What You Will Build

2023-07-11

This guide walks you through the process of creating a simple web application with resources that are protected by Spring Security.

# What You Will Build

You will build a Spring MVC application that secures the page with a login form that is backed by a fixed list of users.

# What You Need

<https://raw.githubusercontent.com/spring-guides/getting-started-macros/main/prereq_editor_jdk_buildtools.adoc>

<https://raw.githubusercontent.com/spring-guides/getting-started-macros/main/how_to_complete_this_guide.adoc>

# Starting with Spring Initializr

You can use this [pre-initialized project](https://start.spring.io/#!type=maven-project&language=java&platformVersion=2.5.5&packaging=jar&jvmVersion=11&groupId=com.example&artifactId=securing-web&name=securing-web&description=Demo%20project%20for%20Spring%20Boot&packageName=com.example.securing-web&dependencies=web,thymeleaf) and click Generate to download a ZIP file. This project is configured to fit the examples in this tutorial.

To manually initialize the project:

1. Navigate to <https://start.spring.io>. This service pulls in all the dependencies you need for an application and does most of the setup for you.
2. Choose either Gradle or Maven and the language you want to use. This guide assumes that you chose Java.
3. Click **Dependencies** and select **Spring Web** and **Thymeleaf**.
4. Click **Generate**.
5. Download the resulting ZIP file, which is an archive of a web application that is configured with your choices.

If your IDE has the Spring Initializr integration, you can complete this process from your IDE.

You can also fork the project from Github and open it in your IDE or other editor.

# Create an Unsecured Web Application

Before you can apply security to a web application, you need a web application to secure. This section walks you through creating a simple web application. Then you will secure it with Spring Security in the next section.

The web application includes two simple views: a home page and a “Hello, World” page. The home page is defined in the following Thymeleaf template (from src/main/resources/templates/home.html):

<!DOCTYPE html>  
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="https://www.thymeleaf.org" xmlns:sec="https://www.thymeleaf.org/thymeleaf-extras-springsecurity3">  
 <head>  
 <title>Spring Security Example</title>  
 </head>  
 <body>  
 <h1>Welcome!</h1>  
 <h2><p th:utext="${jsonText}"></p></h2>  
  
 <p>Click <a th:href="@{/hello}">here</a> to see a greeting.</p>  
 <div th:if="${#authorization.expression('hasRole(''ADMIN'')')}">  
 <p>Click <a th:href="@{/register}">here</a> to see the registration page.</p>  
 <p>Click <a th:href="@{/admin}">here</a> to see the admin page.</p>  
 <p>Click <a th:href="@{/listUsers}">here</a> to see the list users page.</p>  
 </div>  
 <div th:if="${#authorization.expression('hasRole(''CATALOG\_MGR'')')}">  
 <p>Click <a th:href="@{/catalog}">here</a> to see the catalog page.</p>  
 </div>  
 <div th:if="${#authorization.expression('hasRole(''USER'')')}">  
 <p>Click <a th:href="@{/user}">here</a> to see the user page.</p>  
 </div>  
  
 <form th:action="@{/logout}" method="post">  
 <input type="submit" value="Sign Out"/>  
 </form>  
  
 <!-- <p>Click <a th:href="@{/admin}">here</a> to see the admin page.</p>-->  
<!-- <p>Click <a th:href="@{/catalog}">here</a> to see the catalog page.</p>-->  
  
 </body>  
</html>

This simple view includes a link to the /hello page, which is defined in the following Thymeleaf template (from src/main/resources/templates/hello.html):

<!DOCTYPE html>  
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="https://www.thymeleaf.org"  
 xmlns:sec="https://www.thymeleaf.org/thymeleaf-extras-springsecurity3">  
 <head>  
 <title>Hello World!</title>  
 </head>  
 <body>  
 <h1 th:inline="text">Hello [[${#httpServletRequest.remoteUser}]]!</h1>  
 <form th:action="@{/logout}" method="post">  
 <input type="submit" value="Sign Out"/>  
 </form>  
 </body>  
</html>

