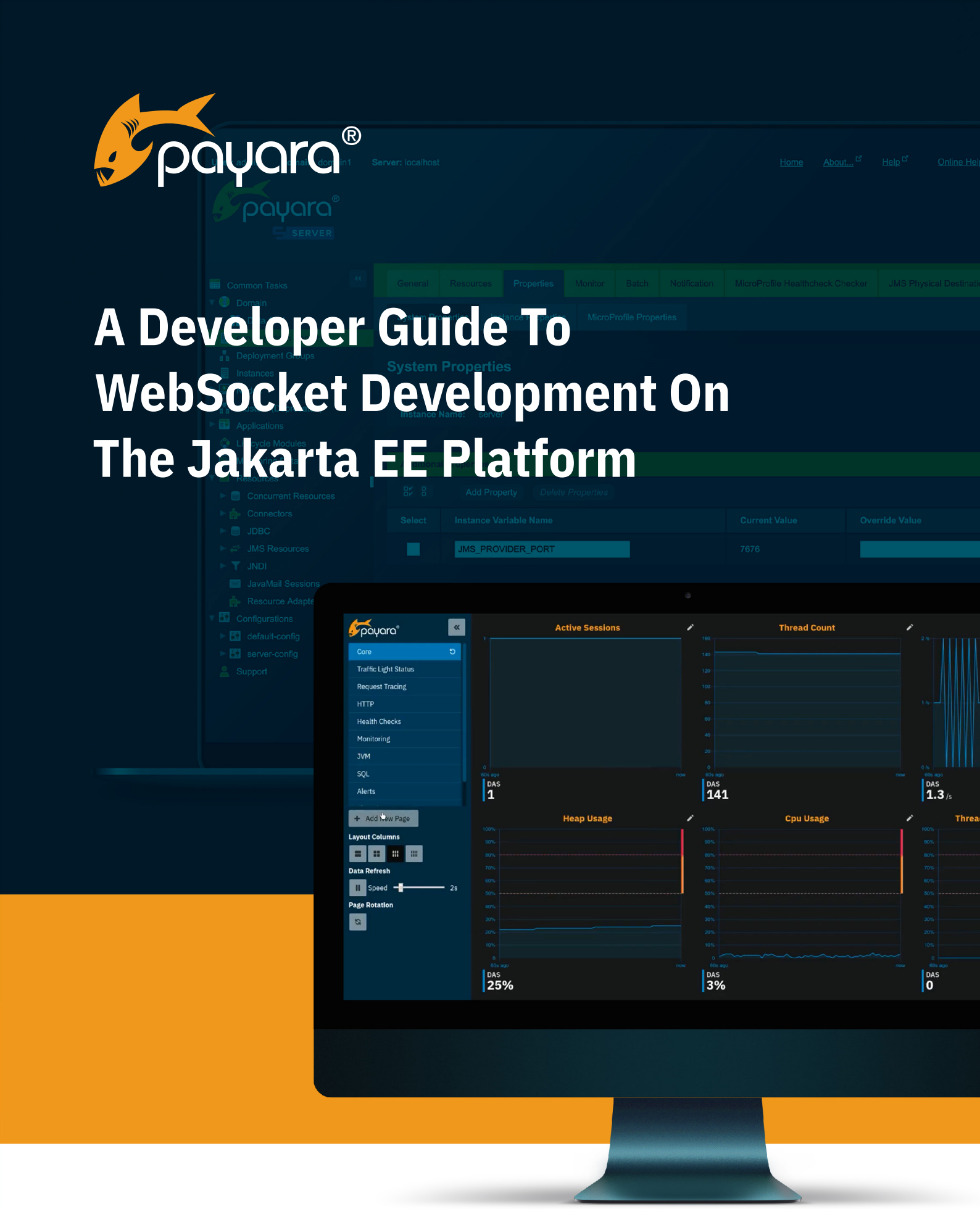
A Developer Guide To WebSocket Development On The Jakarta EE Platform



The Payara® Platform - Production-Ready, Cloud Native and Aggressively Compatible.

