Creating Java Applications with WebSockets

10

ORACLE

Objectives

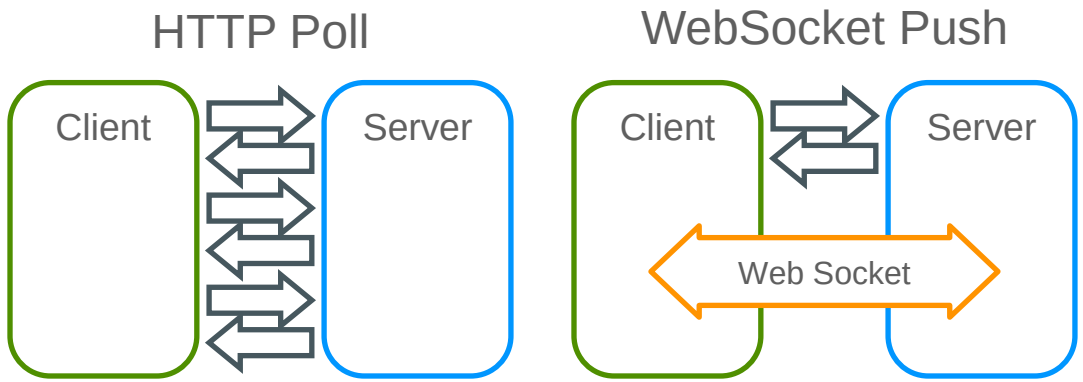
After completing the lesson, you should be able to:

- Explain WebSockets communication style
- Create WebSocket Endpoint Handlers using JSR 356 API
- Manage WebSocket Endpoint life cycle
- Produce and consume WebSocket messages
- Handle errors
- Encode and decode JSON messages
- Provide WebSocket Client Endpoint handler using Java and JavaScript

WebSockets Network Protocol

WebSocket protocol supports bidirectional communication between client and server.

- Unlike WebSocket, HTTP Protocol only allows clients to be originators of calls.
- WebSocket connection starts with an HTTP handshake but does not use HTTP after that.
- Once a WebSocket connection is established, it allows either side (client or server) to send or receive messages in any order, enabling server to push information to the client.
- To support a WebSocket connection, the server can provide one instance of endpoint handler for the life of this connection, making application stateful in a very similar way to that of an HTTP Session.
- WebSocket clients are typically implemented using JavaScript.

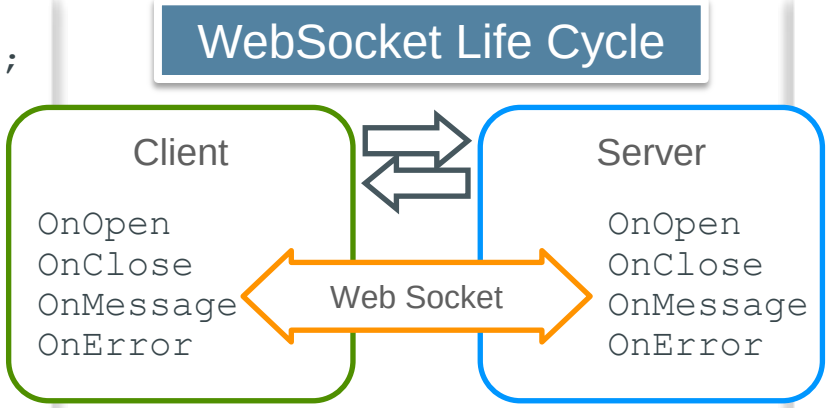


WebSocket Life Cycle

WebSocket defines Client and Server communication endpoints handlers.