The web application is based on Spring MVC. As a result, you need to configure Spring MVC and set up view controllers to expose these templates. The following listing (from src/main/java/com/example/securingweb/MvcConfig.java) shows a class that configures Spring MVC in the application:

package com.example.securingweb;  
  
import org.springframework.context.annotation.Configuration;  
import org.springframework.web.servlet.config.annotation.ViewControllerRegistry;  
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;  
  
@Configuration  
public class MvcConfig implements WebMvcConfigurer {  
  
 @Override  
 public void addViewControllers(ViewControllerRegistry registry) {  
 registry.addViewController("/home").setViewName("home");  
 registry.addViewController("/").setViewName("home");  
 registry.addViewController("/hello").setViewName("hello");  
 registry.addViewController("/admin").setViewName("admin");  
 registry.addViewController("/catalog").setViewName("catalog");  
 registry.addViewController("/login").setViewName("login");  
 registry.addViewController("/user").setViewName("user");  
 registry.addViewController("/register").setViewName("register");  
 registry.addViewController("/listUsers").setViewName("listUsers");  
 }  
  
}

The addViewControllers() method (which overrides the method of the same name in WebMvcConfigurer) adds four view controllers. Two of the view controllers reference the view whose name is home (defined in home.html), and another references the view named hello (defined in hello.html). The fourth view controller references another view named login. You will create that view in the next section.

At this point, you could jump ahead to “[Run the Application](#run_the_app)” and run the application without having to log in to anything.

Now that you have an unsecured web application, you can add security to it.

# Set up Spring Security

Suppose that you want to prevent unauthorized users from viewing the greeting page at /hello. As it is now, if visitors click the link on the home page, they see the greeting with no barriers to stop them. You need to add a barrier that forces the visitor to sign in before they can see that page.

You do that by configuring Spring Security in the application. If Spring Security is on the classpath, Spring Boot [automatically secures all HTTP endpoints](http://docs.spring.io/spring-boot/docs/2.4.3/reference/htmlsingle/#boot-features-security) with “basic” authentication. However, you can further customize the security settings. The first thing you need to do is add Spring Security to the classpath.

With Gradle, you need to add two lines (one for the application and one for testing) in the dependencies closure in build.gradle, as the following listing shows:

implementation 'org.springframework.boot:spring-boot-starter-security'  
implementation 'org.springframework.security:spring-security-test'

The following listing shows the finished build.gradle file:

plugins {  
 id 'org.springframework.boot' version '2.7.1'  
 id 'io.spring.dependency-management' version '1.0.11.RELEASE'  
 id 'java'  
}  
  
group = 'com.example'  
version = '0.0.1-SNAPSHOT'  
sourceCompatibility = '1.8'  
  
repositories {  
 mavenCentral()  
}  
  
dependencies {  
 implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'  
 implementation 'org.springframework.boot:spring-boot-starter-web'  
 implementation 'org.springframework.boot:spring-boot-starter-security'  
 implementation 'org.springframework.security:spring-security-test'  
 testImplementation('org.springframework.boot:spring-boot-starter-test')  
}  
  
test {  
 useJUnitPlatform()  
}

With Maven, you need to add two extra entries (one for the application and one for testing) to the <dependencies> element in pom.xml, as the following listing shows:

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-security</artifactId>  
</dependency>  
<dependency>  
 <groupId>org.springframework.security</groupId>  
 <artifactId>spring-security-test</artifactId>  
 <scope>test</scope>  
</dependency>

The following listing shows the finished pom.xml file:

<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
 <parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>2.7.1</version>  
 <relativePath/> <!-- lookup parent from repository -->  
 </parent>  
 <groupId>com.example</groupId>  
 <artifactId>securing-web-complete</artifactId>  
 <version>0.0.1-SNAPSHOT</version>  
 <name>securing-web-complete</name>  
 <description>Demo project for Spring Boot</description>  
  
