A Simple Web App with Spring Boot, Spring Security and Stormpath – in 15 Minutes

by Micah Silverman
November 3, 2015

Java (https://stormpath.com/blog/category/java)

UPDATE: We recently released a revision to our Stormpath Spring Security integration. You no longer have to inherit from a special Stormpath security configurer adapter. Instead, you apply a Stormpath DSL (domain specific language). Look below to see how easy this is.

Here at Stormpath, we Spring Boot. It makes it so easy and fun to build rich Java webapps.

We're very excited for our latest Java SDK release (https://github.com/stormpath/stormpath-sdk-java) which includes a major overhaul to our Spring Security and Spring Boot support.

If you've built a web app before, you know that all the "user stuff" is a royal pain. Stormpath gives developers all that "user stuff" out-of-the-box (https://docs.stormpath.com/java/spring-boot-web/quickstart.html) so you can get on with what you really care about – your app! By the time you're done with this tutorial (< 15 minutes, I promise), you'll have a fully-working Spring Boot webapp that protects user access to restricted paths with Spring Security and is backed by Stormpath.

We'll focus on our Spring Boot integration (https://github.com/stormpath/stormpath-sdk-java/tree/master/extensions/spring/boot) to roll out a simple Spring Boot web application example, with a complete user registration and login system. It will have these features:

- Login and Registration pages
- Password reset workflows
- Restricting access according to Group membership
- The ability to easily enable other Stormpath features in our Java library (API authentication, SSO, social login, and more)

In this demo we will be using the stormpath-default-spring-boot-starter (https://github.com/stormpath/stormpath-sdk-java/tree/master/extensions/spring/boot/stormpath-default-spring-boot-starter). Its modular design bakes in Spring Boot 1.4.1 and Spring Security 4.1.2 as well as Spring Boot WebMVC and the Thymeleaf templating engine. I will be using my Mac, the Terminal app, and the IntelliJ IDE.

All the code for this tutorial can be found here (https://github.com/stormpath/spring-boot-spring-security-tutorial).

Throughout this post, you can see the example code in action by clicking on the Deploy to Heroku button. All you need to do is register for a free Heroku (https://heroku.com) account.

What Is Stormpath?

Stormpath is an API service that allows developers to create, edit, and securely store
user accounts and user account data, and connect them with one or multiple applications. Our API enables you to:

- Authenticate and authorize your users
- Store data about your users
- Perform password and social based login
- Send password reset messages
- Issue API keys for API-based web apps
- And much more! Check out our Product Documentation (https://docs.stormpath.com/)

In short: we make user account management a lot easier, more secure, and more
scalable than what you're probably used to.

Ready to get started? Register for a free developer account here (https://api.stormpath.com/register).

Start Your Spring Boot Web App Project

Got your Stormpath developer account? Great! Let's get started…

Whether you are a Maven maven (http://www.urbandictionary.com/define.php?term=maven) or Gradle grete (http://www.urbandictionary.com/define.php?term=Grete), getting your project setup is a snap.

Here's a `pom.xml` file to start with:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xml

    <modelVersion>4.0.0</modelVersion>

    <groupId>com.stormpath.sample</groupId>
    <artifactId>stormpath-spring-boot-spring-security-
    <version>0.1.0</version>
    <name>Spring Boot Spring Security Stormpath Tutori
    <description>A simple Spring Boot Web MVC applicat

    <dependencies>
        <dependency>
            <groupId>com.stormpath.spring</groupId>
            <artifactId>stormpath-default-spring-boot-
            <version>1.1.1</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</gro
                <artifactId>spring-boot-maven-plugin</
                <version>1.4.1.RELEASE</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>repackage</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>

</project>
```

And, here's a `build.gradle` file to start with:

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boo
    }
}

apply plugin: 'java'
apply plugin: 'maven'
apply plugin: 'spring-boot'

group = 'com.stormpath'
version = '0.1.0'

repositories {
    mavenCentral()
}

dependencies {
    compile group: 'com.stormpath.spring', name: 'stor
}
```

You may notice that both for Maven and Gradle there is a single dependency: `stormpath-default-spring-boot-starter`. Yup. That's it. As you will see below, that one dependency gives you all of the Spring Boot, Spring Security and Stormpath magic at once.

