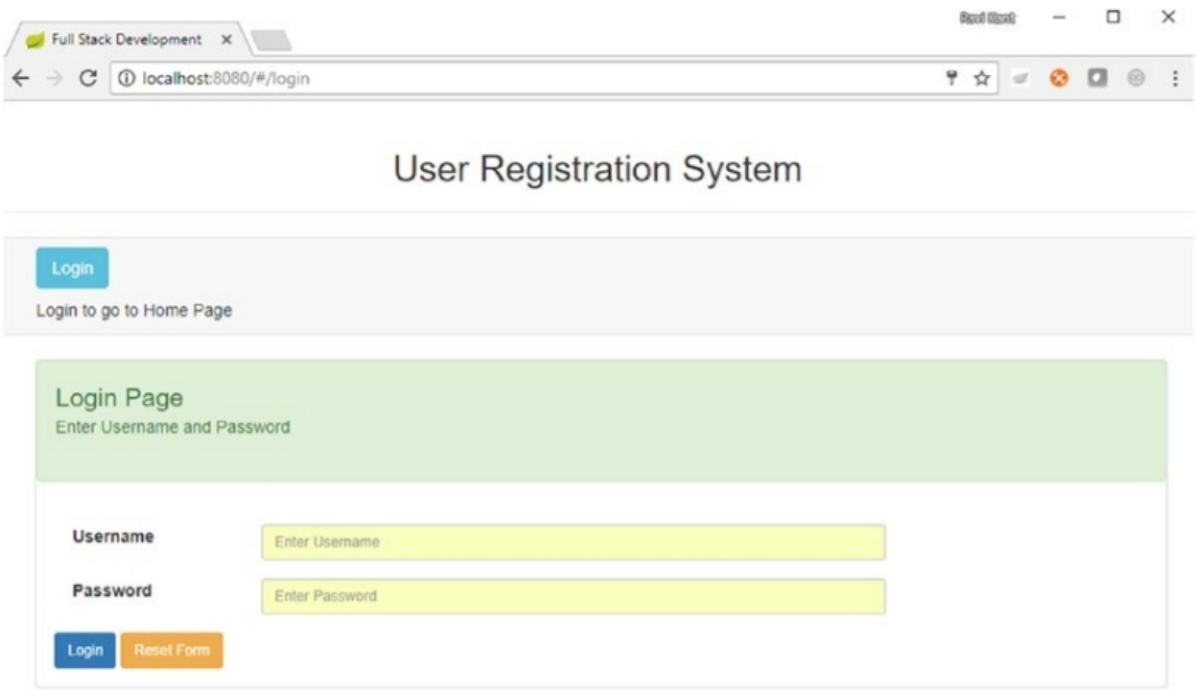

Figure 5-7. *Navigating to the login page*

On successful login, after entering the username and password and clicking the Login button, you will be redirected to the home page, as shown in Figure 5-8.

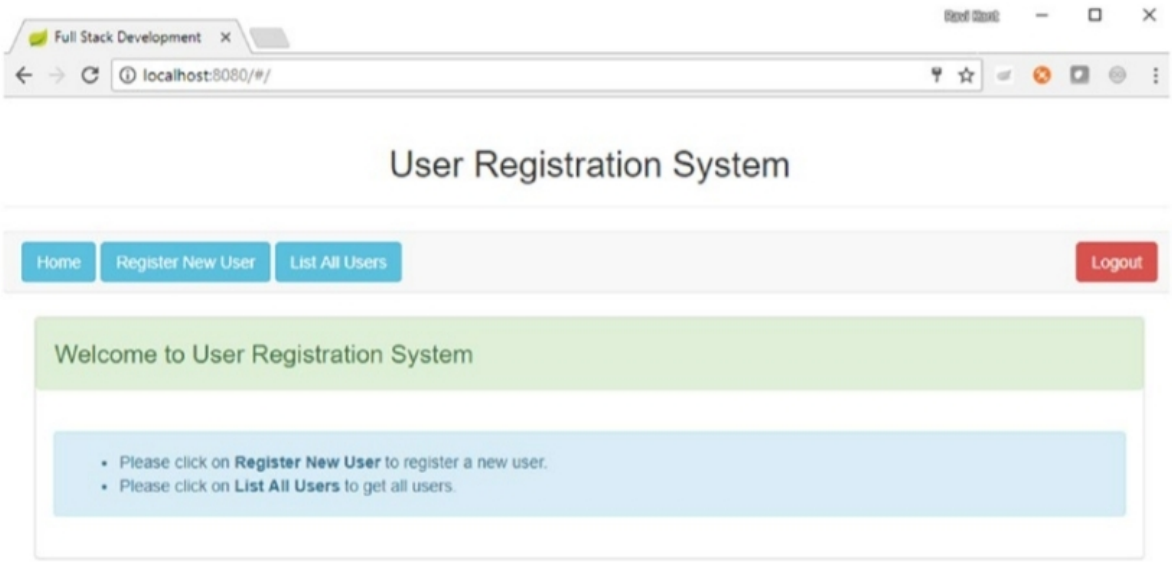

Figure 5-8. *Navigating to the home page after logging in to the application*

Once you have logged in to your application, you can directly call an endpoint from the same browser. Let's open a new tab in the same browser where you performed a successful login to your application and visit `http://localhost:8080/api/user/`, as shown in Figure 5-9.