User Guide

Contents

Digital transformation, largely enabled by the commoditization of massive compute resources by cloud computing vendors has changed the way applications are developed, deployed and maintained. It has allowed for much faster application development iterations by organisations to meet ever changing and increasingly complex customer expectations.

The customised, real-time digital experiences pioneered by the likes of Amazon and Netflix has created a defacto standard of customer experience that all organisations aspire to to acquire and keep customers. One technology that can enhance the customer experience in applications is WebSockets.

WebSocket is a HTTP communication protocol that allows real-time full duplex interaction between a set of clients and a given server. A single WebSocket server can accept N number of client connections. Both the clients and the server can independently send messages to each other at any time during the connection. WebSocket allows the pushing of updates to connected clients without the clients having to explicitly poll the server. This way, web socket applications deliver enhanced digital experiences to users.

This guide looks at how to develop WebSocket applications on the Jakarta EE Platform using the WebSocket Specification. It starts by looking at the two main ways of creating WebSocket endpoints, then continues to show how to create endpoints using both approaches. It then proceeds to show the three kinds of messages supported out of the box and how to create custom message types using encoders and decoders. The guide then shows how to configure your WebSocket application and finally a brief look at the WebSocket client API. By the end of this guide, you will be able to create WebSocket applications on the Jakarta EE Platform using the WebSockets API.

Jakarta EE Websocket

There are two main ways of creating WebSocket endpoints using the Websocket API on the Jakarta EE Platform. One is programmatically by extending an abstract class and implementing its methods. The other is through annotations to describe the endpoint and callback methods. Let us start with the programmatic WebSocket endpoint declaration.

Programmatic WebSocket Endpoints

Creating a programmatic WebSocket endpoint entails extending the *jakarta.websocket.Endpoint* and implementing the *onOpen* abstract method as shown below.

```
public class ProgrammaticSockets extends Endpoint {

    @Override
    public void onOpen(final Session session, final EndpointConfig config)
{

    }

    @Override
    public void onClose(final Session session, final CloseReason
closeReason) {
        System.err.println("Closing: " + closeReason.getReasonPhrase());
    }

    @Override
    public void onError(final Session session, final Throwable thr) {
        System.err.println("Error: " + thr.getLocalizedMessage());
    }
}
```

The ProgrammaticSockets class extends the *Endpoint* abstract class and implements the methods *onOpen*, *onClose* and *onError* (we discuss the other two later). Of these three, only the *onOpen* method must be implemented by your code. The *onOpen* method is called when a WebSocket connection is established with a client. The method is passed a *jakarta.websocket.Session* and *jakarta.websocket.EndpointConfig* objects.

The *Session* object has methods about the remote connecting client. One of these methods allows registration of interest in receiving messages sent by the remote client. The code below shows registration of interest in receiving text messages sent by the remote connecting client within the onOpen method using the passed *Session* object.

```
session.addMessageHandler((MessageHandler.Whole<String>) message -> {
    try {
//Forward to single, current session
session.getBasicRemote().sendText(message);

    } catch (final IOException ex) {
        Logger.getLogger(ProgrammaticSockets.class.getName()).log(Level.SEVERE,
null, ex);
    }
});
```

The *addMessageHandler* method of the Session object takes a *jakarta.websocket.MessageHandler* interface object. As shown above, we use the lambda function to shorten the code. The passed in message variable is the message sent by the client. The sub interface *MessageHandler*.Whole is parametrized to take a type. In the above example, the type String is passed in, which means this code is registering its interest in receiving basic text or String messages sent by the remote client. The other types of messages that can be registered for are *java.nio.ByteBuffer* and *jakarta.websocket.PongMessage*.

