How to build applications with the WebSocket API for Java EE and Jakarta EE

Digging into the specification

Configuring a WebSocket project

Creating a chat application using WebSocket

The client endpoint

WebSocket customization

Conclusion

Dig deeper

JAKARTA EE

How to build applications with the WebSocket API for Java EE and Jakarta EE

WebSocket is a two-way communication protocol that lets clients send and receive messages over a single connection to a server endpoint.

by Josh Juneau

January 29, 2021

WebSocket is a two-way communication protocol that lets clients send and receive messages over a single connection to a server endpoint. The Jakarta WebSocket API, part of the Jakarta EE platform, can be used to develop WebSocket server endpoints as well as WebSocket clients. This article provides a brief overview of the Jakarta WebSocket specification, and I'll show how to construct an application using WebSockets.

I'll cover the Jakarta WebSocket API as it stands as part of the Jakarta EE 9 platform release. That said, the examples in this article will work with Jakarta EE 8 or Java EE 7 or Java EE 8 applications. The main difference is that the namespace for Jakarta EE 9 is `jakarta.*`; in earlier releases, it was `javax.*`. Therefore, if you are using a previous release, change the namespace to `javax.*`.

WebSocket is a vendor-independent standard. If you're curious about the WebSocket protocol, it's covered in depth in IETF RFC 6455. Many tutorials are published online. You can also read the documentation for the WebSocket interface in JDK 15.

To communicate with WebSocket, you must configure a server endpoint. The simplest endpoint is a standard Java class that either is annotated with `@ServerEndpoint` or extends the `jakarta.websocket.Endpoint` abstract class.

An endpoint also contains a method that's annotated with `@OnMessage`. The `@ServerEndpoint` annotation accepts the URI at which the WebSocket server will accept messages that need to be sent. The URI can also be used to register clients as recipients for WebSocket messages.

The following simple endpoint accepts a string-based message at the endpoint URI `/basicEndpoint` and performs an activity with that message once it has been received. A client can connect to the server endpoint URI to open the connection, which will remain open for sending and receiving messages for the duration of the session.

```
@ServerEndpoint(value = "/basicEndpoint")
public class BasicEndpoint {
    @OnMessage
    public void onMessage(Session session,
                                    String mess
        // perform an action
    }
}
```

In the following sections, you'll see the wide variety of options available for developing more-sophisticated WebSocket solutions. However, the overall concept for generating a WebSocket endpoint remains very much the same as the previous example.

Digging into the specification

You can develop WebSocket endpoints using either an annotation-based or programmatic approach. You can use the `@ServerEndpoint` annotation to specify that a class is used as a WebSocket server endpoint. The alternative to using `@ServerEndpoint` is to extend the `jakarta.websocket.Endpoint` abstract class. The examples for this article use the annotation approach. Similarly, you can use the `@ClientEndpoint` annotation to specify that a standard Java class is used to accept WebSocket messages. `@ServerEndpoint` and `@ClientEndpoint` can specify the following attributes:

- `value`: Specifies a URI path at which the server endpoint will be deployed.
- `decoders`: Specifies a list of classes that can be used to decode incoming messages to the WebSocket endpoint. Classes implement the `Decoder` interface.
- `encoders`: Specifies a list of classes that can be used to encode outgoing messages from the WebSocket endpoint. Classes implement the `Encoder` interface.
- `subprotocols`: Specifies a string-based list of supported subprotocols.
- `configurator`: Lists a custom implementation of `ServerEndpointConfiguration.Configurator`.