- Both Client and Server endpoints provide symmetrical capabilities.
- They are defined by the following operations:
  - OnOpen - is invoked when an new WebSocket connection is opened
  - OnClose - is invoked when an new WebSocket connection is closed
  - OnMessage - is invoked every time a message arrives from Server or from Client via the opened socket
  - OnError - is invoked when errors occurred when handling socket communications
- This example illustrates JavaScript Client and Java Server Endpoint handlers

```
var socket = new WebSocket("uri");
socket.onopen
    = function(event){...}
socket.onclose
    = function(event){...}
socket.onmessage
    = function(event){...}
socket.onerror
    = function(event){...}
```




```
@ServerEndpoint("uri")
public class ServerHandler {
    @OnOpen
    public void openSocket...
    @OnClose
    public void closeSocket...
    @OnMessage
    public void handleMessage...
    @OnError
    public void handleError...
}
```

Defining WebSocket Endpoints

WebSocket server endpoints are annotated and packaged in a WAR.

- ServerEndpoint class is mapped to WebSocket url, possibly with parameters.
- Both @ClientEndpoint and @ServerEndpoint classes contain methods that respond to various events:
  - @OnOpen may accept EndpointConfig and custom parameters.
  - @OnClose may accept CloseReason and custom parameters.
  - @OnMessage must accept text or binary message and may accept custom parameters and may return a value.
  - @OnError must accept Throwable and may accept custom parameters.
- All these event handler methods may optionally accept WebSocket Session Object.

```
ws://www.example.com/demos/order
```

```
@ServerEndpoint("/order")
public class OrderServerWebSocketHandler {
   @OnOpen public void openSocket(Session session){...}
   @OnClose public void closeSocket(CloseReason reason) {...}
   @OnMessage public void handleMessage(String message) {...}
   @OnError public void handleError(Throwable error) {...}
}
```

Using PathParam Annotation

The PathParam annotation maps WebSocket operation parameters to URL.

- Such parameters form part of the URI template for the given WebSocket Server endpoint.
- WebSocket operations may access such parameters.
- Different other values may be passed between the client and server endpoints OnMessage operations.

```
ws://www.example.com/demos/order/101
```

```
@ServerEndpoint("/order/{id}")
public class OrderServerWebSocketHandler {
    @OnOpen
    public void openSocket(Session session, @PathParam("id") int id){...}
    @OnClose
    public void closeSocket(CloseReason reason, @PathParam("id") int id){...}
    @OnMessage
    public void handleMessage(String message, @PathParam("id") int id){...}
    @OnError
    public void handleError(Throwable error, @PathParam("id") int id){...}
}
```

Using WebSocket Session

A WebSocket session represents a conversation between two web socket endpoints.

A WebSocket session allows to:

- Find session status with the isOpen method
- Register one of each type of message handlers to handle incoming messages for this session:
  - Text Message Handler
  - Binary Message Handler
  - Pong Message Handler
- Close session when the conversation between client and server is over
  - Session closed with no parameters defaults to NORMAL_CLOSURE and no reason message
- Access security information
- Access set of other sessions related to the same endpoint to implement cross-session conversations

```
session.isOpen();
session.addMessageHandler(...);
session.setMaxIdleTimeout(...);
Principal principal = session.getUserPrincipal();
Set<Session> allSessions = session.getOpenSessions();
session.close(CloseReason.CloseCodes.TRY_AGAIN_LATER);
```

Using RemoteEndpoint Objects

RemoteEndpoint Objects are used to send messages to the session counterpart.

- RemoteEndpoint represents a ClientEndpoint on a server or ServerEndpoint on a client.
- RemoteEndpoints could be of two types:
  - Synchronous
  - Asynchronous

```
RemoteEndpoint.Basic br = session.getBasicRemote();
br.sendObject(...);
```

```
RemoteEndpoint.Async ar = session.getAsyncRemote();
Future<Void> transmissionStatus = ar.sendObject(...);
```

