Securing Web Applications with Apache Shiro

This document is an introductory step-by-step tutorial to securing a web application with Apache Shiro. It assumes an introductory knowledge of Shiro, and assumes familiarity with at least the following two introductory documents:

- Application Security with Apache Shiro
- Apache Shiro 10 Minute Tutorial

This step-by-step tutorial should take ~ 45 minutes to 1 hour to complete. When you are finished, you will have a very good idea of how Shiro works in a web application.

Table of Contents

Overview

While Apache Shiro's core design goals allow it to be used to secure *any* JVM-based application, such as command line applications, server daemons, web apps, etc, this guide will focus on the most common use case: securing a web application running in a Servlet container, such as Tomcat or Jetty.

Prerequisites

The following tools are expected to be installed on your local development machine in order to follow along with this tutorial.

- Git (tested w/ 1.7)
- Java SDK 7
- Maven 3
- Your favorite IDE, like IntelliJ IDEA or Eclipse, or even a simple text editor to view files and make changes.

Tutorial Format

This is a step-by-step tutorial. The tutorial, and all of its steps, exist as a Git repository. When you clone the git repository, the `master` branch is your starting point. Each step in the tutorial is a separate branch. You can follow along simply by checking out the git branch that reflects the tutorial step you are reviewing.

The Application

The web application we will build is a super webapp that can be used as a starting point for your own application. It will demonstrate user login, logout, user-specific welcome messages, access control to certain parts of the web application, and integration with a plugglable security data store.

We will start by setting up the project, including the build tool and declaring dependencies, as well as configuring the servlet `web.xml` file to launch the web application and the Shiro environment.

Once we complete setup, we will then layer in individual pieces of functionality, including integration with a security data store, then enabling user login, logout, and access control.

Project Setup

Instead of having to manually set up a directory structure and initial set of basic files, we've done this for you in a git repository.

1. Fork the tutorial project

On GitHub, visit the tutorial project and click the `Fork` button on the upper right.

2. Clone your tutorial repository

Now that you have forked the repository to your own GitHub account, clone it on your local machine:

```
$ git clone git@github.com:$YOUR_GITHUB_USERNAME/apache-shiro-tutorial-webapp.git
```

(where `$YOUR_GITHUB_USERNAME` is your own GitHub username of course)

You can now `cd` into the cloned directory and see the project structure:

```
$ cd apache-shiro-tutorial-webapp
```

3. Review project structure

After cloning the repo, your current `master` branch will have the following structure:

```
apache-shiro-tutorial-webapp/
  |-- src/
  |   |-- main/
  |     |-- resources/
  |       |-- logback.xml
  |     |-- webapp/
  |       |-- WEB-INF/
  |         |-- web.xml
  |       |-- home.jsp
  |       |-- include.jsp
  |       |-- index.jsp
  |-- .gitignore
  |-- .travis.yml
  |-- LICENSE
  |-- README.md
  |-- pom.xml
```

Here is what each means:

- `pom.xml`: the Maven project/build file. It has Jetty configured so you can test your web app right away by running `mvn jetty:run`.
- `README.md`: a simple project readme file
- `LICENSE`: the project's Apache 2.0 license
- `.travis.yml`: A Travis CI config file in case you want to run continuous integration on your project to ensure it always builds.
- `.gitignore`: A git ignore file, containing suffixes and directories that shouldn't be checked in to version control.
- `src/main/resources/logback.xml`: A simple Logback config file. For this tutorial, we've chosen SLF4J as our logging API and Logback as the logging implementation. This could have easily been Log4J or JUL.
- `src/main/webapp/WEB-INF/web.xml`: Our initial `web.xml` file that we'll configure soon to enable Shiro.
- `src/main/webapp/include.jsp`: A page that contains common imports and declarations, included in other JSP pages. This allows us to manage imports and declarations in one place.
- `src/main/webapp/home.jsp`: our webapp's simple default home page. Includes `include.jsp` (as will others, as we will soon see).
- `src/main/webapp/index.jsp`: the default site index page - this merely forwards the request on to our `home.jsp` homepage.

4. Run the webapp

Now that you've cloned the project, you can run the web application by executing the following on the command line:

```
$ mvn jetty:run
```

Next, open your web browser to localhost:8080, and you'll see the home page with a **Hello, World!** greeting.

Hit `ctl-C` (or `cmd-C` on a mac) to shut down the web app.

Step 1: Enable Shiro

Our initial repository `master` branch is just a simple generic web application that could be used as a template for any application. Let's add the bare minimum to enable Shiro in the web app next.