The specification defines a number of annotations that can be placed on method declarations of a WebSocket endpoint class. Each of the annotations can be used only once per class, and they are used to decorate methods which contain implementations that are to be invoked when the corresponding WebSocket events occur. The method annotations are as follows:

- `@OnOpen`: When it is specified on a method, it will be invoked when a WebSocket connection is established. The method can optionally specify `Session` as the first parameter and `EndpointConfig` as a second parameter.
- `@OnMessage`: When it is specified on a method, it will be invoked when a message is received. The method can optionally specify `Session` as the first parameter and `String (message)` as a second parameter.
- `@OnClose`: When it is specified on a method, it will be invoked when a WebSocket connection is closed. The method can optionally specify `Session` as the first parameter and `CloseReason` as a second parameter.
- `@OnError`: When it is specified on a method, it will be invoked when an `Exception` is being thrown by any method annotated with `@OnOpen`, `@OnMessage`, or `@OnClose`. The method can optionally specify `Session` as the first parameter along with `Throwable` parameters.

Configuring a WebSocket project

To get started with Jakarta WebSocket, you must either add the `websocket-api` dependency to a project or add the `jakarta-ee` dependency to make use of the entire platform. Both the Jakarta EE full profile and the web profile contain the Jakarta WebSocket dependency.

```
<dependency>
            <groupId>jakarta.platform</groupI
            <artifactId>jakarta.jakartaee-api
            <version>${jakartaee}</version>
</dependency>
```

For projects that will contain an `@ClientEndpoint`, you must add an implementation as a dependency. In this case, I add the Tyrus client implementation by adding the following dependency. (Project Tyrus, from Oracle, is a JSR 356 Java API for WebSocket reference implementation.)

```
<dependency>
            <groupId>org.glassfish.tyrus.bund
            <artifactId>tyrus-standalone-clie
            <version>2.0.0-M3</version>
 </dependency>
```

Creating a chat application using WebSocket

Here's an application that uses WebSocket server endpoints with a JavaScript WebSocket client to send and receive messages. This particular example, called AcmeChat, uses Maven, but another build system such as Gradle would work just as well. This example will be deployed to Payara 5.202 running on Jakarta EE 9.

To follow along, you can clone the source code from GitHub.

The WebSocket endpoint. To begin, create a Maven web application and add the Jakarta EE 9 API dependency, along with any others that may be used, as shown in **Listing 1**. In this situation, you could also use the Jakarta EE Web Profile to make the application lighter.

Listing 1. Adding the Jakarta EE 9 API dependency

```
<project xmlns="http://maven.apache.org/POM/4
         xsi:schemaLocation="http://maven.apa
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.employeeevent</groupId>
    <artifactId>AcmeChat</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>
    <name>AcmeChat-1.0-SNAPSHOT</name>

    <properties>
        <maven.compiler.source>1.8</maven.com
        <maven.compiler.target>1.8</maven.com
        <endorsed.dir>${project.build.directo
        <project.build.sourceEncoding>UTF-8</
        <failOnMissingWebXml>false</failOnMis
        <jakartaee>9.0.0-RC3</jakartaee>
    </properties>

    <dependencies>
        <dependency>
            <groupId>jakarta.platform</groupI
            <artifactId>jakarta.jakartaee-api
            <version>${jakartaee}</version>
        </dependency>
        <dependency>
            <groupId>org.primefaces</groupId>
            <artifactId>primefaces</artifactI
            <version>8.0</version>
        </dependency>
        <dependency>
            <groupId>org.glassfish.tyrus.bund
            <artifactId>tyrus-standalone-clie
            <version>2.0.0-M3</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plu
                <artifactId>maven-compiler-pl
                <version>3.1</version>
```

```
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                    <compilerArguments>
                        <endorseddirs>${endor
                    </compilerArguments>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plu
                <artifactId>maven-war-plugin<
                <version>2.3</version>
                <configuration>
                    <failOnMissingWebXml>fals
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plu
                <artifactId>maven-dependency-
                <version>2.6</version>
                <executions>
                    <execution>
                        <phase>validate</phas
                        <goals>
                            <goal>copy</goal>
                        </goals>
                        <configuration>
                            <outputDirectory>
                            <silent>true</sil
                            <artifactItems>
                                <artifactItem
                                    <groupId>
                                    <artifact
                                    <version>
                                    <type>pom
                                </artifactIte
                            </artifactItems>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

Next, create the WebSocket server endpoint class named `com.employeeevent.acmechat.ChatEndpoint`. The source code for this class is shown in **Listing 2**. Annotate the class with `@ServerEndpoint` and specify a URI path of `"/chatEndpoint/{username}"` for the `value` attribute. Note the path parameter that is enclosed in curly braces at the end of the URI. This allows the endpoint to accept a parameter. In this case, I will be sending a message that's composed of a Java object. Therefore, I need to use an encoder and decoder to translate the message from the client to the server. I can specify an encoder and decoder via attributes of `@ServerEndpoint`.

Listing 2. Creating the WebSocket server endpoint class