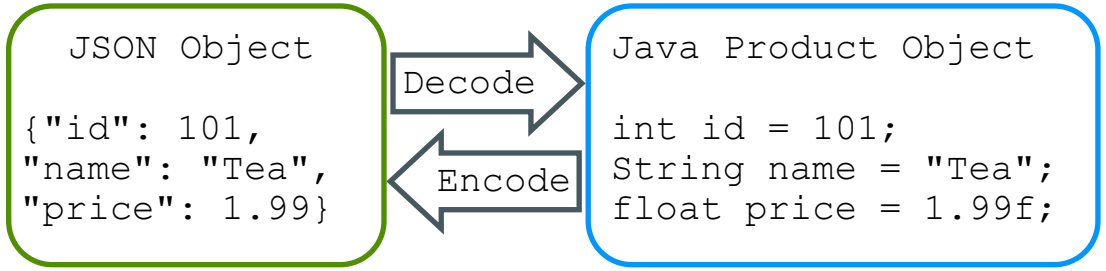
- RemoteEndpoints are used to implement communications at any phase of a WebSocket lifecycle within the OnOpen, OnClose, OnMessage, and OnError operations.
- The onMessage operation may simply return a value if response is produced as a direct result of a received call.

Encode and Decode Messages

Messages that travel between server and client endpoints must be converted to and from Java.

- JavaScript Object Notation (JSON) is commonly used by RESTful web services and WebSocket applications.
- Java API for JSON Processing (JSON-P) JSR 353 is used to perform JSON message encoding and decoding.
- Streaming JSON-P classes:
  - JsonParser – A pull parser for reading JSON data
  - JsonGenerator – A JSON generator that uses method chaining
- Object-based JSON-P classes:
  - JsonReader – Reads from an InputStream and produces an object graph
  - JsonWriter – Writes a JSON-P–specific object graph to an OutputStream
- JsonObject class represents JSON Object in Java.
  - Messages can be encoded and decoded within server and client endpoint classes, or by separate classes, registered with endpoint handlers.

❖ For more information on JavaScript and how it is used to implement WebSocket communications, refer to "JavaScript and HTML5: Develop Web Applications" training course.



Handle WebSocket Messages

The OnMessage operation handles client calls within a server or server calls within a client endpoint.

- Must accept text or binary message and may accept custom parameters
- May accept additional parameters mapped with PathParam annotation
- May accept Session parameter
- May be void or may return a value

```
@OnMessage
public String findProduct(String value) {
   JsonObject inObj = Json.createReader(new StringReader(value)).readObject();
   int id = inObj.getInt("id");
   Product product = pm.findProduct(id);
   if (product == null) {
      throw new RuntimeException("Product with id " + id + " not found");
   }
   JsonObject outObj = Json.createObjectBuilder().add("id", product.getId())
                           .add("name", product.getName())
                           .add("price", product.getPrice()).build();
   StringWriter stringWriter = new StringWriter();
   JsonWriter writer = Json.createWriter(stringWriter);
   writer.writeObject(outObj);
   writer.close();
   return stringWriter.getBuffer().toString();
}
```

Handle WebSocket Errors

The OnError operation handles exceptions produced by other WebSocket operations.

- Must accept Throwable parameter
- May accept additional parameters mapped with PathParam annotation
- May accept Session parameter
- RemoteEndpoint object used to dispatch error messages to the WebSocket counterpart

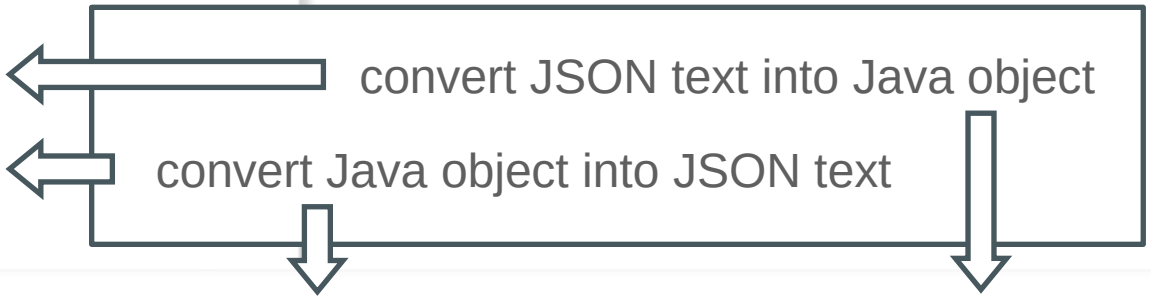
```
@OnError
public void handleError(Session session, Throwable exception) {
   RemoteEndpoint.Basic remote = session.getBasicRemote();
   try {
      remote.sendObject(exception.getMessage());
   } catch (IOException ex | EncodeException ex) {
      Logger.getLogger(WebSocketServer.class.getName()).log(Level.SEVERE, null, ex);
      throw new RuntimeException("Error sending product", ex);
   }
}
```