Gather Your API Credentials and Application Href

The connection between your app and Stormpath is secured with an "API Key Pair". You will provide these keys
to your web app and it will use them when it communicates with Stormpath. You can download your API key pair from our Admin Console (https://api.stormpath.com). After you log in you can

download your API key pair from the homepage, it will download the `apiKey.properties` file – we will use this in a moment.

While you are in the Admin Console you want to get the href for your default Stormpath Application. In Stormpath, an Application object is used to link your web app to your user stores inside Stormpath. All new developer accounts have an app called "My Application". Click on "Applications" in the Admin Console, then click on "My Application". On that page you will see the Href for the Application. Copy this — we will need it later.

Note: If you use the Heroku Deploy button below, a Stormpath account will automatically be provisioned and the API Keys will automatically be set for you.

Writing the Spring Boot Application

The code for this section can be found in the LockedDown tag of the code repo.

Deploy to Heroku

(https://heroku.com/deploy?template=https://github.com/stormpath/spring-boot-spring-security-tutorial/tree/02-LockedDown)

We need three small Java classes and an html template to fire up the first version of our webapp. They're small enough that I will put them right here. Let's get to it!

Spring Boot Application Entry Point

All Spring Boot applications have an entry point that works just like an ordinary Java program. It has a `main` method and everything.

Here's `Application.java`:

```java
Java
@SpringBootApplication
public class Application  {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, arg
    }
}
```

6 lines of code including the `@SpringBootApplication` annotation kicks off the party.

Spring Security Configuration

By default (and by best security practices), Spring Security locks down your entire application. It locks it down to the point that it's not even accessible! While this conforms to best security practices, it's not terribly useful. Additionally, we need to hook Spring Security and Stormpath together. That brings us to our `SpringSecurityWebAppConfig.java`:

```java
Java
```

```java
@Configuration
public class SpringSecurityWebAppConfig extends WebS
    @Override
    protected void configure(HttpSecurity http) thro
        http.apply(stormpath());
    }
}
```

The `@Configuration` annotation causes Spring Boot to instantiate this class as a configuration. `.apply(stormpath())` hooks all of the Stormpath goodness (authentication and authorization workflows) into Spring Security.

Because there is no further configuration in the `configure` method, we will still see the default behavior of having everything locked down. However, instead of the default Spring Security authentication flows, we will see the default Stormpath flows. That is, attempting to browse to any path in the application will result in a redirect to the Stormpath `login` page.

So, a 1 line method call and we've got security!

Spring WebMVC Ties It All Together

Our security configuration above ensures that all paths in the application will be secured.

In this Spring Boot web app tutorial, we utilize the Controller to determine how requested paths get directed to display which templates.

Here's our `HelloController.java`:

Java

```java
1  @Controller
2  public class HelloController {
3      @RequestMapping("/")
4      String home() {
5          return "home";
6      }
7  }
```

The `@Controller` annotation signals Spring Boot that this is a controller. We have one path defined on line 3, `/`. Line 5 returns the Thymeleaf template named `home`.

7 lines of code and we have Model-View-Controller (MVC) routing.

Bring Us `home.html`

By default, the Thymeleaf templating engine will look for templates returned from controllers in a folder called `templates` in your classpath. It will also look for the default extension of `.html`.

So, when our controller above returns `"home"`, Thymeleaf will find the template in `resources/templates/home.html`.