```
package com.employeeevent.acmechat;
```

```
import jakarta.inject.Inject;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;
import java.util.concurrent.CopyOnWriteArrayS
import jakarta.websocket.EncodeException;
import jakarta.websocket.OnClose;
import jakarta.websocket.OnError;
import jakarta.websocket.OnMessage;
import jakarta.websocket.OnOpen;
import jakarta.websocket.Session;
import jakarta.websocket.server.PathParam;
import jakarta.websocket.server.ServerEndpoin

@ServerEndpoint(value = "/chatEndpoint/{usern
        encoders = {MessageEncoder.class},
        decoders = {MessageDecoder.class})
public class ChatEndpoint {

    @Inject
    ChatSessionController chatSessionControll

    private static Session session;
    private static Set<Session> chatters = ne

    @OnOpen
    public void messageOpen(Session session,
            @PathParam("username") String use
            EncodeException {
        this.session = session;
        Map<String,String> chatusers = chatSe
        chatusers.put(session.getId(), userna
        chatSessionController.setUsers(chatus
        chatters.add(session);
        Message message = new Message();
        message.setUsername(username);
        message.setMessage("Welcome " + usern
        broadcast(message);
    }

    @OnMessage
    public void messageReceiver(Session sessi
            Message message) throws IOExcepti
        Map<String,String> chatusers = chatSe
        message.setUsername(chatusers.get(ses
        broadcast(message);
    }

    @OnClose
    public void close(Session session) {
        chatters.remove(session);
        Message message = new Message();
        Map<String,String> chatusers = chatSe
        String chatuser = chatusers.get(sessi
        message.setUsername(chatuser);
        chatusers.remove(chatuser);
        message.setMessage("Disconnected from

    }

    @OnError
    public void onError(Session session, Thro
        System.out.println("There has been an
    }
```

```
    private static void broadcast(Message mes
            throws IOException, EncodeExcepti

        chatters.forEach(session -> {
            synchronized (session) {
                try {
                    session.getBasicRemote().
                            sendObject(messag
                } catch (IOException | Encode
                    e.printStackTrace();
                }
            }
        });
    }

}
```

Then, the endpoint class declares a field, identified as `session`, that's used to hold the WebSocket `Session` and another `Set<Session>`, identified as `chatters`, to hold each of the connected chat user sessions. The class also injects an `@ApplicationScoped` controller class entitled `ChatSessionController` for storing users in a simple `HashMap`, which is shown in **Listing 3**.

Listing 3. Endpoint class declaring fields to hold the WebSocket session and chat user sessions

```
@Named
@ApplicationScoped
public class ChatSessionController implements

    private Map<String, String> users = null;

    public ChatSessionController(){}

    @PostConstruct
    public void init(){
         users = new HashMap<>();
    }

    /**
     * @return the users
     */
    public Map<String, String> getUsers() {
        return users;
    }

    /**
     * @param for the users
     */
    public void setUsers(Map<String, String>
        this.users = users;
    }

}
```

The `ChatEndpoint` class declares four methods for handling the WebSocket server events and a method named

`broadcast()` that's used to broadcast messages to each of the connected clients, all of which are described below:

```
private static void broadcast(Message message
            throws IOException, EncodeExcepti

        chatters.forEach(session -> {
            synchronized (session) {
                try {
                    session.getBasicRemote().
                            sendObject(messag
                } catch (IOException | Encode
                    e.printStackTrace();
                }
            }
        });
}
```

- The `broadcast()` method is private and static, and it accepts a `Message` object. The method simply traverses the set of chat sessions, stored within the `chatters` field, and within a synchronized block calls upon the `getBasicRemote().sendObject()` method for each session, sending the `Message` object.
- The `messageOpen()` method, annotated with `@OnOpen`, is executed when the connection is opened. The method accepts a `Session` and an `@PathParam` string, which accepts the `username` substitute variable that's contained within the `@ServerEndpoint value` attribute. Next, the `Session` and `username` are both stored, and a `Message` object is constructed using the `username` and `message` text, and finally the message is broadcast via the invocation of the `broadcast()` method.
- The `messageReceiver()` method, annotated with `@OnMessage`, is executed when the WebSocket message is received. The method accepts a `Session` and `Message`; it uses the `ChatSessionController` to obtain the `username` of the user associated with the session and stores it in the `Message` object. The message is then broadcast by passing the `Message` to the `broadcast()` method.
- The `close()` method, annotated with `@OnClose`, is invoked when the connection is closed. This method accepts a `Session`, which is then removed from the `Set` of `chatters`, as well as the `chatusers Map`. The session is then used to obtain the corresponding username from the `ChatSessionController`, and it is stored in a new `Message` object, which is subsequently broadcast to alert the other chatters that the user has disconnected.
- The `onError()` method, annotated with `@OnError`, is invoked whenever one of the other annotated methods throws an exception. This WebSocket endpoint can accept messages from any WebSocket client, as long as the client has an active session with the endpoint. To communicate with the endpoint, the client will connect to the following URI:

`ws://<hostname>:<port>/AcmeChat/chatEndpoint`.

The WebSocket client. You can write a client in a variety of languages and still have the ability to communicate with the WebSocket endpoint. In this example, I wrote the client in JavaScript and invoked it via a Jakarta Server Faces front end.

Look at **Listing 4**, which contains the source code for the client. Note that the body of the client is written in Jakarta Server Faces and uses PrimeFaces components for the user interface. The user interface contains an `inputText` field for the `username`, an `inputTextarea` for the `message`, and two `commandButton` widgets.

One of the `commandButton` widgets invokes a JavaScript function named `chatRelay()`, which opens a connection to the WebSocket. The other button invokes a JavaScript function named `send()` to send the message from the `inputTextarea` to the WebSocket endpoint.

Listing 4. Source code for the client

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:p="http://primefaces.org/ui"
      xmlns:f="http://xmlns.jcp.org/jsf/core"
    <h:head>
        <script type="text/javascript">
                var ws;
                function chatRelay()
                {
                    var username = document.g

                    if ("WebSocket" in window
                    {
                        var json = {
                            'username': usern
                            'message': ""
                        };

                        // Open WebSocket
                        ws = new WebSocket("w
                        ws.onopen = function
                        {
                            // Perform handli
                        };
                        ws.onmessage = functi
                        {
                            var json = JSON.p
                            var currentValue
                            document.getEleme
                                    currentVa
                                    '<br />'
                                    json.user

                        };

                        ws.onclose = function
                        {
```

```
                        // websocket is c
                        alert("Connection
                    };

                } else
                {
                    // The browser doesn'
                    alert("WebSocket NOT
                }
            }

            function send() {
                var username = document.getEl
                var message = document.getEle
                var json = {
                    'username': username.valu
                    'message': message.value
                };
                ws.send(JSON.stringify(json))
                return false;
            }
        </script>
    </h:head>
    <h:body>
        <h:form id="chatForm">
            <h:outputLabel for="username" val
            <p:inputText id="username" />
            <br/>
            <p:commandButton id="wsRelay" typ
                             onclick="chatRel
            <br/><br/>

            <p:inputTextarea id="chatText" co
            <br/><br/>

            <p:commandButton id="sendMessage"
                             style="visibilit
                             onclick="send();
        </h:form>
        <br/><br/>
        <div id="output"></div>
    </h:body>

</html>
```