Encoding and Decoding WebSocket Messages

Encodes and Decoders convert Java Objects to and from WebSocket operable formats such as JSON.

- OnMessage operation or RemoteEndpoint objects are allowed to send of receive custom Java Objects only if the appropriate encoders and decodes are registered.

```
@ServerEndpoint(value="/order")
public class ProductServerSocketHandler {
   @OnMessage
   public String updatePrice(String product) {
      ...
      product.setPrice(2.99f); // use Java object
      ...
      return jsonText;
   }
...
```




```
@ServerEndpoint(value="/order",encoders={Product2JSON.class},decoders={JSON2Product.class})
public class ProductServerSocketHandler {
   @OnMessage
   public Product updatePrice(Product product) {
      product.setPrice(2.99f); // just use Java object
      return product;
   }
...
```

Implementing WebSocket Message Encoder

Encoders are used to convert Java Objects into WebSocket writeable format such as JSON.

- Override encode method:
  - Accept Java Object that requires conversion
  - Return conversion result such as JSON String
- Depending on required conversion result type implement on of the following interfaces:
  - Encoder.Text
  - Encoder.TextStream
  - Encoder.Binary
  - Encoder.BinaryStream

```
public class Product2JSON implements Encoder.Text<Product> {
   @Override public String encode(Product product)
                                      throws EncodeException {
      JsonObject obj = Json.createObjectBuilder()
                           .add("id", product.getId())
                           .add("name", product.getName())
                           .add("price", product.getPrice()).build();
      StringWriter stringWriter = new StringWriter();
      Json.createWriter(stringWriter).writeObject(product).close();
      return stringWriter.getBuffer().toString();
   }
   @Override public void init(EndpointConfig config) {}
   @Override public void destroy() {}
}
```

Implementing WebSocket Message Decoder

Decoders are used to convert WebSocket operable format such as JSON into Java Objects.

- Override decode method
  - Accept value that requires conversion
  - Return converted Java Object
- Depending on accepted value type implement on of the following interfaces:
  - Decoder.Text
  - Decoder.TextStream
  - Decoder.Binary
  - Decoder.BinaryStream

```
public class JSON2Product implements Decoder.Text<Product> {
   @Override public Product decode(String message)
                                      throws DecodeException {
      JsonObject obj = Json.createReader(
                     new StringReader(message)).readObject();
      int id = obj.getInt("id");
      String name = obj.getString("name");
      float price = (float)(obj.getJsonNumber("price").doubleValue());
      return new Product(id, name, price);
   }
   @Override public boolean willDecode(String message) {
      return message != null; //can write message validation here
   }
   @Override public void init(EndpointConfig config) {}
   @Override public void destroy() {}
}
```

Creating JSON Messages

When implementing JSON Encoder, choose between:

- JsonGenerator, builds JSON text and writes it to an OutputStream or a Writer

  Writing data directly to stream

```
JsonGenerator jgen = Json.createGenerator(<target stream>);
jgen.writeStartObject().write("id", "101")
                       .write("name", "Tea").writeEnd();
jgen.flush();
```

- JsonWriter, writes that object graph as JSON text to an OutputStream or a Writer

  A JsonObjectBuilder allows to build an object graph of JSON-specific objects.

```
JsonWriter jw = Json.createWriter(<target stream>);
JsonObject obj = Json.createObjectBuilder()
                        .add("id", "101")
                        .add("name", "Tea").build();
jw.write(obj);
```

Parsing JSON Messages

When implementing JSON Decoder choose between:

- A JsonParser converts JSON text into a sequence of events.

