JAKARTA EE

Jakarta WebSocket Specification

Jakarta WebSocket Team, https://projects.eclipse.org/projects/ee4j.websocket

2.1, December 22, 2021:

Table of Contents

```
Specification: Jakarta WebSocket Specification

Version: 2.1

Status: Final Release

Release: December 22, 2021
```


Eclipse Foundation Specification License

By using and/or copying this document, or the Eclipse Foundation document from which this statement is linked, you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions:

Permission to copy, and distribute the contents of this document, or the Eclipse Foundation document from which this statement is linked, in any medium for any purpose and without fee or royalty is hereby granted, provided that you include the following on ALL copies of the document, or portions thereof, that you use:

- link or URL to the original Eclipse Foundation document.
- All existing copyright notices, or if one does not exist, a notice (hypertext is preferred, but a textual representation is permitted) of the form: "Copyright © [$date-of-document] Eclipse Foundation, Inc. <<url to this license>>"

Inclusion of the full text of this NOTICE must be provided. We request that authorship attribution be provided in any software, documents, or other items or products that you create pursuant to the implementation of the contents of this document, or any portion thereof.

No right to create modifications or derivatives of Eclipse Foundation documents is granted pursuant to this license, except anyone may prepare and distribute derivative works and portions of this document in software that implements the specification, in supporting materials accompanying such software, and in documentation of such software, PROVIDED that all such works include the notice below. HOWEVER, the publication of derivative works of this document for use as a technical specification is expressly prohibited.

The notice is:

"Copyright © 2018, 2020 Eclipse Foundation. This software or document includes material copied from or derived from Jakarta ® WebSocket https://jakarta.ee/specifications/websocket/2.1/"

Disclaimers

THIS DOCUMENT IS PROVIDED "AS IS," AND THE COPYRIGHT HOLDERS AND THE ECLIPSE FOUNDATION MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DOCUMENT ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

THE COPYRIGHT HOLDERS AND THE ECLIPSE FOUNDATION WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE DOCUMENT OR THE PERFORMANCE OR IMPLEMENTATION OF THE CONTENTS THEREOF.

The name and trademarks of the copyright holders or the Eclipse Foundation may NOT be used in advertising or publicity pertaining to this document or its contents without specific, written prior permission. Title to copyright in this document will at all times remain with copyright holders.

Jakarta WebSocket Specification, Version 2.1

The Jakarta WebSocket Team - December 22, 2021

Comments to: websocket-dev@eclipse.org

Chapter 1. Introduction

This specification defines a set of Java APIs for the development of WebSocket applications. Readers are assumed to be familiar with the WebSocket protocol. The WebSocket protocol, developed as part of the collection of technologies that make up HTML5, promises to bring a new level of ease of development and network efficiency to modern, interactive web applications. For more information on the WebSocket protocol see:

- The WebSocket Protocol specification (Fette and Melnikov 2011)
- The WebSocket API for JavaScript (Hickson 2012)

1.1. Purpose of this document

This document in combination with the API documentation for the Jakarta WebSocket API is the specification of the Jakarta WebSocket API. The specification defines the requirements that an implementation must meet in order to be an implementation of the Jakarta WebSocket API. This specification has been developed under the Eclipse Foundation Specification Process. Together with the Test Compatibility Kit (TCK) which tests that a given implementation meets the requirements of the specification, and Compatible Implementations (CIs) that implement this specification and which pass the TCK, this specification defines the Jakarta standard for WebSocket application development.

While there is much useful information in this document for developers using the Jakarta WebSocket API, its purpose is not to be a developers guide. Similarly, while there is much useful information in this document for developers who are creating an implementation of the Jakarta WebSocket API, its purpose is not to be a 'How To' guide as to how to implement all the required features.

1.2. Goals of the Specification

The goal of this specification is to define the requirements on containers that wish to support APIs for WebSocket programming on the Jakarta and Java Platforms. While the document may be a useful reference for developers who use the APIs defined by this specification, this document is not a developer guide.

1.3. Terminology used throughout the Specification

endpoint

A WebSocket endpoint is a Java component that represents one side of a sequence of WebSocket interactions between two connected peers.

connection

A WebSocket connection is the networking connection between the two endpoints which are interacting using the WebSocket protocol.

peer

Used in the context of a WebSocket endpoint, the WebSocket peer is used to represent another participant of the WebSocket interactions with the endpoint.

session

The term WebSocket session is used to represent a sequence of WebSocket interactions between an endpoint and a single peer.

client endpoints and server endpoints

A client endpoint is one that initiates a connection to a peer but does not accept new ones. A server endpoint is one that accepts WebSocket connections from peers but does not initiate connections to peers.