Let's take a look at the `home.html` file:

XHTML

```
1  <html xmlns:th="http://www.thymeleaf.org">
2      <head>
3          <!--/*/ <th:block th:include="fragments/hea
4      </head>
5      <body>
6          <div class="container-fluid">
7              <div class="row">
8                  <div class="box col-md-6 col-md-off
9                      <div class="stormpath-header">
10                         <img src="http://stormpath
11                     </div>
12
13                     <h1 th:inline="text">Hello, [[$
14                     <a th:href="@{/logout}" class='
15                 </div>
16             </div>
17         </div>
18     </body>
19 </html>
```

Line 1 sets up the `th` namespace for Thymeleaf.

Line 3 looks like an html/xml comment. However, this is a directive that Thymeleaf picks up on to include a fragment in this template. The fragment is found at: `resources/templates/fragments/head.html`. This fragment contains all the setup to hook in Bootstrap styling for our views.

Lines 13 and 14 is where the action is. Since every pathway in our application is locked down, we know that the only way to get to this page is after having logged in. Part of the Stormpath magic is that once logged in, an `account` object is always in scope to your views. Line 13 shows the logged in user's full name. Line 14 provides a link to log out when clicked.

For more information on working with Thymeleaf templates, click here (http://www.thymeleaf.org/).

Let's Fire It Up!

Creating a Stormpath account, 15 lines of Java code and 19 lines of html template code (3 of which are significant) has brought us to the point of a fully functional Spring Boot WebMVC app protected by Spring Security and backed by Stormpath.

If you stored your `apiKey.properties` file from before in the standard location: `~/.stormpath/apiKey.properties` and if you have only the default Stormpath Application that was created for you, no other configuration is necessary to start up the application.

Here's the Maven way:

```
mvn clean package
mvn spring-boot:run
```

Note: The `spring-boot-maven-plugin` also creates an uber-jar due to the presence of the `repackage` execution. You can exercise the same code by just running java:

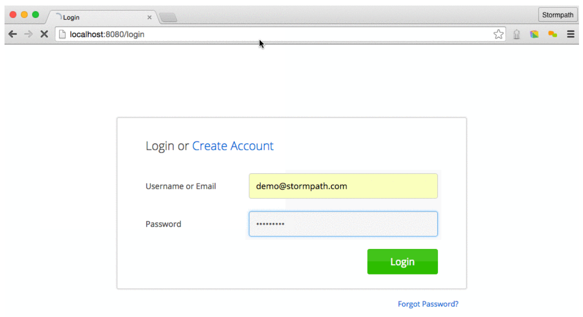
```
mvn clean package
java -jar target/*.jar
```

Here's the Gradle way:

```
gradle clean build
java -jar build/libs/*.jar
```

Really? Is that it? Yup!

You can browse to http://localhost:8080/ (http://localhost:8080/) and see it in action.



What if you stored your `apiKey.properties` file somewhere else or you have other Stormpath Applications defined? No Problem!

Remember the Application Href you saved from earlier? We are going to use it now.

Here's the Maven way:

```
mvn clean package
STORMPATH_API_KEY_FILE=~/.stormpath/apiKey.properties
STORMPATH_APPLICATION_HREF=https://api.stormpath.com/v
mvn spring-boot:run
```

Here's the Gradle way:

```
gradle clean build
STORMPATH_API_KEY_FILE=~/.stormpath/apiKey.properties
STORMPATH_APPLICATION_HREF=https://api.stormpath.com/v
java -jar build/libs/*.jar
```

By adding `STORMPATH_API_KEY_FILE` and `STORMPATH_APPLICATION_HREF` environment variables to the command line, we can easily tell our app where to find the api keys and which Stormpath Application to use.

The Stormpath Java SDK has an extremely flexible configuration mechanism. We will see more of that below when we get to restricting access to your application by Stormpath Group membership.

Refined Access Control With Spring Security

The code for this section can be found in the BasicAccessControl (https://github.com/stormpath/spring-boot-spring-security-tutorial/tree/03-BasicAccessControl) tag of the code repo.