To open a connection to the endpoint, the `chatRelay()` function accepts the `username` from the client. Next, it checks to ensure that the client's browser will work with WebSockets and, if it won't, a message is presented on the client. If the browser is compatible with WebSockets, a new JSON object is created, passing the username and message text. The WebSocket is then opened by passing the URI to the WebSocket endpoint and appending the username to be passed in as a path parameter, for example:

```
ws = new
WebSocket("ws://localhost:8080/AcmeChat/chatEndpoint/"
+ username.value);
```

At this point, the WebSocket client is listening for responses from the server, and there are callback functions that await the server responses. The `ws.onopen` function, shown below, is

invoked when the connection is opened, invoking any handling code that may be present:

```
ws.onopen = function ()
        {
  // Perform handling
        };
```

The `ws.onmessage` function, shown below, accepts an `event` parameter. The event is the message that has been received from the server endpoint. In this case, I used the JavaScript JSON API to parse the data and populate the chat screen with the incoming message text.

```
ws.onmessage = function (evt)
        {
            var json = JSON.parse(evt.data);
            var currentValue = document.getEl
            document.getElementById('output')
                    currentValue +
                    '<br />' +
                    json.username + ": " + js

        };
```

The `ws.onclose` function, shown below, is invoked when the WebSocket server connection is disconnected, performing any processing code, as required. An example would be a case where the network connection was lost or the WebSocket endpoint was shut down. In such a case, the client could be alerted that the connection was closed.

```
ws.onclose = function ()
        {
            // websocket is closed.
            alert("Connection is closed...");
        };
```

Once the client session has been started and the WebSocket client is listening, any messages received from the WebSocket endpoint will be published via the `ws.onmessage` handler. The JavaScript `send()` function, shown below, is then used to send any messages that the user types into the `inputTextarea` to the server endpoint for broadcasting to any listening clients. The `send()` function creates a JSON object from the client username and message and sends it to the endpoint using the `ws.send` function, along with a little help from the `JSON.stringify` utility to help parse the JSON.

```
function send() {
    var username = document.getElementById('c
    var message = document.getElementById('ch
```

```
    var json = {
        'username': username.value,
        'message': message.value
    };
    ws.send(JSON.stringify(json));
    return false;
}
```

Using this client configuration, two or more different clients can connect to the same WebSocket endpoint and communicate with each other in chat-room style.

The decoder and encoder. When the JavaScript client sends a message to the endpoint, it is in JSON format. The WebSocket endpoint accepts a plain old Java object named `Message`, which contains the username and message. The decoder and encoder classes transform the client-side messages to the server-side message object, and vice versa. The Jakarta WebSocket API makes it easy to develop decoders and encoders by simply implementing the `Decoder` or `Encoder` interfaces, respectively.

Listing 5 shows the `Decoder` class implementation, which is named `MessageDecoder`. This class decodes the client-side message into a `Message` object for processing by the WebSocket server. The interface uses generics to implement the decoder for the accepted Java object. The class overrides four methods: `init()`, `willDecode()`, `decode()`, and `destroy()`.

Much like the WebSocket endpoint, the decoder is very much event-based. The `init()` method accepts an `EndpointConfig` object, and it is invoked when the message is sent from the client to the endpoint. The `willDecode()` method, which accepts a string-based message, is invoked next to return a boolean indicating whether the incoming message is in the correct format. If the message is in the correct format, the `decode()` method is invoked, again accepting a string-based message in JSON format, and the message is decoded into the `Message` object for processing via the endpoint. Lastly, the `destroy()` method is invoked when the client session becomes invalid.

