Spring Session - WebSocket

Rob Winch

Version 2.3.0.RELEASE

Table of Contents

This guide describes how to use Spring Session to ensure that WebSocket messages keep your HttpSession alive.

NOTE	Spring Session's WebSocket support works only with Spring's WebSocket support. Specifically,it does not work with using JSR-356 directly, because JSR-356 does not have a mechanism for intercepting incoming WebSocket messages.

Index

HttpSession Setup

The first step is to integrate Spring Session with the HttpSession. These steps are already outlined in the HttpSession with Redis Guide.

Please make sure you have already integrated Spring Session with HttpSession before proceeding.

Spring Configuration

In a typical Spring WebSocket application, you would implement `WebSocketMessageBrokerConfigurer`. For example, the configuration might look something like the following:

```
@Configuration
@EnableScheduling
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/messages").withSockJS();
    }

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.enableSimpleBroker("/queue/", "/topic/");
        registry.setApplicationDestinationPrefixes("/app");
    }

}
```

We can update our configuration to use Spring Session's WebSocket support. The following example shows how to do so:

src/main/java/samples/config/WebSocketConfig.java

```
@Configuration
@EnableScheduling
@EnableWebSocketMessageBroker
public class WebSocketConfig extends
AbstractSessionWebSocketMessageBrokerConfigurer<Session> { ①

    @Override
    protected void configureStompEndpoints(StompEndpointRegistry registry) { ②
        registry.addEndpoint("/messages").withSockJS();
    }

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.enableSimpleBroker("/queue/", "/topic/");
        registry.setApplicationDestinationPrefixes("/app");
    }

}
```

To hook in the Spring Session support we only need to change two things:

① Instead of implementing `WebSocketMessageBrokerConfigurer`, we extend `AbstractSessionWebSocketMessageBrokerConfigurer`

② We rename the `registerStompEndpoints` method to `configureStompEndpoints`

What does `AbstractSessionWebSocketMessageBrokerConfigurer` do behind the scenes?

- `WebSocketConnectHandlerDecoratorFactory` is added as a `WebSocketHandlerDecoratorFactory` to `WebSocketTransportRegistration`. This ensures a custom `SessionConnectEvent` is fired that contains the `WebSocketSession`. The `WebSocketSession` is necessary to terminate any WebSocket connections that are still open when a Spring Session is terminated.
- `SessionRepositoryMessageInterceptor` is added as a `HandshakeInterceptor` to every `StompWebSocketEndpointRegistration`. This ensures that the `Session` is added to the WebSocket properties to enable updating the last accessed time.
- `SessionRepositoryMessageInterceptor` is added as a `ChannelInterceptor` to our inbound `ChannelRegistration`. This ensures that every time an inbound message is received, that the last accessed time of our Spring Session is updated.
- `WebSocketRegistryListener` is created as a Spring bean. This ensures that we have a mapping of all of the `Session` IDs to the corresponding WebSocket connections. By maintaining this mapping, we can close all the WebSocket connections when a Spring Session (HttpSession) is terminated.

`websocket` Sample Application

The `websocket` sample application demonstrates how to use Spring Session with WebSockets.

Running the `websocket` Sample Application

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

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

TIP

For the purposes of testing session expiration, you may want to change the session expiration to be 1 minute (the default is 30 minutes) by adding the following configuration property before starting the application:

src/main/resources/application.properties

```
server.servlet.session.timeout=1m # Session timeout. If a duration
suffix is not specified, seconds will be used.
```

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 `websocket` Sample Application

Now you can try using the application. Authenticate with the following information:

- **Username** *rob*
- **Password** *password*

Now click the **Login** button. You should now be authenticated as the user **rob**.

Open an incognito window and access http://localhost:8080/

You are prompted with a login form. Authenticate with the following information:

- **Username** *luke*
- **Password** *password*

Now send a message from rob to luke. The message should appear.

Wait for two minutes and try sending a message from rob to luke again. You can see that the message is no longer sent.

NOTE

Why two minutes?

Spring Session expires in 60 seconds, but the notification from Redis is not guaranteed to happen within 60 seconds. To ensure the socket is closed in a reasonable amount of time, Spring Session runs a background task every minute at 00 seconds that forcibly cleans up any expired sessions. This means you need to wait at most two minutes before the WebSocket connection is terminated.

You can now try accessing http://localhost:8080/ You are prompted to authenticate again. This demonstrates that the session properly expires.

Now repeat the same exercise, but instead of waiting two minutes, send a message from each of the users every 30 seconds. You can see that the messages continue to be sent. Try accessing http://localhost:8080/ You are not prompted to authenticate again. This demonstrates the session is kept alive.

NOTE

Only messages sent from a user keep the session alive. This is because only messages coming from a user imply user activity. Received messages do not imply activity and, thus, do not renew the session expiration.