Deploy to Heroku
(https://heroku.com/deploy?template=https://github.com/stormpath/spring-boot-spring-security-tutorial/tree/03-BasicAccessControl)

In the previous section, we created an application that was locked up tight. Every path, including `/`, required you to be logged in first.

Perhaps you want a publicly accessible home page. Perhaps you want some parts of the site that any authenticated user can get to and another part of the site where only members belonging to a particular group can get to.

We are going to explore those fine grained controls in this section.

We'll start by allowing public access to the home page. Users will still have to authenticate to access any another page.

Spring Security: Your Bouncer at the Door

Remember our empty `SpringSecurityWebAppConfig.java` from before? We are going to add a little something to it now:

Java

```java
1   @Configuration
2   public class SpringSecurityWebAppConfig extends Web
3       @Override
4       protected void configure(HttpSecurity http) thr
5           http
6               .apply(stormpath()).and()
7               .authorizeRequests()
8               .antMatchers("/").permitAll();
9       }
10  }
```

Spring Security provides (http://projects.spring.io/spring-security/) a fluent interface (http://en.wikipedia.org/wiki/Fluent_interface) for providing access rules.

On Lines 5 – 8 above, we are building up a rule set for how Spring Security will allow access to our application.

You might state it like this in plain English:

```
Permit anyone to go to the front door
Ensure that they've authenticated for anything else
```

The rules we are specifying take precedence before the default behavior of locking everything down.

Let's update our `home.html` template as well:

XHTML

```
1  <html xmlns:th="http://www.thymeleaf.org">
2      <head>
3          <!--/*/ <th:block th:include="fragments/hea
4      </head>
5      <body>
6          <div class="container-fluid">
7              <div class="row">
8                  <div class="box col-md-6 col-md-off
9                      <div class="stormpath-header">
10                         <img src="https://stormpath
11                     </div>
12
13                     <!--/* displayed if account IS
14                     <div th:if="${account}">
15                         <h1 th:inline="text">Hello,
16                         <form method="post" th:acti
17                             <a href="/restricted" c
18                             <input type="submit" cl
19                         </form>
20                     </div>
21
22                     <!--/* displayed if account IS
23                     <div th:unless="${account}">
24                         <h1>Who are you?</h1>
25                         <a href="/restricted" class
26                     </div>
27                 </div>
28             </div>
29         </div>
30     </body>
31 </html>
```

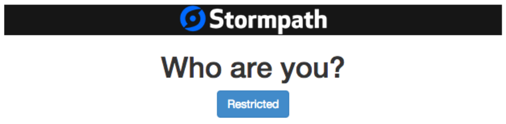
Notice how we now have two distinct sections. The first starts on line 14 and is displayed if the user is logged in.

The second section starting on line 23 is displayed if the user is not logged in.

This is Thymeleaf templates in action. It provides very powerful controls for conditionally showing parts of a template.

Before we make any additional changes, let's pause here and startup the app as before.

When you browse to http://localhost:8080 (http://localhost:8080), you'll see the unauthenticated version of the `home` template.



(http://stormpath.com/wp-content/uploads/2015/11/restricted.png)

Click the `Restricted` button, and you'll be redirected to the `login` form as expected. After you authenticate, you'll end up at a `404` page, because we haven't defined the restricted page yet.

Let's set that up to finish up this section.

Defining the restricted page is as easy as adding a route in our controller and creating a template to show. Here's the updated `HelloController.java`:

Java

```java
@Controller
public class HelloController {

    @RequestMapping("/")
    String home() {
        return "home";
    }

    @RequestMapping("/restricted")
    String restricted() {
        return "restricted";
    }
}
```

And, here's a new `restricted.html` template:

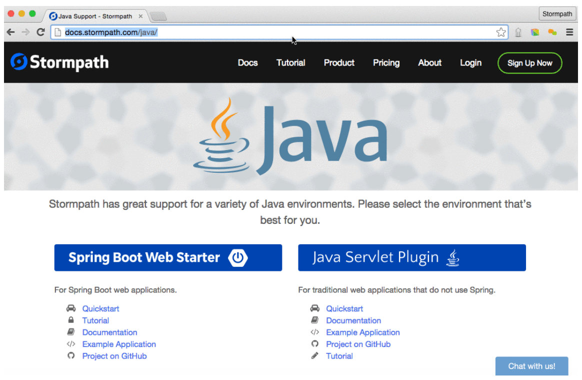
```xhtml
<html xmlns:th="http://www.thymeleaf.org">
    <head>
        <!--/*/ <th:block th:include="fragments/hea
    </head>
    <body>
        <div class="container-fluid">
            <div class="row">
                <div class="box col-md-6 col-md-off
                    <div class="stormpath-header">
                        <img src="http://stormpath.
                    </div>

                    <h1 th:inline="text">[[${accour
                    <a href="/" class="btn btn-prir

                </div>
            </div>
        </div>
    </body>
</html>
```

Notice how we re-use the `head` fragment to provide Bootstrap styling for this template.

Re-start the app again, and you will get the full experience. Take note of how the home page changes depending on whether or not you are logged in.



Stormpath has great support for a variety of Java environments. Please select the environment that's best for you.

Spring Boot Web Starter

For Spring Boot web applications.

- Quickstart
- Tutorial
- Documentation
- Example Application
- Project on GitHub

Java Servlet Plugin

For traditional web applications that do not use Spring.

- Quickstart
- Documentation
- Example Application
- Project on GitHub
- Tutorial

Chat with us!

Spring Security Access Control By Group Membership

The code for this section can be found in the GroupAccessControl tag of the code repo.

Deploy to Heroku
(https://heroku.com/deploy?template=https://github.com/stormpath/spring-boot-spring-security-tutorial/tree/04-GroupAccessControl)

Spring Security provides a set of annotations and a rich expression language for controlling access to methods in your application. Among the most commonly used Spring

Security Annotations is `@PreAuthorize`. And, among the most commonly used SpringEL expressions is `hasRole`.

Stormpath integrates with this mechanism connecting Stormpath Groups to Spring Security roles.

Let's look at the code for this and then break it down. We are going to add a new service that restricts access by Group membership. Here's `AdminService`:

```java
1  @Service
2  public class AdminService {
3      @PreAuthorize("hasRole(@roles.ADMIN)")
4      public boolean ensureAdmin() {
5          return true;
6      }
7  }
```

Line 3 above is the key, here. The annotation along with the SpringEL (http://docs.spring.io/spring/docs/current/spring-framework-reference/html/expressions.html) expression could be stated in plain English as:

```
Before this method is even entered,
check to see that user is authenticated and
is a member of the ADMIN group
```

The `check to see that user is authenticated` part of this may not be obvious. What's going on is that a `@PreAuthorize` check can only be done on an authenticated user. Spring Security is smart enough to check that the user is logged in before checking to see if they are a member of the specified group.

Let's dig in to that Spring Expression Language above. Where is Spring Security looking for `@roles.ADMIN`? The `@` symbol is special – it identifies a Java bean. In this case, a bean named `roles`. Defined inside that bean we expect to find a constant named `ADMIN`.

Ultimately, `hasRole` needs to be checking against a unique Href representing a Stormpath Group. So, our `ADMIN` constant needs to be a Java `String` that holds the Href to our Stormpath Group used for admin.

To complete this configuration and to make it awesomely dynamic, we need a new class called `Roles.java`:

```java
@Component
public class Roles {
    public final String ADMIN;

    @Autowired
    public Roles(Environment env) {
        ADMIN = env.getProperty("stormpath.authorize
    }
}
```

These 9 lines are so amazingly powerful, I'm geeking out over here! Let's dig in.

By annotating this class with `@Component` on line 1, Spring will instantiate it and expose it as a bean. Guess what the name of the bean is? Spring will take the name of the class and camel-case it to derive the bean name by default. So, the bean name is `roles`. Sweet!