Annotation Endpoints

The second way you can create WebSocket endpoints is through annotations. You annotate a Java class with the respective annotations, and the WebSocket implementation automatically picks it up. The code below shows the creation of a WebSocket endpoint hosted at the path "/annotated".

```
@ServerEndpoint(value = "/annotated")
public class AnnotatedSockets {

    @OnOpen
    public void onOpen(final Session session) {

    }

    @OnMessage
    public void handleTextMessage(final Session session, final String message)
{

    }

    @OnMessage
    public void handleBinaryMessage(final Session session, final ByteBuffer
message) {

    }

    @OnMessage
    public void handlePongMessage(final Session session, final PongMessage
message) {

    }
}
```

The AnnotatedSockets class is a Java class annotated *@ServerEndpoint(value = "/annotated")*. The parameter passed to the annotation is the endpoint at which this WebSocket will be hosted. The method *onOpen* is annotated with *@OnOpen*. This method is equivalent to the onOpen method of the programmatic endpoint class looked at in the previous section. This one however, takes only the Session as the parameter. The other three methods each have the *@OnMessage* annotation. This annotation marks each method as a callback listener when the respective message type is received.

Sending Messages

The server can send, or push messages to the connected client by calling the sendXXX method on the *jakarta.websocket.RemoteEndpoint.Basic* object returned by calling the *getBasicRemote* method on the *Session*. The code below shows sending of text and binary messages to the remote client.

```
@OnMessage
public void handleTextMessage(final Session session, final String message) {

        try {

        session.getBasicRemote().sendText(message);

    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}

@OnMessage
public void handleBinaryMessage(final Session session, final ByteBuffer
message) {
    try {
        session.getBasicRemote().sendBinary(message);
    } catch (final IOException ex) {
        Logger.getLogger(ProgrammaticSockets.class.getName()).log(Level.SEVERE,
null, ex);
    }
}
```

Sending Messages Asynchronously

The above calls to *sendXXX* are both blocking calls. This means execution will stall until they return. The *Session* interface has the *getAsyncRemote* method that returns a *RemoteEndpoint.Asyc* object. The Async object has the same *sendXXX* methods for sending the same types of messages. The code below shows sending text and binary messages asynchronously.

```
@OnMessage
    public void handleTextMessage(final Session session, final String message)
{

        session.getAsyncRemote().sendText(message);

    }

    @OnMessage
    public void handleBinaryMessage(final Session session, final ByteBuffer
message) {
        session.getAsyncRemote().sendBinary(message);
    }
```

Custom Message Types

By default, there are three kinds of messages that can be sent to and received from WebSocket clients from a Jakarta EE application. They are text, binary and *PongMessage*. In a real world production application, these three may not suffice. You can create your own custom message types to be sent and received from the client through encoders and decoders.

Encoders

An encoder is a class that takes a Java type and converts it to either plain text, a text stream, a binary or binary stream to be sent to a WebSocket client. Through encoders, you can create custom message types that can be sent to and from connecting clients. To create a WebSocket encoder, you extend the *jakarta.websocket.Encoder* interface, specifying the sub interface corresponding to one of the message types. The code below shows a text encoder for custom Java class.

```
public class CustomTextEncoder implements Encoder.Text<BusinessData> {
    private Jsonb jsonb;

    @Override
    public void init(final EndpointConfig config) {

        if (jsonb == null) {
            try (final Jsonb json = JsonbBuilder.create()) {
                jsonb = json;
            } catch (final Exception ex) {
                Logger.getLogger(CustomTextEncoder.class.getName()).log(Level.
SEVERE, null, ex);

            }

        }
    }

    @Override
    public String encode(final BusinessData object) throws EncodeException {
        if (object != null) {
            return jsonb.toJson(object);
        }
        return null;
    }
}
```

The *CustomTextEncoder* class implements the *Encoder.Text* interface, signalling that this encoder will take the passed Java type, in this case *BusinessData* and convert it to a string. The WebSocket runtime doesn't care about the actual implementation. The *Encoder* interface has the encode method which must be implemented. In the above sample, we use the Jakarta JSON Binding API to convert the passed instance of the BusinessData in the encode method to a json string. The init method has a default implementation that you can override should you have any custom initialisation to carry out. In the above example, we create the Jsonb object to be used to convert the object to a json string. The *BusinessData* Java class is shown below with getters and setters omitted for brevity.

```
public class BusinessData implements Serializable {
    private String messageId;
    private BigDecimal contractAmount;
    private String clientId;
    private LocalDate contractValidFrom;
}
```

The *Encoder* interface as stated earlier, has sub interfaces for encoding text stream, binary and binary stream. The only difference between them is the return type of the encode methods. Each one returns the corresponding type.