Perform the following git checkout command to load the `step1` branch:

```
$ git checkout step1
```

Checking out this branch, you will find two changes:

1. A new `src/main/webapp/WEB-INF/shiro.ini` file was added, and
2. `src/main/webapp/WEB-INF/web.xml` was modified.

1a: Add a `shiro.ini` file

Shiro can be configured in many different ways in a web application, depending on the web and/or MVC framework you use. For example, you can configure Shiro via Spring, Guice, Tapestry, and many many more.

To keep things simple for now, we'll start a Shiro environment by using Shiro's default (and very simple) INI-based configuration.

If you checked out the `step1` branch, you'll see the contents of this new `src/main/webapp/WEB-INF/shiro.ini` file (header comments removed for brevity):

```
[main]

# Let's use some in-memory caching to reduce the number of runtime lookups against Stormpath.
# A real application might want to use a more robust caching solution (e.g. ehcache or a
# distributed cache).  When using such caches, be aware of your cache TTL settings: too high
# a TTL and the cache won't reflect any potential changes in Stormpath fast enough.  Too low
# and the cache could evict too often, reducing performance.
cacheManager = org.apache.shiro.cache.MemoryConstrainedCacheManager
securityManager.cacheManager = $cacheManager
```

This .ini contains simply a `[main]` section with some minimal configuration:

- It defines a new `cacheManager` instance. Caching is an important part of Shiro's architecture - it reduces constant round-trip communications to various data stores. This example uses a `MemoryConstrainedCacheManager` which is only really good for single JVM applications. If your application is deployed across multiple hosts (e.g. a clustered webserver farm), you will want to use a clustered CacheManager implementation instead.
- It configures the new `cacheManager` instance on the Shiro `securityManager`. A Shiro `SecurityManager` instance always exists, so it did not need to be defined explicitly.

1b: Enable Shiro in `web.xml`

While we have a `shiro.ini` configuration, we need to actually *load* it and start a new Shiro environment and make that environment avaiable to the web application.

We do all of this by adding a few things to the existing `src/main/webapp/WEB-INF/web.xml` file:

```
<listener>
    <listener-class>org.apache.shiro.web.env.EnvironmentLoaderListener</listener-class>
</listener>

<filter>
    <filter-name>ShiroFilter</filter-name>
    <filter-class>org.apache.shiro.web.servlet.ShiroFilter</filter-class>
</filter>

<filter-mapping>
    <filter-name>ShiroFilter</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>INCLUDE</dispatcher>
    <dispatcher>ERROR</dispatcher>
</filter-mapping>
```

- The `<listener>` declaration defines a `ServletContextListener` that starts up the Shiro environment (including the Shiro `SecurityManager`) upon web application startup. By default, this listener automatically knows to look for our `WEB-INF/shiro.ini` file for Shiro configuration.
- The `<filter>` declaration defines the master `ShiroFilter`. This filter is expected to filter *all* requests into the web application so Shiro can peform necessary identity and access control operations before allowing a request to reach the application.
- The `<filter-mapping>` declaration ensures that *all* request types are filed by the `ShiroFilter`. Often `filter-mapping` declarations don't specify `<dispatcher>` elements, but Shiro needs them all defined so it can filter all of the different request types that might execute for a web app.

1c: Run the webapp

After checking out the `step1` branch, go ahead and run the web app:

```
$ mvn jetty:run
```

This time, you will see log output similar to the following, indicating that Shiro is indeed running in your webapp:

```
16:04:19.807 [main] INFO  o.a.shiro.web.env.EnvironmentLoader - Starting Shiro environment
initialization.
16:04:19.904 [main] INFO  o.a.shiro.web.env.EnvironmentLoader - Shiro environment initialized in 95
ms.
```

Hit `ctl-C` (or `cmd-C` on a mac) to shut down the web app.

Step 2: Connect to a User Store

Perform the following git checkout command to load the `step2` branch:

```
$ git checkout step2
```

Now we have Shiro integrated and running within a webapp. But we haven't actually told Shiro to do anything yet!

Before we can login, or logout, or perform role-based or permission-based access control, or anything else security related, we need users!

We will need to configure Shiro to access a *User Store* of some type, so it can look up users to perform login attempts, or check roles for security decisions, etc. There are many types of user stores that any application might need to access: maybe you store users in a MySQL database, maybe in MongoDB, maybe your company stores user accounts in LDAP or Active Directory, maybe you store them in a simple file, or some other proprietary data store.