The `@Autowired` annotation on line 5 causes Spring `Environment` object to be passed into the constructor. Inside the constructor, we have our only opportunity to set `ADMIN` since it's declared `final` – a requirement to be able to use it inside the `hasRoles` clause.

The last piece of the puzzle utilizes some Stormpath configuration magic. Notice that we are setting the value of `ADMIN` to whatever the environment property named `stormpath.authorized.group.admin` is set to. This is standard Spring. If you have a property in your `application.properties` file with this name, it will be available in the Spring Environment.

Stormpath adds the ability to set this as a system environment variable alleviating the need to have the value – a Stormpath Group Href in this case – hardcoded anywhere in your application.

Typically, system environment variables are all caps with words separated by underscores. The Stormpath Java SDK automatically converts these system variables into the lowercase dotted notation.

Dig this:

```
STORMPATH_AUTHORIZED_GROUP_ADMIN=https://api.stormpath
java -jar build/libs/spring-boot-spring-security-tutor
```

Behind the scenes, Stormpath will convert the `STORMPATH_AUTHORIZED_GROUP_ADMIN` system environment variable to a Spring `stormpath.authorized.group.admin` environment variable. That will get picked up by our code above.

Phew! Who thought such magic could be achieved in so few lines of code!

Now, we need to wire the `AdminService` to our Controller. Here are the relevant parts of our updated `HelloController.java`:

```java
@Controller
public class HelloController {

    @Autowired
    AdminService adminService;

    ...

    @RequestMapping("/admin")
    String admin() {
        adminService.ensureAdmin();
        return "admin";
    }
}
```

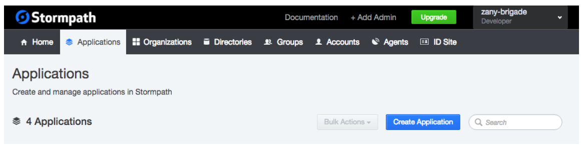
`AdminService` is Autowired in on lines 4 & 5. Notice on line 11, we are calling the `adminService.ensureAdmin` method. If the logged in user is NOT in the `ADMIN` group, a `403` (forbidden) response will be generated.

The last bit of code housekeeping to do here is to create an `admin.html` template. In the code (https://github.com/stormpath/spring-boot-spring-security-tutorial/tree/04-GroupAccessControl) that is associated with this post, there's a simple `admin.html` template that shows a nicely formatted message confirming that you are, indeed, an admin.

Now, to see this in action, you'll need to do a little bit of Stormpath housekeeping in the admin console (https://api.stormpath.com).

Here are the steps:

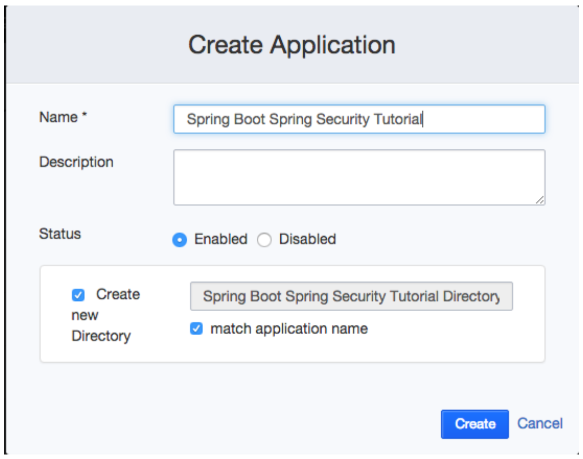
1. Create a new Application



Applications

Create and manage applications in Stormpath

4 Applications



Create Application

Name * Spring Boot Spring Security Tutorial

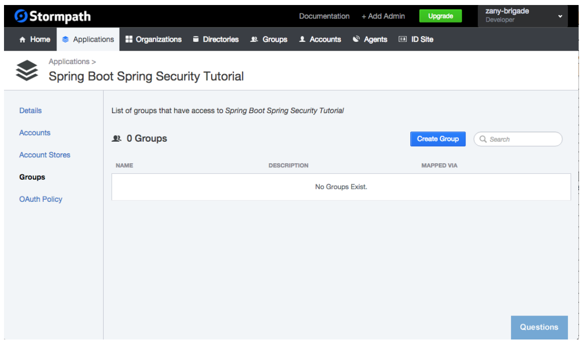
Description

Status [✓ Enabled] Disabled

[✓] Create new Directory: Spring Boot Spring Security Tutorial Directory

[✓] match application name

2. Create a new Group called "admin" for the Application

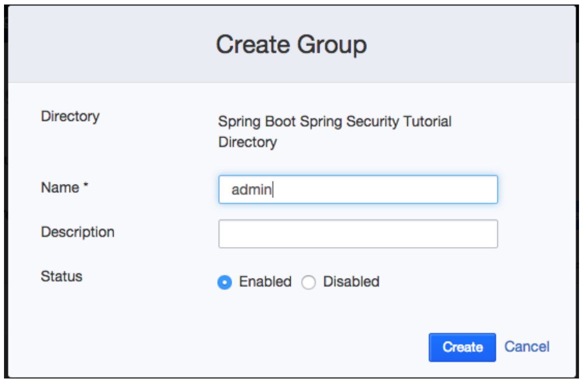


Spring Boot Spring Security Tutorial

List of groups that have access to *Spring Boot Spring Security Tutorial*

0 Groups

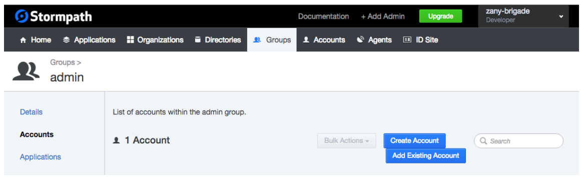
NAME	DESCRIPTION	MAPPED VIA
No Groups Exist.		



Create Group

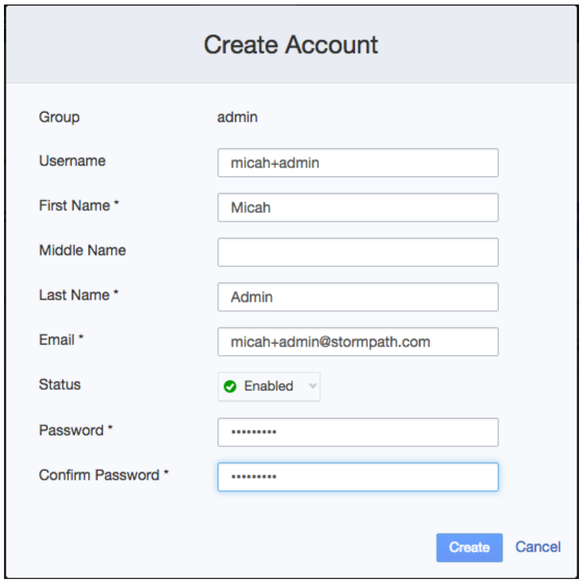
Directory	Spring Boot Spring Security Tutorial Directory
Name *	admin
Description	
Status	[✓ Enabled] Disabled

3. Create a new Account in the admin Group



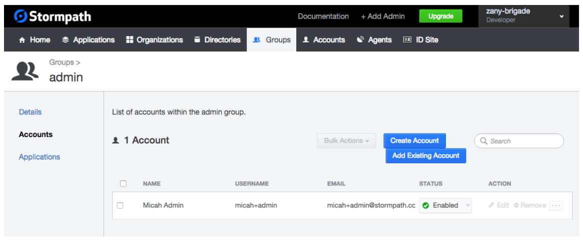
List of accounts within the admin group.

