(/)

Intro to WebSockets with Spring

Last modified: October 20, 2020

by baeldung (https://www.baeldung.com/author/baeldung/)

Spring (https://www.baeldung.com/category/spring/) +

WebSockets (https://www.baeldung.com/tag/websockets/)

1. Overview

In this article, we'll create a simple web application that implements messaging using the **new WebSocket capabilities** introduced with Spring Framework 4.0.

WebSockets is a **bi-directional**, **full-duplex**, **persistent connection** between a web browser and a server. Once a WebSocket connection is established the connection stays open until the client or server decides to close this connection.

A typical use case could be when an app involves multiple users communicating with each other, like in a chat. We will build a simple chat client in our example.

2. Maven Dependencies

Since this is a Maven-based project, we first add the required dependencies to the *pom.xml*:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-websocket</artifactId>
    <version>5.2.2.RELEASE</version>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-messaging</artifactId>
    <version>5.2.2.RELEASE</version>
</dependency>
```

In addition, as we'll use *JSON* to build the body of our messages, we need to add the *Jackson* dependencies. This allows Spring to convert our Java object to/from *JSON*:

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-core</artifactId>
    <version>2.10.2</version>
</dependency>

<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.10.2</version>
</dependency>
```

If you want to get the newest version of the libraries above, look for them on Maven Central (https://search.maven.org/classic/).

3. Enable WebSocket in Spring

The first thing to do is to enable the WebSocket capabilities. To do this we need to add a configuration to our application and annotate this class with *@EnableWebSocketMessageBroker*.

As its name suggests, it enables WebSocket message handling, backed by a message broker:

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig extends
AbstractWebSocketMessageBrokerConfigurer {

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

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

Here, we can see that the method *configureMessageBroker* is used to **configure the message broker**. First, we enable an in-memory message broker to carry the messages back to the client on destinations prefixed with "/topic".

We complete our simple configuration by designating the "/app" prefix to filter destinations targeting application annotated methods (via *@MessageMapping*).

The *registerStompEndpoints* method registers the "/chat" endpoint, enabling **Spring's STOMP (https://stomp.github.io/stomp-specification-1.2.html#Abstract) support**. Keep in mind that we are also adding here an endpoint that works without the SockJS for the sake of elasticity.

This endpoint, when prefixed with "/app", is the endpoint that the *ChatController.send()* method is mapped to handle.

It also **enables the SockJS (https://github.com/sockjs/sockjs-protocol) fallback options,** so that alternative messaging options may be used if WebSockets are not available. This is useful since WebSocket is not supported in all browsers yet and may be precluded by restrictive network proxies.

The fallbacks let the applications use a WebSocket API but gracefully degrade to non-WebSocket alternatives when necessary at runtime.

4. Create the Message Model

Now that we've set up the project and configured the WebSocket capabilities, we need to create a message to send.

The endpoint will accept messages containing the sender name and a text in a STOMP message whose body is a *JSON* object.

The message might look like this:

```
{
    "from": "John",
    "text": "Hello!"
}
```

To model the message carrying the text, we can create a simple Java object with *from* and *text* properties:

```
public class Message {

    private String from;
    private String text;

    // getters and setters
}
```

By default, Spring will use the *Jackson* library to convert our model object to and from JSON.

5. Create a Message-Handling Controller

As we've seen, Spring's approach to working with STOMP messaging is to associate a controller method to the configured endpoint. This is made possible through the *@MessageMapping* annotation.

The association between the endpoint and the controller gives us the ability to handle the message if needed:

```
@MessageMapping("/chat")
@SendTo("/topic/messages")
public OutputMessage send(Message message) throws Exception {
    String time = new SimpleDateFormat("HH:mm").format(new Date());
    return new OutputMessage(message.getFrom(), message.getText(), time);
}
```

For the purposes of our example, we'll create another model object named *OutputMessage* to represent the output message sent to the configured destination. We populate our object with the sender and the message text taken from the incoming message and enrich it with a timestamp.

After handling our message, we send it to the appropriate destination defined with the *@SendTo* annotation. All subscribers to the "*/topic/messages*" destination will receive the message.

6. Create a Browser Client

After making our configurations in the server-side, we'll use the **sockjs-client (https://github.com/sockjs/sockjs-client) library** to build a simple HTML page that interacts with our messaging system.

First of all, we need to import the *sockjs* and *stomp* Javascript client libraries. Next, we can create a *connect()* function to open the communication with our endpoint, a *sendMessage()* function to send our STOMP message and a *disconnect()* function to close the communication:

```
<html>
    <head>
        <title>Chat WebSocket</title>
        <script src="resources/js/sockjs-0.3.4.js"></script>
        <script src="resources/js/stomp.js"></script>
        <script type="text/javascript">
            var stompClient = null;

            function setConnected(connected) {
                document.getElementById('connect').disabled = connected;
                document.getElementById('disconnect').disabled =
!connected;

document.getElementById('conversationDiv').style.visibility
                  = connected ? 'visible' : 'hidden';
                document.getElementById('response').innerHTML = '';
            }

            function connect() {
                var socket = new SockJS('/chat');
                stompClient = Stomp.over(socket);
                stompClient.connect({}, function(frame) {
                    setConnected(true);
                    console.log('Connected: ' + frame);
                    stompClient.subscribe('/topic/messages',
function(messageOutput) {
                        showMessageOutput(JSON.parse(messageOutput.body));
                    });
                });
            }

            function disconnect() {
                if(stompClient != null) {
                    stompClient.disconnect();
                }
                setConnected(false);
                console.log("Disconnected");
            }

            function sendMessage() {
                var from = document.getElementById('from').value;
                var text = document.getElementById('text').value;
                stompClient.send("/app/chat", {},
                  JSON.stringify({'from':from, 'text':text}));
            }

            function showMessageOutput(messageOutput) {
                var response = document.getElementById('response');
                var p = document.createElement('p');
                p.style.wordWrap = 'break-word';
                p.appendChild(document.createTextNode(messageOutput.from +
": "
                  + messageOutput.text + " (" + messageOutput.time +
")"));
                response.appendChild(p);
            }
        </script>
    </head>
    <body onload="disconnect()">
        <div>
            <div>
                <input type="text" id="from" placeholder="Choose a
nickname"/>
            </div>
            <br />
            <div>
                <button id="connect" onclick="connect();">Connect</button>
                <button id="disconnect" disabled="disabled"
onclick="disconnect();">
                    Disconnect
                </button>
            </div>
            <br />
            <div id="conversationDiv">
                <input type="text" id="text" placeholder="Write a
message..."/>
                <button id="sendMessage"
onclick="sendMessage();">Send</button>
                <p id="response"></p>
            </div>
        </div>

    </body>
</html>
```

7. Testing the Example

To test our example, we can open a couple of browser windows and access the chat page at:

```
http://localhost:8080
```

Once this is done, we can join the chat by entering a nickname and hitting the connect button. If we compose and send a message we can see it in all browser sessions that have joined the chat.

Take a look at the screenshot to see an example:

(/wp-

content/uploads/2016/05/websockets-chat.png)

8. Conclusion

In this tutorial, we've explored Spring's WebSocket support. We've seen its server-side configuration and built a simple client-side counterpart with the use of *sockjs* and *stomp* Javascript libraries.

The example code can be found in the GitHub project (https://github.com/eugenp/tutorials/tree/master/spring-websockets).

7 COMMENTS Oldest

View Comments

Comments are closed on this article!

COURSES