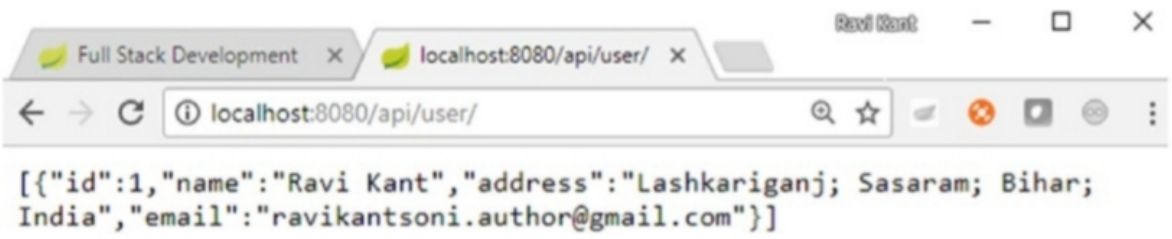

Figure 5-9. *Calling the endpoint from the browser*

Once you click the Logout button from the web page (other than the login page after successful login), you will be redirected to the login page. To verify whether you have successfully logged out from the application, try calling the endpoint from the same browser by visiting `http://localhost:8080/api/user/`. You will be prompted with a pop-up, as shown in Figure 5-10.

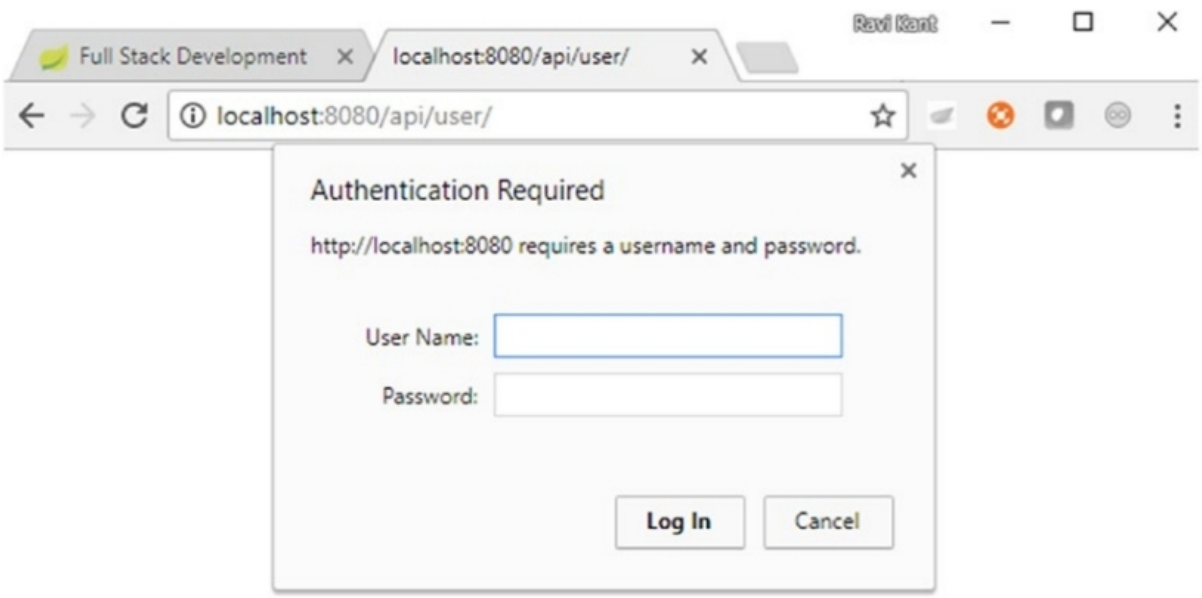

Figure 5-10. *Authentication pop-up opening once a user is logged out and tries to call the endpoint*

Summary

In this chapter, you successfully consumed a secure RESTful service using AngularJS. You started by enabling Basic Authentication in Spring Security. Then you sent an authorization header with each request. Finally, you created a login page to perform the login/logout authentication process.

In the next chapter, you will build a RESTful client and test the RESTful services.