Registering Encoders

With the encoder in place, we need to register it with the runtime. There are two ways of registering encoders, depending on how you create the WebSocket endpoint. If you use annotations to create the WebSocket, then the *@ServerEndpoint* annotation has the encoders parameter, which takes an array of encoder classes. The code below shows registering the CustomTextEncoder implementation discussed above in the *@ServerEndpoint* annotation.

```
@ServerEndpoint(value = "/annotated-custom",
encoders = { CustomTextEncoder.class }
)
```

We discuss registering encoders and decoders programmatically in a later section when we talk about programmatic configuration. For now let us continue to look at how to create decoders.

Decoders

A decoder is a class that takes the message passed from the WebSocket client and converts it to a Java type. A decoder is the opposite of an encoder. To create a decoder, implement the *jakarta.websocket.Decoder* interface, specifying the specific sub interface that corresponds to the message type to decode. The code below shows a text decoder that does the opposite of the encoder - take a string and converting it to a Java type.

```
public class CustomTextDecoder implements Decoder.Text<BusinessData> {

    private Jsonb jsonb;

    private BusinessData businessData;
    @Override
    public void init(EndpointConfig config) {
        if (jsonb == null) {
            try (final Jsonb json = JsonbBuilder.create()) {
                jsonb = json;
            } catch (final Exception ex) {
                Logger.getLogger(CustomTextEncoder.class.getName()).log(Level.
SEVERE, null, ex);

            }

        }
    }

    @Override
    public BusinessData decode(final String s) throws DecodeException {
        return businessData;
    }

    @Override
    public boolean willDecode(final String s) {
        if (s != null && !s.isBlank()) {
            try {
                businessData = jsonb.fromJson(s, BusinessData.class);
                return true;
            } catch (final Exception ex) {
                Logger.getLogger(CustomTextDecoder.class.getName()).log(Level.
SEVERE, null, ex);

            }
        }
        return false;
    }
}
```

CustomTextDecoder implements the *Decoder.Text* interface, signalling it takes a text string and converts it to the *BusinessData* java type. The decoder has an extra boolean method - *willDecode*. This method first tests if this decoder can decode the specified type. The above implementation checks if the passed string can indeed be converted back from JSON through the Jakarta JSON-B API. It returns the right response based on how the conversion form string, which we expect to be JSON as that is the message format the server sends to the client.The decode method simply returns the already converted method. This decoder will not be put to service if the *willDecode* method returns false.

Registering Decoders

Like encoders, decoders must be registered with the WebSocket runtime. For annotation endpoints, the *@ServerEndpoint* annotation has the decoders parameter that takes an array of decoder implementations. The code below shows registration of the above implemented decoder through the annotation.

```
@ServerEndpoint(value = "/annotated-custom",
        decoders = { CustomTextDecoder.class })
```

Configuration

A WebSocket application on the Jakarta EE Platform can be configured through an implementation of the *jakarta.websocket.server.ServerApplicationConfig* interface. This interface has two methods for configuring programmatic endpoints and for deciding which annotation endpoint to be deployed. The code below shows an implementation that maps the *ProgrammaticSockets* looked at earlier to the endpoint “/programmatic” and registers the *CustomTextEncoder* and *CustomTextDecoder* classes as well.

```
public class WebsocketsConfig implements ServerApplicationConfig {
    @Override
    public Set<ServerEndpointConfig> getEndpointConfigs(Set<Class<? extends
Endpoint>> endpointClasses) {
        final var serverEndpointConfig =
                ServerEndpointConfig.Builder
                        .create(ProgrammaticSockets.class, "/programmatic")
                        .decoders(List.of(CustomTextDecoder.class))
                        .encoders(List.of(CustomTextEncoder.class))
                        .build();
        return Set.of(serverEndpointConfig);
    }

    @Override
    public Set<Class<?>> getAnnotatedEndpointClasses(Set<Class<?>> scanned) {
        return scanned;
    }
}
```

The *getEndpointConfigs* method takes a set of Endpoint classes and returns a set of *ServerEndpointConfig*. Within the method, we use the builder on the *ServerEndpointConfig* to map the programmatic WebSocket endpoint, register the decoder and encoder and finally build it. We then return the build config as a set. The *getAnnotatedEndpointClasses* method takes a set of classes that are all the annotated endpoint classes in the archive containing the implementation of this interface. In the above implementation, we simply return the passed set because we want all annotated endpoints deployed.