Shiro does this via what it calls a `Realm`. From Shiro's documentation:

> Realms act as the 'bridge' or 'connector' between Shiro and your application's security data. When it comes time to actually interact with security-related data like user accounts to perform authentication (login) and authorization (access control), Shiro looks up many of these things from one or more Realms configured for an application.
>
> In this sense a Realm is essentially a security-specific DAO: it encapsulates connection details for data sources and makes the associated data available to Shiro as needed. When configuring Shiro, you must specify at least one Realm to use for authentication and/or authorization. The SecurityManager may be configured with multiple Realms, but at least one is required.
>
> Shiro provides out-of-the-box Realms to connect to a number of security data sources (aka directories) such as LDAP, relational databases (JDBC), text configuration sources like INI and properties files, and more. You can plug-in your own Realm implementations to represent custom data sources if the default Realms do not meet your needs.

So, we need to configure a Realm so we can access users.

2a: Set up Stormpath

In the spirit of keeping this tutorial as simple as possible, without introducing complexity or scope that distracts us from the purpose of learning Shiro, we'll use one of the simplest realms we can: a Stormpath realm.

Stormpath is a cloud hosted user management service, totally free for development purposes. This means that after enabling Stormpath, you'll have the following ready to go:

- A user interface for managing Applications, Directories, Accounts and Groups. Shiro does not provide this at all, so this will be convenient and save time while you go through this tutorial.
- A secure storage mechanism for user passwords. Your application never needs to worry about password security, password comparisons or storing passwords. While Shiro can do these things, you would have to configure them and be aware of cryptographic concepts. Stormpath automates password security so you (and Shiro) don't need to worry about it or be on the hook for 'getting it right'.
- Security workflows like account email verification and password reset via email. Shiro has no support for this, as it is often application specific.
- Hosted/managed 'always on' infrastructure - you don't have to set anything up or maintain anything.

For the purposes of this tutorial, Stormpath is much simpler than setting up a separate RDBMS server and worrying about SQL or password encryption issues. So we'll use that for now.

Of course, Stormpath is only one of many back-end data stores that Shiro can communicate with. We'll cover more complicated data stores and application-specific configuration of them later.

Sign up for Stormpath

1. Fill out and submit the Stormpath registration form. This will send a confirmation email.
2. Click the link in the confirmation email.

Get a Stormpath API Key

A Stormpath API Key is required for the Stormpath Realm to communicate with Stormpath. To get a Stormpath API Key:

1. Log in to the Stormpath Admin Console using the email address and password you used to register with Stormpath.
2. In the top-right corner of the resulting page, visit **Settings > My Account**.
3. On the Account Details page, under **Security Credentials**, click **Create API Key**.

   This will generate your API Key and download it to your computer as an `apiKey.properties` file. If you open the file in a text editor, you will see something similar to the following:

   ```
   apiKey.id = 144JVZINOF5EBNCMG9EXAMPLE
   apiKey.secret = lWxOiKqKPNwJmSldbiSkEbkNjgh2uRSNAb+AEXAMPLE
   ```

4. Save this file in a secure location, such as your home directory in a hidden `.stormpath` directory. For example:

   ```
   $HOME/.stormpath/apiKey.properties
   ```

5. Also change the file permissions to ensure only you can read this file. For example, on *nix operating systems:

   ```
   $ chmod go-rwx $HOME/.stormpath/apiKey.properties
   ```

Register the web application with Stormpath

We have to register our web application with Stormpath to allow the app to use Stormpath for user management and authentication. You register the web app with Stormpath simply by making a REST request, `POST`ing a new Application resource to the Stormpath `applications` URL:

```
curl -X POST --user $YOUR_API_KEY_ID:$YOUR_API_KEY_SECRET \
    -H "Accept: application/json" \
```

```
    -H "Content-Type: application/json" \
    -d '{
            "name" : "Apache Shiro Tutorial Webapp"
        }' \
    'https://api.stormpath.com/v1/applications?createDirectory=true'
```

where:

- $YOUR_API_KEY_ID is the apiKey.id value in apiKey.properties and
- YOUR_API_KEY_SECRET is the apiKey.secret value in apiKey.properties

This will create your application. Here's an example response:

```
{
    "href": "https://api.stormpath.com/v1/applications/aLoNGrAnDoMAppIdHeRe",
    "name": "Apache Shiro Tutorial Webapp",
    "description": null,
    "status": "ENABLED",
    "tenant": {
        "href": "https://api.stormpath.com/v1/tenants/sOmELoNgRaNDoMIdHeRe"
    },
    "accounts": {
        "href": "https://api.stormpath.com/v1/applications/aLoNGrAnDoMAppIdHeRe/accounts"
    },
    "groups": {
        "href": "https://api.stormpath.com/v1/applications/aLoNGrAnDoMAppIdHeRe/groups"
    },
    "loginAttempts": {
        "href": "https://api.stormpath.com/v1/applications/aLoNGrAnDoMAppIdHeR/loginAttempts"
    },
    "passwordResetTokens": {
        "href":
"https://api.stormpath.com/v1/applications/aLoNGrAnDoMAppIdHeRe/passwordResetTokens"
    }
}
```

Make note of the top-level `href`, e.g. `https://api.stormpath.com/v1/applications/$YOUR_APPLICATION_ID` - we will use this href in the `shiro.ini` configuration next.