Listing 5. The Decoder class implementation

```
package com.employeeevent.acmechat;

import java.io.StringReader;
import jakarta.json.Json;
import jakarta.json.JsonObject;
import jakarta.websocket.DecodeException;
import jakarta.websocket.Decoder;
import jakarta.websocket.EndpointConfig;

public class MessageDecoder implements Decode

  @Override
  public Message decode(String jsonMessage) t
```

```
    JsonObject jsonObject = Json
        .createReader(new StringReader(jsonMe
    Message message = new Message();
    message.setUsername(jsonObject.getString(
    message.setMessage(jsonObject.getString("
    return message;

  }

  @Override
  public boolean willDecode(String jsonMessag
    try {
      // Check if incoming message is valid J
      Json.createReader(new StringReader(json
      return true;
    } catch (Exception e) {
      return false;
    }
  }

  @Override
  public void init(EndpointConfig ec) {
    System.out.println("Initializing message
  }

  @Override
  public void destroy() {
    System.out.println("Destroyed message dec
  }

}
```

Listing 6 shows the `Encoder` class implementation, which is named `MessageEncoder`. This class encodes the server-side `Message` object to a `JsonObject` to be passed back to the client for processing. The interface uses generics to implement the encoder for the accepted Java object.

The class then overrides three methods: `init()`, `encode()`, and `destroy()`. Again, much like the WebSocket endpoint, the encoder is very much event-based in that the `init()` method accepts an `EndpointConfig` object, and it's initiated once for each client session that is opened. The `encode()` method accepts the object being encoded, in this case `Message`, and performs processing to translate that object into JSON before it's sent back to the client. Lastly, the `destroy()` method is invoked when the client session becomes invalid.

Listing 6. The Encoder class implementation

```
package com.employeeevent.acmechat;

import jakarta.json.Json;
import jakarta.json.JsonObject;
import jakarta.websocket.EncodeException;
import jakarta.websocket.Encoder;
import jakarta.websocket.EndpointConfig;

public class MessageEncoder implements Encode
```

```
  @Override
  public String encode(Message message) throw

    JsonObject jsonObject = Json.createObject
        .add("username", message.getUsername(
        .add("message", message.getMessage())
    return jsonObject.toString();

  }

  @Override
  public void init(EndpointConfig ec) {
    System.out.println("Initializing message
  }

    @Override
    public void destroy() {
        System.out.println("Destroying encode
    }

}
```

The client endpoint

You can develop a client endpoint to communicate with a WebSocket server endpoint. The simplest client endpoint is a standard Java class that is annotated with `@ClientEndpoint`. You can see the full source code of a `ClientEndpoint` example in **Listing 7**.

Listing 7. Code for a client endpoint

```
@ClientEndpoint
public class BasicClient {

    Session session = null;
    private MessageHandler handler;

    public BasicClient(URI endpointURI) {
        try {
            WebSocketContainer container = Co
            container.connectToServer(this, e
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    @OnOpen
    public void onOpen(Session session){
        this.session = session;
        try {
        session.getBasicRemote().sendText("Op
        } catch (IOException ex){
            System.out.println(ex);
        }
    }

    public void addMessageHandler(MessageHand
        this.handler = msgHandler;
    }
```

```
    @OnMessage
    public void processMessage(String message
        System.out.println("Received message
    }

    public void sendMessage(String message) {
        try {
            this.session.getBasicRemote().sen
        } catch (IOException ex) {
            Logger.getLogger(BasicClient.clas
        }
    }


     public static interface MessageHandler {

        public void handleMessage(String mess
    }
```

In this example, the `ClientEndpoint` is named `BasicClient`. A `Session` and `MessageHandler` are declared within the class, and the constructor accepts a URI. Upon instantiation via the constructor, a `ContainerProvider.getWebsocketContainer()` is called to obtain a `WebsocketContainer` instance identified as `container`. The `container.connectToServer()` method is then invoked, passing the endpoint URI to instantiate the client connection.

