Spring Session - Spring Boot

Rob Winch, Vedran Pavić

Version 2.3.0.RELEASE

Table of Contents

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

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

Index

Updating Dependencies

Before you use Spring Session, you must ensure your dependencies. We assume you are working with a working Spring Boot web application. If you are using Maven, you must add the following dependencies:

pom.xml

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

    <dependency>
        <groupId>org.springframework.session</groupId>
        <artifactId>spring-session-data-redis</artifactId>
    </dependency>
</dependencies>
```

Spring Boot provides dependency management for Spring Session modules, so you need not explicitly declare dependency version.

Spring Boot Configuration

After adding the required dependencies, we can create our Spring Boot configuration. Thanks to first-class auto configuration support, setting up Spring Session backed by Redis is as simple as adding a single configuration property to your `application.properties`, as the following listing shows:

src/main/resources/application.properties

```
spring.session.store-type=redis # Session store type.
```

Under the hood, Spring Boot applies configuration that is equivalent to manually adding `@EnableRedisHttpSession` annotation. This 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.

Further customization is possible by using `application.properties`, as the following listing shows:

src/main/resources/application.properties

```
server.servlet.session.timeout= # Session timeout. If a duration suffix is not
specified, seconds is used.
spring.session.redis.flush-mode=on_save # Sessions flush mode.
spring.session.redis.namespace=spring:session # Namespace for keys used to store
sessions.
```

For more information, see the Spring Session portion of the Spring Boot documentation.

Configuring the Redis Connection

Spring Boot automatically creates a `RedisConnectionFactory` that connects Spring Session to a Redis Server on localhost on port 6379 (default port). In a production environment, you need to update your configuration to point to your Redis server. For example, you can include the following in your application.properties:

src/main/resources/application.properties

```
spring.redis.host=localhost # Redis server host.
spring.redis.password= # Login password of the redis server.
spring.redis.port=6379 # Redis server port.
```

For more information, see the Connecting to Redis portion of the Spring Boot documentation.

Servlet Container Initialization

Our Spring Boot 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. Last, we need to ensure that our servlet container (that is, Tomcat) uses our `springSessionRepositoryFilter` for every request. Fortunately, Spring Boot takes care of both of these steps for us.

Boot Sample Application

The Boot Sample Application demonstrates how to use Spring Session to transparently leverage Redis to back a web application's `HttpSession` when you use Spring Boot.

Running the Boot Sample Application

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

```
$ ./gradlew :spring-session-sample-boot-redis:bootRun
```

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 try using 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 by using redis-cli. For example, on a Linux based system you can type the following:

```
$ 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
```

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