Create an application test user account

Now that we have an application, we'll want to create a sample/test user for that application:

```
curl -X POST --user $YOUR_API_KEY_ID:$YOUR_API_KEY_SECRET \
    -H "Accept: application/json" \
    -H "Content-Type: application/json" \
    -d '{
            "givenName": "Jean-Luc",
            "surname": "Picard",
            "username": "jlpicard",
            "email": "capt@enterprise.com",
            "password":"Changeme1"
        }' \
 "https://api.stormpath.com/v1/applications/$YOUR_APPLICATION_ID/accounts"
```

Again, don't forget to change `$YOUR_APPLICATION_ID` in the URL above to match your application's ID!

2b: Configure the Realm in `shiro.ini`

Once you choose at least one user store to connect to for Shiro's needs, we'll need to configure a `Realm` that represents that data store and then tell the Shiro `SecurityManager` about it.

If you've checked out the `step2` branch, you'll notice the `shiro.ini` file's `[main]` section now has the following additions:

```
# Configure a Realm to connect to a user datastore.  In this simple tutorial, we'll just point to
Stormpath since it
# takes 5 minutes to set up:
stormpathClient = com.stormpath.shiro.client.ClientFactory
stormpathClient.cacheManager = $cacheManager
stormpathClient.apiKeyFileLocation = $HOME/.stormpath/apiKey.properties
stormpathRealm = com.stormpath.shiro.realm.ApplicationRealm
stormpathRealm.client = $stormpathClient

# Find this URL in your Stormpath console for an application you create:
# Applications -> (choose application name) --> Details --> REST URL
stormpathRealm.applicationRestUrl =
https://api.stormpath.com/v1/applications/$STORMPATH_APPLICATION_ID
stormpathRealm.groupRoleResolver.modeNames = name
securityManager.realm = $stormpathRealm
```

Make the following changes:

1. Change the `$HOME` placeholder to be your actual home directory path, e.g. `/home/jsmith` so the final `stormpathClient.apiKeyFileLocation` value is something like `/home/jsmith/.stormpath/apiKey.properties`. This path must match the location of the `apiKey.properties` file you downloaded from Stormpath in Step 2a.

2. Change the `$STORMPATH_APPLICATION_ID` placeholder to be the actual ID value in the `href` returned from Stormpath at the end of Step 2a. The final `stormpathRealm.applicationRestUrl` value should look something like `https://api.stormpath.com/v1/applications/6hsPwoRZ0hCk6ToytVxi4D` (with a different application ID of course).

2c: Commit your changes

Your replaced `$HOME` and `STORMPATH_APPLICATION_ID` values are specific to your application. Go ahead and commit those changes to your branch:

```
$ git add . && git commit -m "updated app-specific placeholders" .
```

2d: Run the webapp

After making the changes as specified in Step 2b and 2c, go ahead and run the web app:

```
$ mvn jetty:run
```

This time, you will see log output similar to the following, indicating that Shiro and the new Realm are configured properly in your webapp:

```
16:08:25.466 [main] INFO  o.a.shiro.web.env.EnvironmentLoader - Starting Shiro environment
initialization.
16:08:26.201 [main] INFO  o.a.s.c.IniSecurityManagerFactory - Realms have been explicitly set on
the SecurityManager instance - auto-setting of realms will not occur.
16:08:26.201 [main] INFO  o.a.shiro.web.env.EnvironmentLoader - Shiro environment initialized in
731 ms.
```

Hit `ctl-C` (or `cmd-C` on a mac) to shut down the web app.

Step 3: Enable Login and Logout

Now we have users, and we can add, remove and disable them easily in a UI. Now we can start enabling features like login/logout and access control in our application.

Perform the following git checkout command to load the `step3` branch:

```
$ git checkout step3
```

This checkout will load the following 2 additions:

- A new `src/main/webapp/login.jsp` file has been added with a simple login form. We'll use that to login.
- The `shiro.ini` file has been updated to support web (URL)-specific capabilities.