1.4. Specification Conventions

The keywords 'MUST', 'MUST NOT', 'REQUIRED', 'SHALL', 'SHALL NOT', 'SHOULD', 'SHOULD NOT', 'RECOMMENDED', 'MAY', and 'OPTIONAL' in this document are to be interpreted as described in RFC 2119 (Bradner 1997).

Additionally, requirements of the specification that can be tested using the conformance test suite are marked with the figure WSC (WebSocket Compatibility) followed by a number which is used to identify the requirement, for example 'WSC-12'.

Java code and sample data fragments are formatted as shown below:

```
package com.example.hello;

public class Hello {

    public static void main(String args[]) {
        System.out.println("Hello World");
    }
}
```

URIs of the general form 'http://example.org/...' and 'http://example.com/...' represent application or context-dependent URIs.

All parts of this specification are normative, with the exception of examples, notes and sections explicitly marked as 'Non-Normative'. Non-normative notes are formatted as shown below.

Note: *This is a note.*

1.5. Previous work in the JCP

Prior to version 2.0, this specification was developed in the Java Community Process as part of JSR 356.

Chapter 2. Applications

Jakarta WebSocket applications consist of WebSocket endpoints. A WebSocket endpoint is a Java object that represents one end of a WebSocket connection between two peers.

There are two main means by which an endpoint can be created. The first means is to implement certain of the API classes from the Jakarta WebSocket API with the required behavior to handle the endpoint lifecycle, consume and send messages, publish itself, or connect to a peer. Often, this specification will refer to this kind of endpoint as a *programmatic endpoint*. The second means is to decorate a Plain Old Java Object (POJO) with certain of the annotations from the Jakarta WebSocket API. The implementation then takes these annotated classes and creates the appropriate objects at runtime to deploy the POJO as a WebSocket endpoint. Often, this specification will refer to this kind of endpoint as an *annotated endpoint*. The specification will refer to an endpoint when it is talking about either kind of endpoint: programmatic or annotated.

The endpoint participates in the opening handshake that establishes the WebSocket connection. The endpoint will typically send and receive a variety of WebSocket messages. The endpoint's lifecycle comes to an end when the WebSocket connection is closed.

2.1. API Overview

This section gives a brief overview of the Jakarta WebSocket API in order to set the stage for the detailed requirements that follow.

2.1.1. Endpoint Lifecycle

A logical WebSocket endpoint is represented in the Jakarta WebSocket API by instances of the **Endpoint** class. Developers may subclass the **Endpoint** class with a public, concrete class in order to intercept lifecycle events of the endpoint: those of a peer connecting, an open connection ending and an error being raised during the lifetime of the endpoint.

Unless otherwise overridden by a developer provided configurator (see Section 3.1.7), the WebSocket implementation must use one instance per application per VM of the **Endpoint** class to represent the logical endpoint per connected peer [WSC 2.1.1-1]. Each instance of the **Endpoint** class in this typical case only handles connections to the endpoint from one and only one peer.

2.1.2. Sessions

The Jakarta WebSocket API models the sequence of interactions between an endpoint and each of its peers using an instance of the **Session** class. The interactions between a peer and an endpoint begin with an open notification, followed by some number, possibly zero, of WebSocket messages between the endpoint and peer, followed by a close notification or possibly a fatal error which terminates the connection. For each peer that is interacting with an endpoint, there is one unique **Session** instance that represents that interaction [WSC 2.1.2-1]. This **Session** instance corresponding to the connection

with that peer is passed to the endpoint instance representing the logical endpoint at the key events in its lifecycle.

Developers may use the user property map accessible through the **getUserProperties()** call on the **Session** object to associate application specific information with a particular session. The WebSocket implementation must preserve this session data for later access until the completion of the **onClose()** method on the endpoint instance [WSC 2.1.2-2]. After that time, the WebSocket implementation is permitted to discard the developer data. A WebSocket implementation that chooses to pool **Session** instances may at that point re-use the same **Session** instance to represent a new connection provided it issues a new unique **Session** id [WSC 2.1.2-3].