 <properties>  
 <java.version>1.8</java.version>  
<!-- <agent.args>-javaagent:/Users/EricB/aceiss/opentelemetry-examples/examples/distro/agent/build/libs/agent-1.0-SNAPSHOT-all.jar</agent.args>-->  
 <!--/Users/EricB/aceiss/opentelemetry-java/aceiss-javaagent-1.0.jar-->  
 <debug.port>8089</debug.port>  
<!-- <agent.args>-javaagent:/Users/EricB/aceiss/opentelemetry-java/aceiss-javaagent-1.0.jar</agent.args>-->  
 <agent.args>-javaagent:/Users/EricB/aceiss/mac/javaagent/agent/build/libs/aceiss-javaagent.jar</agent.args>  
 <debug.args>-Xdebug -Xrunjdwp:transport=dt\_socket,server=y,suspend=n,address=${debug.port}</debug.args>  
 <boot.jvm.memory>-Xmx1536M</boot.jvm.memory>  
 <!-- <boot.jvm.args>${agent.args} ${boot.jvm.memory} ${debug.args}</boot.jvm.args> -->  
 <boot.jvm.args>${agent.args} ${boot.jvm.memory} ${debug.args}</boot.jvm.args>  
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
 <project.uri>${project.baseUri}</project.uri>  
 </properties>  
   
 <dependencies>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-thymeleaf</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-security</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>com.google.code.gson</groupId>  
 <artifactId>gson</artifactId>  
 <version>2.10</version>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.security</groupId>  
 <artifactId>spring-security-test</artifactId>  
 <scope>test</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.thymeleaf.extras</groupId>  
 <artifactId>thymeleaf-extras-springsecurity5</artifactId>  
 <version>3.0.4.RELEASE</version>  
 </dependency>  
  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-test</artifactId>  
 <scope>test</scope>  
 </dependency>  
 <dependency>  
 <groupId>com.h2database</groupId>  
 <artifactId>h2</artifactId>  
 <scope>runtime</scope>  
 </dependency>  
  
 <dependency>  
 <groupId>javax.persistence</groupId>  
 <artifactId>javax.persistence-api</artifactId>  
 <version>2.2</version>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-jdbc</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-jpa</artifactId>  
 </dependency>  
 </dependencies>  
  
 <!-- <build>  
 <plugins>  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 </plugin>  
 </plugins>  
 </build> -->  
 <build>  
 <finalName>spring-io-security</finalName>  
 <pluginManagement>  
 <plugins>  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 <configuration>  
 <jvmArguments>${boot.jvm.args}</jvmArguments>  
 </configuration>  
 </plugin>  
 </plugins>  
 </pluginManagement>  
 <plugins>  
 <plugin>  
 <groupId>org.zeroturnaround</groupId>  
 <artifactId>jrebel-maven-plugin</artifactId>  
 <version>1.1.7</version>  
 <executions>  
 <execution>  
 <id>generate-rebel-xml</id>  
 <phase>process-resources</phase>  
 <goals>  
 <goal>generate</goal>  
 </goals>  
 </execution>  
 </executions>  
 </plugin>  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-assembly-plugin</artifactId>  
 <version>3.4.2</version>  
 <configuration>  
 <archive>  
 <manifest>  
 <addClasspath>true</addClasspath>  
 <mainClass>com.example.securingweb.SecuringWebApplication</mainClass>  
 </manifest>  
 </archive>  
 <descriptorRefs>  
 <descriptorRef>jar-with-dependencies</descriptorRef>  
 </descriptorRefs>  
 </configuration>  
  
 <executions>  
 <execution>  
 <id>make-assembly</id>  
 <phase>package</phase>  
 <goals>  
 <goal>single</goal>  
 </goals>  
 </execution>  
 </executions>  
   
 </plugin>  
  
 </plugins>  
 </build>  
 <profiles>  
 <profile>  
 <id>jrebel</id>  
 <properties>  
<!-- <boot.jvm.args>${boot.jvm.memory} ${debug.args} ${jrebel.agent.args}</boot.jvm.args>-->  
 <boot.jvm.args>${boot.jvm.memory} ${debug.args} </boot.jvm.args>  
 </properties>  
 </profile>  
 </profiles>  
  
