Spring Session - REST

Rob Winch

Version 2.3.0.RELEASE

Table of Contents

Updating Dependencies 2
Spring Configuration 3
Servlet Container Initialization 4
`rest` Sample Application 5
- Running the `rest` Sample Application 5
- Exploring the `rest` Sample Application 5
- How Does It Work? 6

This guide describes how to use Spring Session to transparently leverage Redis to back a web application's `HttpSession` when you use REST endpoints.

NOTE	You can find the completed guide in the rest sample application.

Index

Updating Dependencies

Before you use Spring Session, you must update your dependencies. If you use Maven, you must add the following dependencies:

pom.xml

```
<dependencies>
    <!-- ... -->

    <dependency>
        <groupId>org.springframework.session</groupId>
        <artifactId>spring-session-data-redis</artifactId>
        <version>2.3.0.RELEASE</version>
        <type>pom</type>
    </dependency>
    <dependency>
        <groupId>io.lettuce</groupId>
        <artifactId>lettuce-core</artifactId>
        <version>5.2.2.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
        <version>5.2.6.RELEASE</version>
    </dependency>
</dependencies>
```

Spring Configuration

After adding the required dependencies, we can create our Spring configuration. The Spring configuration is responsible for creating a servlet filter that replaces the `HttpSession` implementation with an implementation backed by Spring Session. To do so, add the following Spring Configuration:

```
@Configuration
@EnableRedisHttpSession ①
public class HttpSessionConfig {

    @Bean
    public LettuceConnectionFactory connectionFactory() {
        return new LettuceConnectionFactory(); ②
    }

    @Bean
    public HttpSessionIdResolver httpSessionIdResolver() {
        return HeaderHttpSessionIdResolver.xAuthToken(); ③
    }

}
```

① The `@EnableRedisHttpSession` annotation creates a Spring bean named `springSessionRepositoryFilter` that implements `Filter`. The filter is in charge of replacing the `HttpSession` implementation to be backed by Spring Session. In this instance, Spring Session is backed by Redis.

② We create a `RedisConnectionFactory` that connects Spring Session to the Redis Server. We configure the connection to connect to localhost on the default port (6379). For more information on configuring Spring Data Redis, see the reference documentation.

③ We customize Spring Session's HttpSession integration to use HTTP headers to convey the current session information instead of cookies.

Servlet Container Initialization

Our Spring Configuration created a Spring Bean named `springSessionRepositoryFilter` that implements `Filter`. The `springSessionRepositoryFilter` bean is responsible for replacing the `HttpSession` with a custom implementation that is backed by Spring Session.

In order for our `Filter` to do its magic, Spring needs to load our `Config` class. We provide the configuration in our Spring `MvcInitializer`, as the following example shows:

src/main/java/sample/mvc/MvcInitializer.java

```
@Override
protected Class<?>[] getRootConfigClasses() {
    return new Class[] { SecurityConfig.class, HttpSessionConfig.class };
}
```

Last, we need to ensure that our Servlet Container (that is, Tomcat) uses our `springSessionRepositoryFilter` for every request. Fortunately, Spring Session provides a utility class named `AbstractHttpSessionApplicationInitializer` that makes doing so easy. To do so, extend the class with the default constructor, as the following example shows:

src/main/java/sample/Initializer.java

```
public class Initializer extends AbstractHttpSessionApplicationInitializer {

}
```

NOTE The name of our class (`Initializer`) does not matter. What is important is that we extend `AbstractHttpSessionApplicationInitializer`.

`rest` Sample Application

This section describes how to use the `rest` sample application.

Running the `rest` Sample Application

You can run the sample by obtaining the source code and invoking the following command:

NOTE	For the sample to work, you must install Redis 2.8+ on localhost and run it with the default port (6379). Alternatively, you can update the `RedisConnectionFactory` to point to a Redis server. Another option is to use Docker to run Redis on localhost. See Docker Redis repository for detailed instructions.

```
$ ./gradlew :spring-session-sample-javaconfig-rest:tomcatRun
```

You should now be able to access the application at http://localhost:8080/

Exploring the `rest` Sample Application

You can now try to use the application. To do so, use your favorite REST client to request http://localhost:8080/

```
    $ curl -v http://localhost:8080/
```

Note that you are prompted for basic authentication. Provide the following information for the username and password:

- **Username** _user-
- **Password** *password*

Then run the following command:

```
$ curl -v http://localhost:8080/ -u user:password
```

In the output, you should notice the following:

```
HTTP/1.1 200 OK
...
X-Auth-Token: 0dc1f6e1-c7f1-41ac-8ce2-32b6b3e57aa3

{"username":"user"}
```

Specifically, you should notice the following things about our response:

- The HTTP Status is now a 200.
- We have a header a the name of `X-Auth-Token` and that contains a new session ID.
- The current username is displayed.

We can now use the `X-Auth-Token` to make another request without providing the username and password again. For example, the following command outputs the username, as before:

```
    $ curl -v http://localhost:8080/ -H "X-Auth-Token: 0dc1f6e1-c7f1-41ac-8ce2-
32b6b3e57aa3"
```

The only difference is that the session ID is not provided in the response headers because we are reusing an existing session.

If we invalidate the session, the `X-Auth-Token` is displayed in the response with an empty value. For example, the following command invalidates our session:

```
    $ curl -v http://localhost:8080/logout -H "X-Auth-Token: 0dc1f6e1-c7f1-41ac-
8ce2-32b6b3e57aa3"
```

You can see in the output that the `X-Auth-Token` provides an empty `String` indicating that the previous session was invalidated:

```
HTTP/1.1 204 No Content
...
X-Auth-Token:
```

How Does It Work?

Spring Security interacts with the standard `HttpSession` in `SecurityContextPersistenceFilter`.

Instead of using Tomcat's `HttpSession`, Spring Security is now persisting the values in Redis. Spring Session creates a header named `X-Auth-Token` in your browser. That header contains the ID of your session.

If you like, you can easily see that the session is created in Redis. To do so, create a session by using the following command:

```
$ curl -v http://localhost:8080/ -u user:password
```

In the output, you should notice the following:

```
HTTP/1.1 200 OK
...
X-Auth-Token: 7e8383a4-082c-4ffe-a4bc-c40fd3363c5e

{"username":"user"}
```

Now you can remove the session by using redis-cli. For example, on a Linux based system, you can type:

```
    $ redis-cli keys '*' | xargs redis-cli del
```

TIP The Redis documentation has instructions for installing redis-cli.

Alternatively, you can also delete the explicit key. To do so, enter the following into your terminal, being sure to replace `7e8383a4-082c-4ffe-a4bc-c40fd3363c5e` with the value of your `SESSION` cookie:

```
    $ redis-cli del spring:session:sessions:7e8383a4-082c-4ffe-a4bc-c40fd3363c5e
```

We can now use the `X-Auth-Token` to make another request with the session we deleted and observe we that are prompted for authentication. For example, the following returns an HTTP 401:

```
    $ curl -v http://localhost:8080/ -H "X-Auth-Token: 0dc1f6e1-c7f1-41ac-8ce2-
32b6b3e57aa3"
```