Step 3a: Enable Shiro form login and logout support

The `step3` branch's `src/main/webapp/WEB-INF/shiro.ini` file contains the following 2 additions:

```
[main]

shiro.loginUrl = /login.jsp

# Stuff we've configured here previously is omitted for brevity

[urls]
/login.jsp = authc
/logout = logout
```

`shiro.*` lines

At the top of the `[main]` section, there is a new line:

```
shiro.loginUrl = /login.jsp
```

This is a special configuration directive that tells Shiro "For any of Shiro's default filters that have a `loginUrl` property, I want that property value to be set to `/login.jsp`."

This allows Shiro's default `authc` filter (by default, a `FormAuthenticationFilter`) to know about the login page. This is necessary for the `FormAuthenticationFilter` to work correctly.

The `[urls]` section

The `[urls]` section is a new web-specific INI section.

This section allows you to use a very succinct name/value pair syntax to tell shiro how to filter request for any given URL path. All paths in `[urls]` are relative to the web application's [HttpServletRequest.getContextPath()] (http://java.sun.com/j2ee/sdk_1.3/techdocs/api/javax/servlet/http/HttpServletRequest.html#getContextPath()) value.

These name/value pairs offer an extremely powerful way to filter requests, allowing for all sorts of security rules. A deeper coverage of urls and filter chains is outside the scope of this document, but please do read more about it if you're interested.

For now, we'll cover the two lines that were added:

```
/login.jsp = authc
/logout = logout
```

- The first line indicates “Whenever Shiro sees a request to the `/login.jsp` url, enable the Shiro `authc` filter during the request”.
- The second line means “whenever Shiro sees a request to the `/logout` url, enable the Shiro `logout` filter during the request.”

Both of these filters are a little special: they don’t actually require anything to be ‘behind’ them. Instead of filtering, they will actually just process the request entirely. This means there isn’t anything for you to do for requests to these URLs - no controllers to write! Shiro will handle the requests as necessary.

Step 3b: Add a login page

Since Step 3a enabled login and logout support, now we need to ensure there is actually a `/login.jsp` page to display a login form.

The `step3` branch contains a new `src/main/webapp/login.jsp` page. This is a simple enough bootstrap-themed HTML login page, but there are four important things in it:

1. The form’s `action` value is the empty string. When a form does not have an action value, the browser will submit the form request to the same URL. This is fine, as we will tell Shiro what that URL is shortly so Shiro can automatically process any login submissions. The `/login.jsp = authc` line in `shiro.ini` is what tells the `authc` filter to process the submission.
2. There is a `username` form field. The Shiro `authc` filter will automatically look for a `username` request parameter during login submission and use that as the value during login (many Realms allow this to be an email or a username).
3. There is a `password` form field. The Shiro `authc` filter will automatically look for a `password` request parameter during login submission.
4. There is a `rememberMe` checkbox whose ‘checked’ state can be a ‘truthy’ value (`true`, `t`, `1`, `enabled`, `y`, `yes`, or `on`).

Our login.jsp form just uses the default `username`, `password`, and `rememberMe` form field names. They naems are configurable if you wish to change them - see the `FormAuthenticationFilter` JavaDoc for information.

Step 3c: Run the webapp

After making the changes as specified in Step 2b and 2c, go ahead and run the web app:

```
$ mvn jetty:run
```

Step 3d: Try to Login

With your web browser, navigate to localhost:8080/login.jsp and you will see our new shiny login form.

Enter in a username and password of the account you created at the end of Step 2, and hit ‘Login’. If the login is successful, you will be directed to the home page! If the login fails, you will be shown the login page again.

Tip: If you want a successful login to redirect the user to a different page other than the home page (context path `/`), you can set the `authc.successUrl = /whatever` in the INI’s `[main]` section.

Hit `ctl-C` (or `cmd-C` on a mac) to shut down the web app.

Step 4: User-specific UI changes

It’s usually a requirement to change a web user interface based on who the user is. We can do that easily because Shiro supports a JSP tag library to do things based on the currently logged-in Subject (user).

Perform the following git checkout command to load the `step4` branch:

```
$ git checkout step4
```

This step makes some additions to our `home.jsp` page:

- When the current user viewing the page is not logged in, they will see a ‘Welcome Guest’ message and see the link to the login page.
- When the current user viewing the page *is* logged in, they will see their own name, ‘Welcome *username*’ and a link to log out.

This type of UI customization is very common for a navigation bar, with user controls on the upper right of the screen.