</project>

The following security configuration (from src/main/java/com/example/securingweb/WebSecurityConfig.java) ensures that only authenticated users can see the secret greeting:

package com.example.securingweb;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.http.HttpMethod;  
import org.springframework.security.config.annotation.method.configuration.EnableGlobalMethodSecurity;  
import org.springframework.security.config.annotation.web.builders.HttpSecurity;  
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;  
import org.springframework.security.web.SecurityFilterChain;  
  
@Configuration  
@EnableWebSecurity  
@EnableGlobalMethodSecurity(prePostEnabled = true, securedEnabled = true)  
public class WebSecurityConfig {  
  
 @Bean  
 public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {  
 http  
 .csrf(csrf -> csrf.disable())  
 .authorizeHttpRequests((requests) -> requests  
 .antMatchers("/h2-console/\*\*").permitAll()  
 .antMatchers("/").permitAll()  
 .antMatchers(HttpMethod.GET, "/register.html").permitAll()  
 .antMatchers(HttpMethod.POST, "/register").permitAll()  
// .antMatchers("/home").hasRole("USER")  
// .antMatchers("/admin/\*\*").hasRole("ADMIN")  
// .antMatchers("/catalog/\*\*").hasRole("CATALOG\_MGR")  
 .anyRequest().authenticated()  
 )  
  
 .formLogin((form) -> form  
 .loginPage("/login")  
 .successForwardUrl("/home")  
 .permitAll()  
 )  
 .logout((logout) -> logout.permitAll())  
 .headers().frameOptions().disable()  
 ;  
  
 return http.build();  
 }  
  
  
/\*  
 @Bean  
 public UserDetailsService userDetailsService() {  
 UserDetails user =  
 User.withDefaultPasswordEncoder()  
 .username("user")  
 .password("password")  
 .roles("USER")  
 .build();  
  
 UserDetails user2 =  
 User.withDefaultPasswordEncoder()  
 .username("adminb")  
 .password("password")  
 .roles("USER","ADMIN","CATALOG\_MGR")  
 .build();  
  
 UserDetails user3 =  
 User.withDefaultPasswordEncoder()  
 .username("catmgr")  
 .password("password")  
 .roles("USER","CATALOG\_MGR")  
 .build();  
   
 return new InMemoryUserDetailsManager(user, user2, user3);  
 }  
\*/  
}

The WebSecurityConfig class is annotated with @EnableWebSecurity to enable Spring Security’s web security support and provide the Spring MVC integration. It also exposes two beans to set some specifics for the web security configuration:

The SecurityFilterChain bean defines which URL paths should be secured and which should not. Specifically, the / and /home paths are configured to not require any authentication. All other paths must be authenticated.

When a user successfully logs in, they are redirected to the previously requested page that required authentication. There is a custom /login page (which is specified by loginPage()), and everyone is allowed to view it.

The UserDetailsService bean sets up an in-memory user store with a single user. That user is given a user name of user, a password of password, and a role of USER.

Now you need to create the login page. There is already a view controller for the login view, so you need only to create the login view itself, as the following listing (from src/main/resources/templates/login.html) shows:

<!DOCTYPE html>  
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="https://www.thymeleaf.org"  
 xmlns:sec="https://www.thymeleaf.org/thymeleaf-extras-springsecurity3">  
 <head>  
 <title><p th:utext="${loginText}"></p></title>  
 </head>  
 <body>  
 <div th:if="${param.error}">  
 Invalid username and password.  
 </div>  
 <div th:if="${param.logout}">  
 You have been logged out.  
 </div>  
 <form th:action="@{/login}" method="post">  
 <div><label> User Name : <input type="text" name="username"/> </label></div>  
 <div><label> Password: <input type="password" name="password"/> </label></div>  
 <div><input type="submit" value="Sign In"/></div>  
 </form>  
 </body>  
</html>

This Thymeleaf template presents a form that captures a username and password and posts them to /login. As configured, Spring Security provides a filter that intercepts that request and authenticates the user. If the user fails to authenticate, the page is redirected to /login?error, and your page displays the appropriate error message. Upon successfully signing out, your application is sent to /login?logout, and your page displays the appropriate success message.