  It is a fast, low-level method for parsing JSON without loading the JSON data into an object graph.

```
JsonParser parser = Json.createParser(<input stream>);
while(parser.hasNext()) {
    Event e = parser.next();
    switch(e) {
        case JsonParser.Event.KEY_NAME:
          String elementName = parser.getString();
        case JsonParser.Event.VALUE_STRING:
          String elementValue = parser.getString();
        case JsonParser.Event.VALUE_NUMBER:
          int numberValue = parser.getInt();
    }
}
```

- A JsonReader converts JSON text into an object graph.

  Convenient to use for smaller objects

```
JsonReader reader =  Json.createReader(<input stream>);
JsonObject obj = reader.readObject();
```

Invoking WebSocket Server from a JavaScript Client

JavaScript may invoke WebSocket Server Endpoint handler and receive callbacks from it.

- New WebSocket object is opened pointing to the WebSocket Server Endpoint.
- Web Socket Client Endpoint life-cycle operations are registered with the socket.
  - onmessage - Handles messages received from the server
  - onerror - Handles WebSocket errors
  - onopen - Executes custom code when socket is opened
  - onclose - Executes custom code when socket is closed
- Use send operation to dispatch messages to the WebSocket Server Endpoint.

```
var socket = new WebSocket("ws://www.example.com/demos/product");
socket.onmessage = function (event) {
   var product = JSON.parse(event.data);
   // handle product object
}
socket.onerror = function(event){ alert(event.data); }
function findProduct() {
    var productId = document.getElementById("pid").value;
    socket.send(productId);
}
```

❖ For more information on JavaScript and how it is used to implement WebSocket communications, refer to "JavaScript and HTML5: Develop Web Applications" training course.

Invoking WebSocket Server from a Java Client

Java Application can implement a WebSocket Client Endpoint handler.

- WebSocket Client Endpoint mirrors the Server Endpoint, providing OnOpen, OnClose, OnError and OnMessage operations
- WebSocket Client:
  - Connects to WebSocket Server
  - Registers Client Endpoint Handler
  - Acquires RemoteEndpoint
  - Prepares and dispatches messages
  - Closes WebSocket session

```
@ClientEndpoint
public class SocketHandler {
@OnOpen
public void onOpen(Session session) {...}
@OnClose
public void onClose(Session session, CloseReason reason) {...}
@OnError
public void onError(Session session, Throwable ex) {...}
@OnMessage
public void onMessage(String message) {...}
```

```
URI uri = new URI("ws://www.example.com/demos/order");
WebSocketContainer container = ContainerProvider.getWebSocketContainer();
Session session = container.connectToServer(new SocketHandler(), uri);
RemoteEndpoint.Async remote = session.getAsyncRemote();
Product product = new Product();
...
remote.sendObject(product);
session.close(new CloseReason(CloseReason.CloseCodes.NORMAL_CLOSURE, "Goodbye"));
```

Summary

In this lesson, you should have learned how to:

- Explain WebSockets communication style
- Create WebSocket Endpoint Handlers using JSR 356 API
- Manage WebSocket Endpoint life cycle
- Produce and consume WebSocket messages
- Handle errors
- Encode and decode JSON messages
- Provide WebSocket Client Endpoint handler using Java and JavaScript

Practice

This practice covers the following tasks:

- Creating WebSocket Chat Server to allow callers to exchange chat messages
- Creating HTML and JavaScript clients to interact with the WebSocket Chat Server
- Creating a Java Client Application to interact with the WebSocket Chat Server