Handling Errors

Errors and exceptions are normal parts of software engineering. They're inevitable in almost every application. The WebSocket API provides an effortless way to handle errors that occur. Depending on how you implement your endpoints, you can either use the annotation or programmatic routes to handle errors. The following code shows how you can catch and handle errors in an annotated endpoint.

```
@OnError
public void handleError(final Session session, final Throwable throwable) {
System.err.println("Error: " + thr.getLocalizedMessage());

}
```

The *handleError* method takes a Session and a *java.lang.Throwable* object. It is annotated *@OnError*, meaning it is called if there is any exception thrown during the client server interaction. The actual implementation of the method is business specific. The WebSocket API gives you the callback to use as per your application requirement. The following code shows error handling in a programmatic endpoint.

```
@Override
public void onError(final Session session, final Throwable thr) {
    System.err.println("Error: " + thr.getLocalizedMessage());
}
```

Programmatic endpoints can override the *onError* method of the extended Endpoint abstract class. The method gets passed the same parameters - a *Session* and *Throwable* objects. Again, the actual implementation of the error handling is business specific. THe above snippets simply log the errors.

WebSocket Client

A WebSocket client is a Java class that is annotated with the *@ClientEndpoint* that declares WebSocket lifecycle methods. You can think of the client as the other end of the server endpoints we have seen so far. The code below shows a simple WebSocket client that sends the message sent from the server back.

```
@ClientEndpoint
public class CustomWebsocketClient{

    @OnMessage
    public void handleTextMessage(final Session session, final String message)
{

        try {
            session.getBasicRemote().sendText(message);

         } catch (IOException e) {
             throw new RuntimeException(e);
         }
    }

}
```

CustomWebsocketClient is annotated *@ClientEndpoint* and has a method called *handleTextMessage* that is annotated on message. When this client is deployed and connected to a server, this method will be called and passed any text message the server sends.

Connecting To WebSocket

Connecting to a WebSocket server in Jakarta EE is very straightforward through the *jakarta.websocket.WebSocketContainer*. The code below shows how to connect to the /annotated endpoint path.

```
WebSocketContainer webSocketContainer = ContainerProvider.
getWebSocketContainer();
        String uri =  "ws://localhost:8080/websocket/annotated";
        try {
             webSocketContainer.connectToServer(CustomWebsocketClient.class,
URI.create(uri));

        } catch (Exception e) {
            throw new RuntimeException(e);
        }
```

The *jakarta.websocket.ContainerProvider* has the *getWebSocketContainer* that returns a *jakarta.websocket.WebSocketContainer* object. We then create the URI to the WebSocket endpoint and pass it together with the client discussed earlier to the *connectToServer* method. This simple call sets up a connection between the client and the deployed endpoint. From then on, the communication between the two is bidirectional and independent. So the server can push to the client, and the client can push to the server.

Summary

This guide has introduced you to WebSocket development on the Jakarta EE Platform. We started by looking at the two kinds of endpoints, then looked at sending messages and creating custom message types through encoders and decoders. We then looked at configuration, handling errors, WebSocket clients and how to connect to a WebSocket. It is my hope that through this quick guide, you have the basis to explore further the full features of the WebSocket API. The specification is quite easy to read and is one of the smallest specs on the platform. Look at it and start building rich client experiences through the power of WebSockets!

If you enjoyed this guide, you might find these useful:

- The Complete Guide To JSON Processing On the Jakarta EE Platform
- A Business Guide To Cloud Deployment Options For Jakarta EE Applications
- A Developer Guide to NoSQL Persistence on The Jakarta EE Platform With Google Firestore

sales@payara.fish

UK: +44 800 538 5490
Intl: +1 888 239 8941

www.payara.fish