1 Account



Create Account

Group	admin
Username	micah+admin
First Name *	Micah
Middle Name	
Last Name *	Admin
Email *	micah+admin@stormpath.com
Status	Enabled
Password *	•••••••••
Confirm Password *	•••••••••

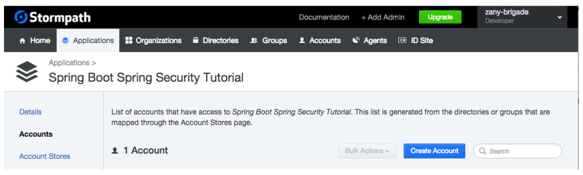


List of accounts within the admin group.

1 Account

NAME	USERNAME	EMAIL	STATUS	ACTION
Micah Admin	micah+admin	micah+admin@stormpath.cc	Enabled	Edit Remove

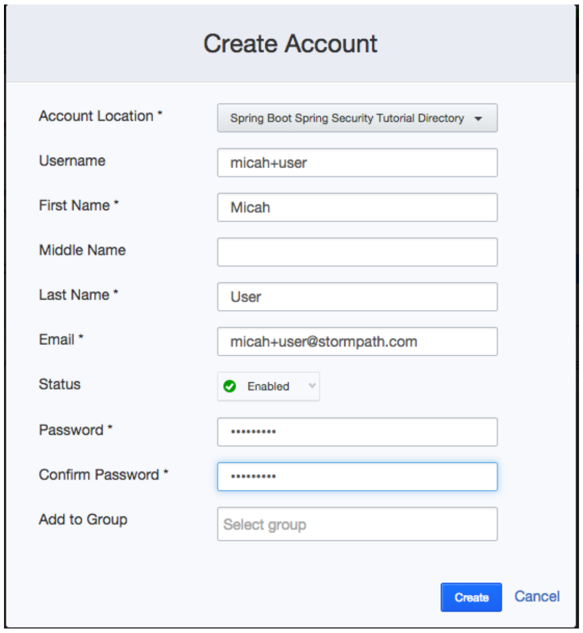
4. Create a new Account NOT in the admin Group



Spring Boot Spring Security Tutorial

List of accounts that have access to *Spring Boot Spring Security Tutorial*. This list is generated from the directories or groups that are mapped through the Account Stores page.

1 Account



Account Location *	Spring Boot Spring Security Tutorial Directory
Username	micah+user
First Name *	Micah
Middle Name	
Last Name *	User
Email *	micah+user@stormpath.com
Status	Enabled
Password *	•••••••••
Confirm Password *	•••••••••
Add to Group	Select group

In the code (https://github.com/stormpath/spring-boot-spring-security-tutorial/tree/04-GroupAccessControl) that is associated with this post, I've also included a handy error page so that if you are not in the admin group you get a nicely formatted page rather than the default `403` page.

Let's see it in action. We are going to start up the app as before, only this time we are going to use the Hrefs found in the Admin Console (https://api.stormpath.com) for the new Application and Group you created in the previous step.

Here's the Maven way:

```
mvn clean package
STORMPATH_API_KEY_FILE=~/.stormpath/apiKey.properties
STORMPATH_APPLICATION_HREF=https://api.stormpath.com/v
STORMPATH_AUTHORIZED_GROUP_ADMIN=https://api.stormpath
mvn spring-boot:run
```

Here's the Gradle way:

```
gradle clean build
STORMPATH_API_KEY_FILE=~/.stormpath/apiKey.properties
STORMPATH_APPLICATION_HREF=https://api.stormpath.com/v
STORMPATH_AUTHORIZED_GROUP_ADMIN=https://api.stormpath
java -jar build/libs/spring-boot-spring-security-tutor
```

Browse to the `/admin` page.

If you log in as the user you created in the Stormpath `admin` Group (micah+admin@stormpath.com in my case), you will have access to the admin page. If you log in as the user you created that's NOT in the Stormpath `admin` Group (micah+user@stormpath.com in my case), you will get the forbidden page.

No matter who you log in as, you will have access to the `/restricted` page as we setup before.