The client contains a method named `onOpen()`, which is annotated with `@OnOpen`, and accepts a `Session`. This method is invoked when the `ClientEndpoint` connection is open, and it sets the `session` and then calls upon the `getBasicRemote().sendText()` method to send a message to the client to indicate the connection is open.

The client also contains a method named `processMessage()`, annotated with `@OnMessage`, which accepts a string. This method is called upon when a message is received from the `ServerEndpoint`. The client `sendMessage()` method also accepts a string, and it calls upon the `session.getBasicRemote().sendText()` method to send the message to the `ServerEndpoint`.

This particular example also contains an internal `MessageHandler` interface and an `addMessageHandler()` method, which are used to send the messages from the client. You can use the following code to work with the client:

```
// open websocket
final BasicClient clientEndPoint = new BasicC
        new URI("ws://localhost:8080/AcmeChat
// add listener
clientEndPoint.addMessageHandler(new BasicCli
    public void handleMessage(String message)
        System.out.println(message);
    }
```

```
});
// send message to websocket
clientEndPoint.sendMessage("Message sent from
```

WebSocket customization

Sometimes you have a requirement to develop custom implementations, such as client/server handshake policies or state processing. For such cases, the `ServerEndpointConfig.Configurator` provides an option allowing you to create your own implementation. You can implement the following methods to provide customized configurations:

- `getNegotiatedSubProtocol(List<String> supported, List<String> requested)`
  : Allows a customized algorithm to determine the selection of the subprotocol that's used
- `getNegotiatedExtensions(List<Extension> installed, List<Extension> requested)`
  : Allows a customized algorithm to determine the selection of the extensions that are used
- `checkOrigin(String originHeaderValue)`: Allows the specification of an origin-checking algorithm
- `modifyHandshake(ServerEndpointConfig sec, HandshakeRequest req, HandshakeResponse res)`
  : Allows for modification of the handshake response that's sent back to the client
- `getEndpointInstance(Class<T> endpointClass)`: Allows a customized implementation for the creation of an `Endpoint` instance

The same holds true for the `ClientEndpoint.Configurator`, in that the configurator allows for customization of some algorithms during the connection initialization phase. You can customize the configuration using these two methods:

- `beforeRequest(Map<String, List<String>> headers)`
  : Allows for the modification of headers before a request is sent
- `afterResponse(HandshakeResponse res)`: Allows for the customization of the processing for a handshake response

Conclusion

The Jakarta WebSocket API provides a means for developing server-side endpoints to process and broadcast messages, as well as client endpoints for sending and receiving messages. Using the API, it's possible to handle textual or binary messages and translate them to Java objects for processing. Moreover,

since the API is part of Jakarta EE, you can code against a standard API, allowing you to customize an implementation based on individual requirements.

There are plenty of great examples on the web for developing WebSocket applications in different ways. As stated previously, you can share code between the various versions of Java EE and Jakarta EE by simply ensuring that you use the correct namespace. Use the links below to learn more about WebSockets and to download the examples for this article.

Dig deeper

- Code examples for this article
- Jakarta WebSocket API, which is part of the Jakarta EE platform
- WebSocket interface in JDK 15
- JSR 356, Java API for WebSocket
- The Java EE 7 tutorial's section on Java API for WebSocket
- Payara Platform Community Edition
- Reactive streams programming over WebSockets with Helidon SE
- Transition from Java EE to Jakarta EE

Josh Juneau

Josh Juneau (@javajuneau) works as an application developer, system analyst, and database administrator. He primarily develops using Java and other JVM languages. He is a frequent contributor to Oracle Technology Network and *Java Magazine* and has written several books for Apress about Java and Java EE. Juneau was a JCP Expert Group member for JSR 372 and JSR 378. He is a member of the NetBeans Dream Team, a Java Champion, leader for the CJUG OSS Initiative, and a regular voice on the *JavaPubHouse Off Heap* podcast.

Share this Page