# Spring Session and Spring Security

Rob Winch

Version 2.3.0.RELEASE

# Table of Contents

This guide describes how to use Spring Session along with Spring Security. It assumes you have already applied Spring Security to your application.

| NOTE | You can find the completed guide in the security 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 Config {

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

}
```

① The `@EnableRedisHttpSession` annotation creates a Spring bean with the name of `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.

# 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. Since our application is already loading Spring configuration by using our `SecurityInitializer` class, we can add our configuration class to it. The following example shows how to do so:

*src/main/java/sample/SecurityInitializer.java*

```
public class SecurityInitializer extends AbstractSecurityWebApplicationInitializer
{

    public SecurityInitializer() {
        super(SecurityConfig.class, Config.class);
    }

}
```

Last, we need to ensure that our Servlet Container (that is, Tomcat) uses our `springSessionRepositoryFilter` for every request. It is extremely important that Spring Session's `springSessionRepositoryFilter` is invoked before Spring Security's `springSecurityFilterChain`. This ensures that the `HttpSession` that Spring Security uses is backed by Spring Session. Fortunately, Spring Session provides a utility class named `AbstractHttpSessionApplicationInitializer` that makes doing so easy. The following example shows how to do so:

*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`.

By extending `AbstractHttpSessionApplicationInitializer`, we ensure that the Spring bean named `springSessionRepositoryFilter` is registered with our Servlet Container for every request before Spring Security's `springSecurityFilterChain` .

# `security` Sample Application

This section describes how to work with the `security` sample application.

## Running the `security` Sample Application

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

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

| 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. |
|---|---|

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

## Exploring the `security` Sample Application

Now you can use the application. Enter the following to log in:

- **Username** *user*
- **Password** *password*

Now click the **Login** button. You should now see a message indicating your are logged in with the user entered previously. The user's information is stored in Redis rather than Tomcat's `HttpSession` implementation.

## How Does It Work?

Instead of using Tomcat's `HttpSession`, we persist the values in Redis. Spring Session replaces the `HttpSession` with an implementation that is backed by Redis. When Spring Security's `SecurityContextPersistenceFilter` saves the `SecurityContext` to the `HttpSession`, it is then persisted into Redis.

When a new `HttpSession` is created, Spring Session creates a cookie named `SESSION` in your browser. That cookie contains the ID of your session. You can view the cookies (with Chrome or Firefox).

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

```
$ 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. Enter the following command 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
```

Now you can visit the application at http://localhost:8080/ and see that we are no longer authenticated.