Step 4a: Add the Shiro Tag Library Declaration

The `home.jsp` file was modified to include two lines at the top:

```
<%@ taglib prefix="shiro" uri="http://shiro.apache.org/tags" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

These two JSP page directives allow the Core (`c:`) and Shiro (`shiro:`) taglibraries in the page.

Step 4b: Add Shiro Guest and User tags

The `home.jsp` file was further modified in the page body (right after the `<h1>` welcome message) to include *both* the `<shiro:guest>` and `<shiro:user>` tags:

```
<p>Hi <shiro:guest>Guest</shiro:guest><shiro:user>
<%
    //This should never be done in a normal page and should exist in a proper MVC controller of
some sort, but for this
    //tutorial, we'll just pull out Stormpath Account data from Shiro's PrincipalCollection to
```

```
reference in the
    //<c:out/> tag next:

    request.setAttribute("account",
org.apache.shiro.SecurityUtils.getSubject().getPrincipals().oneByType(java.util.Map.class));

%>
<c:out value="${account.givenName}"/></shiro:user>!
    ( <shiro:user><a href="<c:url value="/logout"/>">Log out</a></shiro:user>
    <shiro:guest><a href="<c:url value="/login.jsp"/>">Log in</a></shiro:guest> )
</p>
```

It's a little hard to read given the formatting, but two tags are used here:

- `<shiro:guest>`: This tag will only display its internal contents if the current Shiro `Subject` is an application 'guest'. Shiro defines a `guest` as any `Subject` that has not logged in to the application, or is not remembered from a previous login (using Shiro's 'remember me' functionality).
- `<shiro:user>`: This tag will only display its internal contents if the current Shiro `Subject` is an application 'user'. Shiro defines a `user` as any `Subject` that is *currently* logged in to (authenticated with) the application or one that is remembered from a previous login (using Shiro's 'remember me' functionality).

The above code snippet will render the following if the Subject is a guest:

```
Hi Guest! (Log in)
```

where 'Log in' is a hyperlink to `/login.jsp`

It will render the following if the Subject is a 'user':

```
Hi jsmith! (Log out)
```

Assuming 'jsmith' is the username of the account logged in. 'Log out' is a hyperlink to the '/logout' url handled by the Shiro `logout` filter.

As you can see, you can turn off or on entire page sections, features and UI components. In addition to `<shiro:guest>` and `<shiro:user>`, Shiro supports many other useful JSP tags that you can use to customize the UI based on various things known about the current `Subject`.

Step 4c: Run the webapp

After checking out the `step4` branch, go ahead and run the web app:

```
$ mvn jetty:run
```

Try visiting localhost:8080 as a guest, and then login. After successful login, you will see the page content change to reflect that you're now a known user!

Hit `ctl-C` (or `cmd-C` on a mac) to shut down the web app.

Step 5: Allow Access to Only Authenticated Users

While you can change page content based on Subject state, often times you will want to restrict entire sections of your webapp based on if someone has **proven** their identity (authenticated) during their current interaction with the web application.

This is especially important if a user-only section of a webapp shows sensitive information, like billing details or the ability to control other users.

Perform the following git checkout command to load the `step5` branch:

```
$ git checkout step5
```

Step 5 introduces the following 3 changes:

1. We added a new section (url path) of the webapp that we want to restrict to only authenticated users.
2. We changed `shiro.ini` to tell shiro to only allow authenticated users to that part of the web app.
3. We modified the home page to change its output based on if the current `Subject` is authenticated or not.

Step 5a: Add a new restricted section

A new `src/main/webapp/account` directory was added. This directory (and all paths below it) simulates a 'private' or 'authenticated only' section of a website that you might want to restrict to only logged in users. The `src/main/webapp/account/index.jsp` file is just a placeholder for a simulated 'home account' page.

Step 5b: Configure `shiro.ini`

`shiro.ini` was modified by adding the following line at the end of the `[urls]` section:

```
/account/** = authc
```

This Shiro filter chain definition means "Any requests to `/account` (or any of its sub-paths) must be authenticated".

But what happens if someone tries to access that path or any of its children paths?

But do you remember in Step 3 when we added the following line to the `[main]` section:

```
shiro.loginUrl = /login.jsp
```

This line automatically configured the `authc` filter with our webapp's login URL.

Based on this line of config, the `authc` filter is now smart enough to know that if the current Subject is not authenticated when accessing `/account`, it will automatically redirect the `Subject` to the `/login.jsp` page. After successful login, it will then automatically redirect the user back to the page they were trying to access (`/account`). Convenient!

Step 5c: Update our home page

The final change for Step 5 is to update the `/home.jsp` page to let the user know they can access the new part of the web site. These lines were added below the welcome message:

```
<shiro:authenticated><p>Visit your <a href="<c:url value="/account"/>">account page</a>.</p>
</shiro:authenticated>
<shiro:notAuthenticated><p>If you want to access the authenticated-only <a href="<c:url
value="/account"/>">account page</a>,
    you will need to log-in first.</p></shiro:notAuthenticated>
```

The `<shiro:authenticated>` tag will only display the contents if the current Subject has already logged in (authenticated) during their current session. This is how the `Subject` knows they can go visit a new part of the website.

The `<shiro:notAuthenticated>` tag will only display the contents if the current Subject is not yet authenticated during their current session.

But did you notice that the `notAuthenticated` content still has a URL to the `/account` section? That's ok - our `authc` filter will handle the login-and-then-redirect flow as described above.

Fire up the webapp with the new changes and try it out!

Step 5d: Run the webapp

After checking out the `step5` branch, go ahead and run the web app:

```
$ mvn jetty:run
```

Try visiting localhost:8080. Once there, click the new `/account` link and watch it redirect you to force you to log in. Once logged in, return to the home page and see the content change again now that you're authenticated. You can visit the account page and the home page as often as you want, until you log out. Nice!

Hit `ctl-C` (or `cmd-C` on a mac) to shut down the web app.

Step 6: Role-Based Access Control

In addition to controlling access based on authentication, it is often a requirement to restrict access to certain parts of the application based on what role(s) are assigned to the current `Subject`.

Perform the following git checkout command to load the `step5` branch:

```
$ git checkout step6
```

Step 6a: Add Roles

In order to perform Role-Based Access Control, we need Roles to exist.

The fastest way to do that in this tutorial is to populate some Groups within Stormpath (in Stormpath, a Stormpath Group can serve the same purpose of a Role).

To do this, log in to the UI and navigate as follows:

Directories > Apache Shiro Tutorial Webapp Directory > Groups

Add the following three groups:

- Captains
- Officers
- Enlisted

(to keep with our Star-Trek account theme :))

Once you've created the groups, add the `Jean-Luc Picard` account to the `Captains` and `Officers` groups. You might want to create some ad-hoc accounts and add them to whatever groups you like. Make sure some of the accounts don't overlap groups so you can see changes based on separate Groups assigned to user accounts.

Step 6b: RBAC Tags

We update the `/home.jsp` page to let the user know what roles they have and which ones they don't. These messages are added in a new `<h2>Roles</h2>` section of the home page:

```
<h2>Roles</h2>

<p>Here are the roles you have and don't have. Log out and log back in under different user
```

```
    accounts to see different roles.</p>

<h3>Roles you have:</h3>

<p>
    <shiro:hasRole name="Captains">Captains<br/></shiro:hasRole>
    <shiro:hasRole name="Officers">Bad Guys<br/></shiro:hasRole>
    <shiro:hasRole name="Enlisted">Enlisted<br/></shiro:hasRole>
</p>

<h3>Roles you DON'T have:</h3>

<p>
    <shiro:lacksRole name="Captains">Captains<br/></shiro:lacksRole>
    <shiro:lacksRole name="Officers">Officers<br/></shiro:lacksRole>
    <shiro:lacksRole name="Enlisted">Enlisted<br/></shiro:lacksRole>
</p>
```

The `<shiro:hasRole>` tag will only display the contents if the current Subject has been assigned the specified role.

The `<shiro:lacksRole>` tag will only display the contents if the current Subject **has not** been assigned the specified role.

Step 6c: RBAC filter chains

An exercise left to the reader (not a defined step) is to create a new section of the website and restrict URL access to that section of the website based on what role is assigned to the current user.

Hint: Create a filter chain definition for that new part of the webapp using the `roles` filter

Step 6d: Run the webapp

After checking out the `step6` branch, go ahead and run the web app:

```
$ mvn jetty:run
```

Try visiting localhost:8080 and log in with different user accounts that are assigned different roles and watch the home page's **Roles** section content change!

Hit `ctl-C` (or `cmd-C` on a mac) to shut down the web app.

Step 7: Permission-Based Access Control

Role-based access control is good for many use cases, but it suffers from one major problem: you can't add or delete roles at runtime. Role checks are hard-coded with role names, so if you changed the role names or role configuration, or add or remove roles, you have to go back and change your code!

Because of this, Shiro has a powerful marquis feature: built-in support for *permissions*. In Shiro, a permission is a raw statement of functionality, for example 'open a door' 'create a blog entry', 'delete the `jsmith` user', etc. Permissions reflect your application's raw functionality, so you only need to change permission checks when you change your application's functionlity - not if you want to change your role or user model.

To demonstrate this, we will create some permissions and assign them to a user, and then customize our web UI based on a user's authorization (permissions).

Step 7a: Add Permissions

Shiro `Realms` are read-only components: every data store models roles, groups, permissions, accounts, and their relationships differently, so Shiro doesn't have a 'write' API to modify these resources. To modify the underlying the model objects, you just modify them directly via whatever API you desire. Your Shiro Realm then knows how to read this information and represent it in a format Shiro understands.

As such, since we're using Stormpath in this sample app, we'll assign permissions to an account and group in a Stormpath API-specific way.

Let's execute a cURL request to add some permissions to our previously created Jean-Luc Picard account. Using that account's `href` URL, we'll post some `apacheShiroPermissions` to the account via custom data:

```
curl -X POST --user $YOUR_API_KEY_ID:$YOUR_API_KEY_SECRET \
    -H "Accept: application/json" \
    -H "Content-Type: application/json" \
    -d '{
            "apacheShiroPermissions": [
                "ship:NCC-1701-D:command",
                "user:jlpicard:edit"
            ]
        }' \
"https://api.stormpath.com/v1/accounts/$JLPICARD_ACCOUNT_ID/customData"
```

where `$JLPICARD_ACCOUNT_ID` matches the uid of the Jean-Luc Picard you created at the beginning of this tutorial.

This adds two permissions directly to the Stormpath Account:

- `ship:NCC-1701-D:command`
- `user:jlpicard:edit`

These use Shiro's WildcardPermission syntax.

The first basically means *the ability to 'command' the 'ship' with identifier 'NCC-1701-D'*. This is an example of an *instance-level* permission: controlling access to a specific *instance* `NCC-1701-D` of a resource `ship`. The second is also an instance-level permission that states *the ability to `edit` the `user` with identifier `jlpicard`*.

How permissions are stored in Stormpath, as well as how to customize storage and access options in Stormpath is out of scope for this document, but this is explained in the Shiro Stormpath plugin documentation.

Step 7b: Permission Tags

Just as we have JSP tags for role checks, parallel tags exist for permission checks as well. We update the `/home.jsp` page to let the user know if they're allowed to do something or not based on the permissions that are assigned to them. These messages are added in a new `<h2>Permissions</h2>` section of the home page:

```
<h2>Permissions</h2>

<ul>
    <li>You may <shiro:lacksPermission name="ship:NCC-1701-D:command"><b>NOT</b>
</shiro:lacksPermission> command the <code>NCC-1701-D</code> Starship!</li>
    <li>You may <shiro:lacksPermission name="user:${account.username}:edit"><b>NOT</b>
</shiro:lacksPermission> edit the ${account.username} user!</li>
</ul>
```

When you visit the home page the first time, before you log in, you will see the following output:

```
You may NOT command the NCC-1701-D Starship!
You may NOT edit the user!
```

But after you log in with your Jean-Luc Picard account, you will see this instead:

```
You may command the NCC-1701-D Starship!
You may edit the user!
```

You can see that Shiro resolved that the authenticated user had permissions, and the output was rendered in an appropriate way.

You can also use the `<shiro:hasPermission>` tag for affirmative permission checks.

Finally, we'll call to attention an extremely powerful feature with permission checks. Did you see how the second permission check used a *runtime* generated permission value?

```
<shiro:lacksPermission name="user:${account.username}:edit"> ...
```

The `${account.username}` value is interpreted at runtime and forms the final `user:aUsername:edit` value, and then the final String value is used for the permission check.

This is *extremely* powerful: you can perform permission checks based on who the current user is and *what is currently being interacted with*. These runtime-based instance-level permission checks are a foundational technique for developing highly customizable and secure applications.

Step 7c: Run the webapp

After checking out the `step7` branch, go ahead and run the web app:

```
$ mvn jetty:run
```

Try visiting localhost:8080 and log in and out of the UI with your Jean-Luc Picard account (and other accounts), and see the page output change based on what permissions are assigned (or not)!

Hit `ctl-C` (or `cmd-C` on a mac) to shut down the web app.

Summary

We hope you have found this introductory tutorial for Shiro-enabled webapps useful. In coming editions of this tutorial, we will cover:

- Plugging in different user data stores, like an RDBMS or NoSQL data store.

Fixes and Pull Requests

Please send any fixes for errata as a GitHub Pull Request to the `https://github.com/lhazlewood/apache-shiro-tutorial-webapp` repository. We appreciate it!!!