Last, you need to provide the visitor a way to display the current user name and sign out. To do so, update the hello.html to say hello to the current user and contain a Sign Out form, as the following listing (from src/main/resources/templates/hello.html) shows:

<!DOCTYPE html>  
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="https://www.thymeleaf.org"  
 xmlns:sec="https://www.thymeleaf.org/thymeleaf-extras-springsecurity3">  
 <head>  
 <title>Hello World!</title>  
 </head>  
 <body>  
 <h1 th:inline="text">Hello [[${#httpServletRequest.remoteUser}]]!</h1>  
 <form th:action="@{/logout}" method="post">  
 <input type="submit" value="Sign Out"/>  
 </form>  
 </body>  
</html>

We display the username by using Spring Security’s integration with HttpServletRequest#getRemoteUser(). The “Sign Out” form submits a POST to /logout. Upon successfully logging out, it redirects the user to /login?logout.

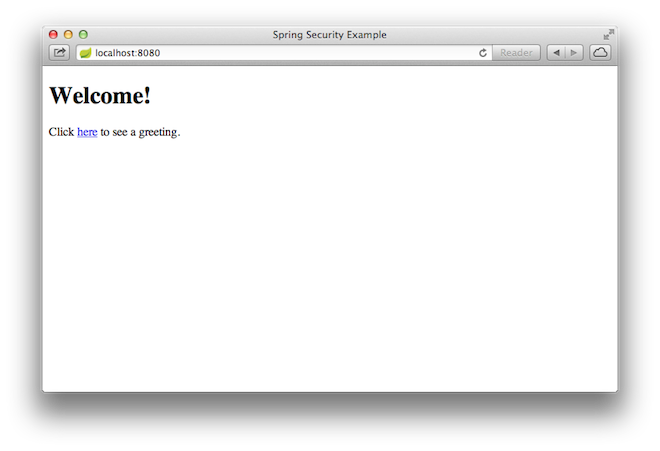
# Run the Application

The Spring Initializr creates an application class for you. In this case, you need not modify the class. The following listing (from src/main/java/com/example/securingweb/SecuringWebApplication.java) shows the application class:

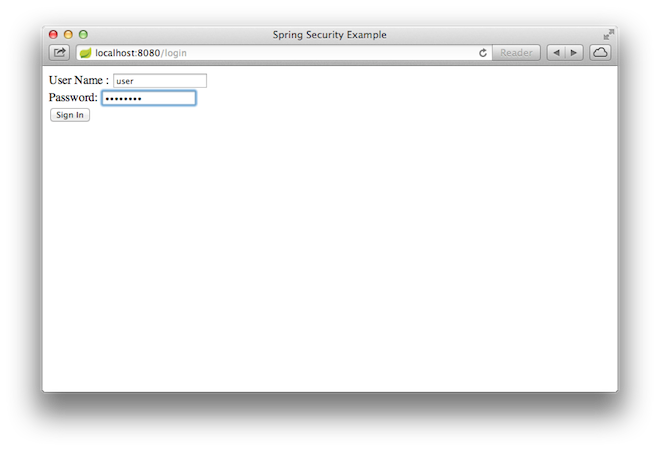
package com.example.securingweb;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
@SpringBootApplication  
public class SecuringWebApplication {  
  
 public static void main(String[] args) throws Throwable {  
 SpringApplication.run(SecuringWebApplication.class, args);  
 }  
  
}

<https://raw.githubusercontent.com/spring-guides/getting-started-macros/main/build_an_executable_jar_subhead.adoc> <https://raw.githubusercontent.com/spring-guides/getting-started-macros/main/build_an_executable_jar_with_both.adoc>

Once the application starts up, point your browser to http://localhost:8080. You should see the home page, as the following image shows:

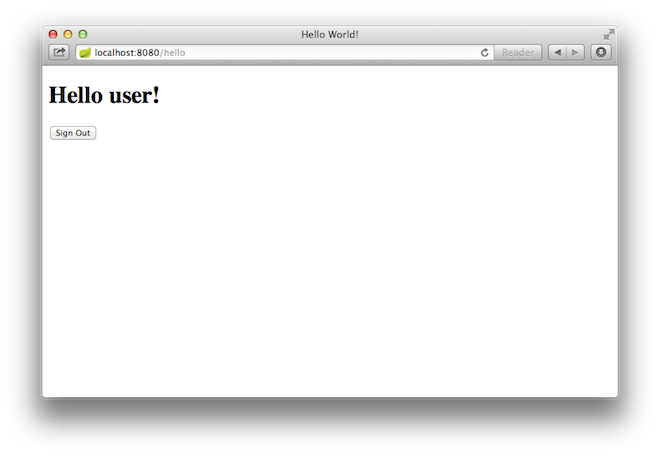


When you click on the link, it attempts to take you to the greeting page at /hello. However, because that page is secured and you have not yet logged in, it takes you to the login page, as the following image shows:



If you jumped down here with the unsecured version, you do not see the login page. You should back up and write the rest of the security-based code.

At the login page, sign in as the test user by entering user and password for the username and password fields, respectively. Once you submit the login form, you are authenticated and then taken to the greeting page, as the following image shows:



If you click on the **Sign Out** button, your authentication is revoked, and you are returned to the login page with a message indicating that you are logged out.

# Summary

Congratulations! You have developed a simple web application that is secured with Spring Security.

# See Also

The following guides may also be helpful:

* [Building an Application with Spring Boot](https://spring.io/guides/gs/spring-boot/)
* [Serving Web Content with Spring MVC](https://spring.io/guides/gs/serving-web-content/)
* [Spring Security Architecture](https://spring.io/guides/topicals/spring-security-architecture/) (Reference guide)
* [Spring Security and Angular JS](https://spring.io/guides/tutorials/spring-security-and-angular-js/) (Tutorial)

<https://raw.githubusercontent.com/spring-guides/getting-started-macros/main/footer.adoc>

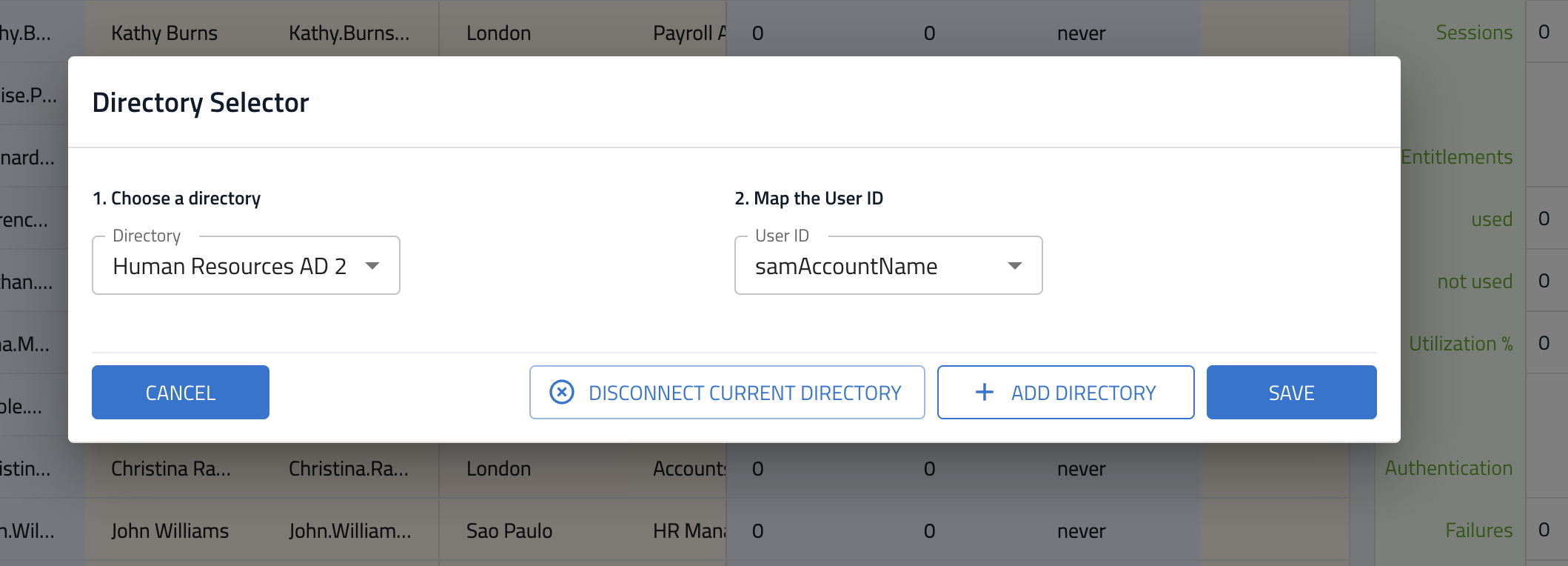
* Sailpoint users in this app:

Michael.Miller

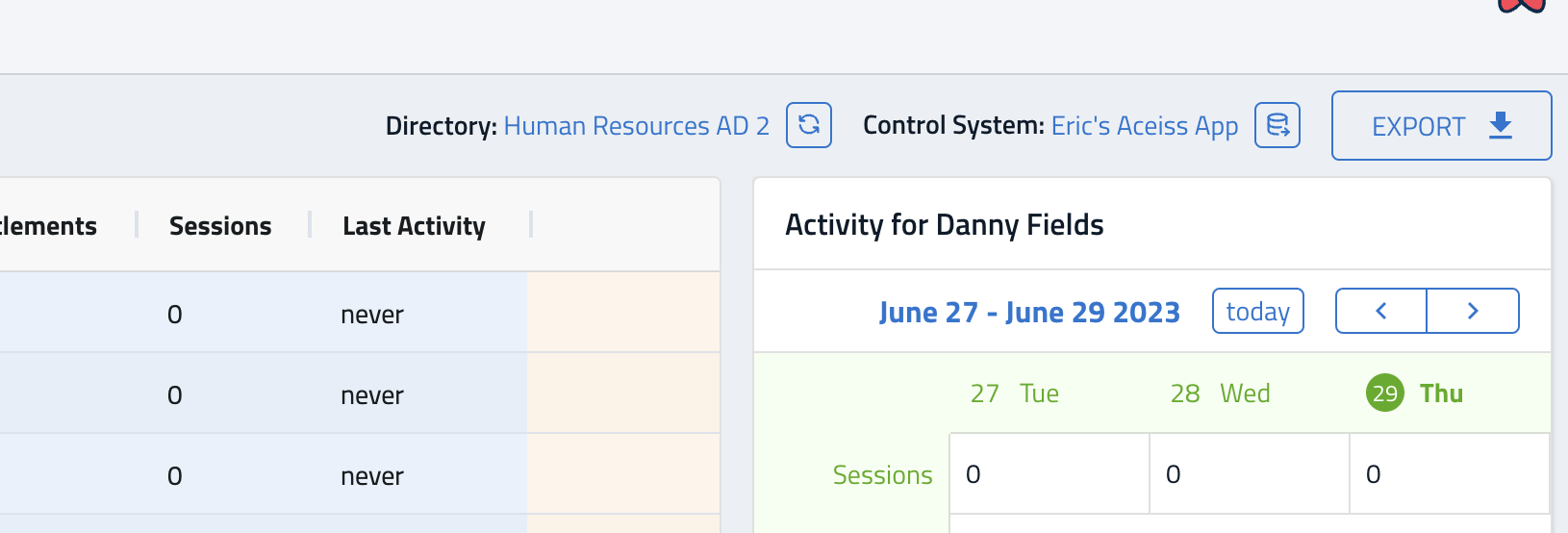
Maria.White

Charles.Harris

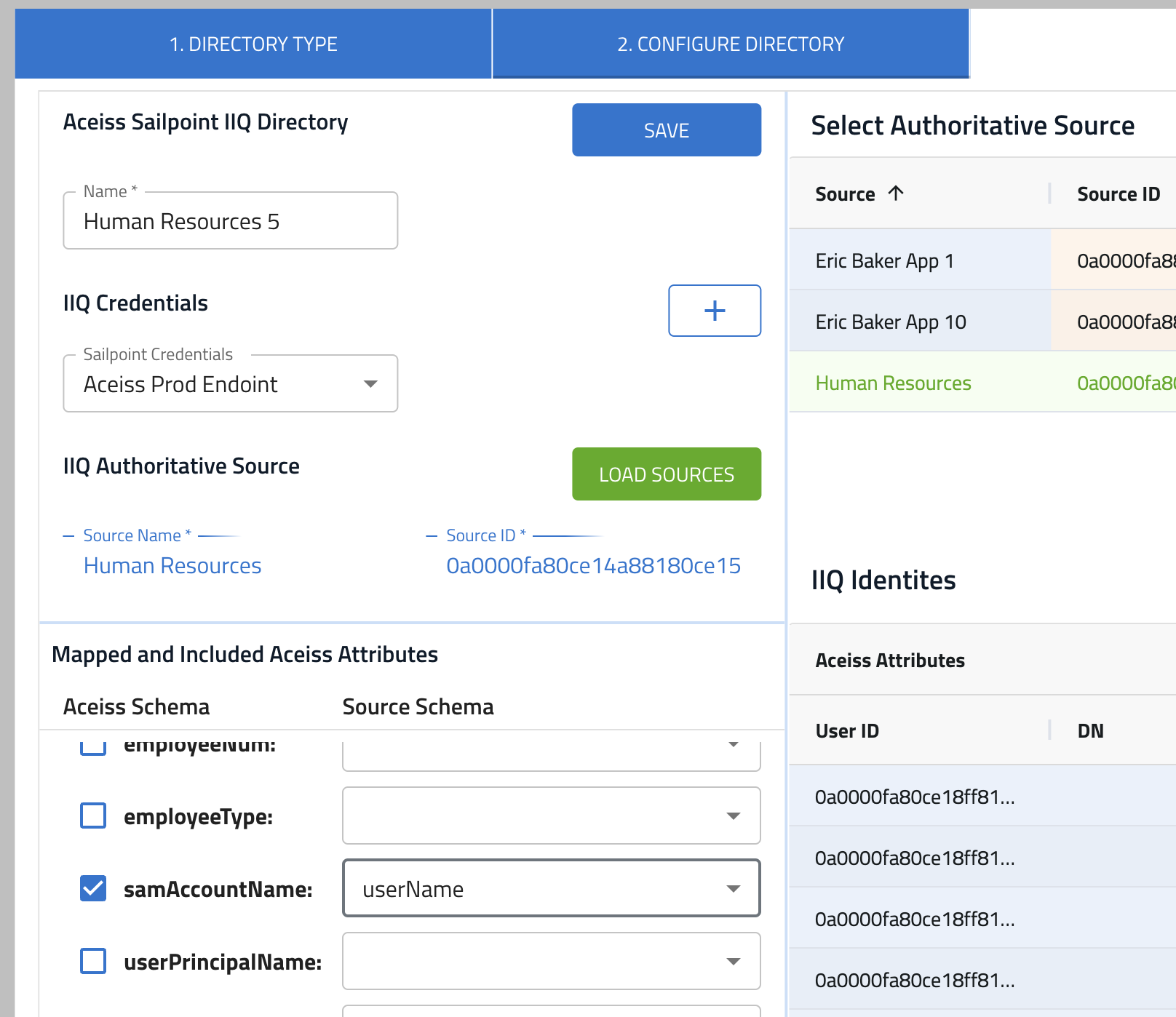
Within the Directories Section in the Aceiss app, within the "Mapped and Included Aceiss Attributes" section, select samAccountName and map it to the userName of the Source Schema



Within the Source Application section, select the "Directory" for the Sailpoint Active Directory created above.



Select the samAccountName as the "Map the UserID" value



Then, when you look at the Activities page, the Spring Basic users should be tied to the same user within the Sailpoint Authoritative Source.