WebSocket implementations that are part of a distributed container may need to migrate WebSocket sessions from one node to another in the case of a failover. Implementations are required to preserve developer data objects inserted into the WebSocket session if the data is marked **java.io.Serializable** [WSC 2.1.2-4].

The user properties provided by the **Session** object are initially populated from the per **Session** shallow copy of the user properties provided by **EndpointConfig.getUserProperties()** that was passed to the **modifyHandshake()** method on the **ServerEndpointConfig.Configurator** including any modifications made during the execution of that method.

2.1.3. Receiving Messages

The Jakarta WebSocket API presents a variety of means for an endpoint to receive messages from its peers. Developers implement the subtype of the **MessageHandler** interface with the message delivery style that best suits their needs, and register the interest in messages from a particular peer by registering the handler on the Session instance corresponding to the peer.

The API limits the registration of **MessageHandlers** per **Session** to be one **MessageHandler** per native WebSocket message type [WSC 2.1.3-1]. In other words, the developer can only register at most one **MessageHandler** for incoming text messages, one **MessageHandler** for incoming binary messages, and one **MessageHandler** for incoming pong messages. The WebSocket implementation must generate an error if this restriction is violated [WSC 2.1.3-2].

Future versions of the specification may lift this restriction.

The registered **MessageHandlers** for a **Session** may be changed at runtime. To avoid any ambiguity, once the container has identified a **MessageHandler** for a message, the **MessageHandler** is used for the entirety of the message irrespective of any subsequent changes to the **MessageHandlers** configured for the **Session**.

Method **Session.addMessageHandler(MessageHandler)** is not safe for use in all circumstances, especially when using Lambda Expressions. The API forces implementations to get the **MessageHandler**'s type parameter in runtime, which is not always possible. The only case where you can safely use this method is when you are directly implementing **MessageHandler.Whole** or **MessageHandler.Partial** as an anonymous class. This approach guarantees that generic type

information will be present in the generated class file and the runtime will be able to get it. For any other case (Lambda Expressions included), one of following methods have to be used: **Session.addMessageHandler(Class<T>, MessageHandler.Partial<T>)** or **Session.addMessageHandler(Class<T>, MessageHandler.Whole<T>)**.

2.1.4. Sending Messages

The Jakarta WebSocket API models each peer of a session with an endpoint as an instance of the **RemoteEndpoint** interface. This interface and its two subtypes (**RemoteEndpoint.Whole** and **RemoteEndpoint.Partial**) contain a variety of methods for sending WebSocket messages from the endpoint to its peer.

Here is an example of a server endpoint that waits for incoming text messages, and responds immediately when it gets one to the client that sent it. The example endpoint is shown, first using only the API classes:

```
public class HelloServer extends Endpoint {
    @Override
    public void onOpen(Session session, EndpointConfig ec) {
        final RemoteEndpoint.Basic remote = session.getBasicRemote();
        session.addMessageHandler(String.class,
            new MessageHandler.Whole<String>() {
                public void onMessage(String text) {
                    try {
                        remote.sendText("Got your message (" + text + "). Thanks !");
                    } catch (IOException ioe) {
                        ioe.printStackTrace();
                    }
                }
        });
    }
}
```

and second using the annotations in the API:

```
@ServerEndpoint("/hello")
public class MyHelloServer {
    @OnMessage
    public String handleMessage(String message) {
        return "Got your message (" + message + "). Thanks !";
    }
}
```

Note: *The examples are almost equivalent save for the annotated endpoint carries its own path mapping.*

2.1.5. Closing Connections

If an open connection to a WebSocket endpoint is to be closed for any reason, whether as a result of receiving a WebSocket close event from the peer, or because the underlying implementation has reason to close the connection, the WebSocket implementation must invoke the **onClose()** method of the WebSocket endpoint [WSC 2.1.5-1].

If the close was initiated by the remote peer, the implementation must use the close code and reason sent in the WebSocket protocol close frame. If the close was initiated by the local container, for example if the local container determines the session has timed out, the local implementation must use the WebSocket protocol close code `1006` (a code especially disallowed in close frames on the wire), with a suitable close reason. That way the endpoint can determine whether the close was initiated remotely or locally. If the session is closed locally, the implementation must attempt to send the WebSocket close frame prior to calling the **onClose()** method of the WebSocket endpoint.

2.1.6. Clients and Servers

The WebSocket protocol is a two-way protocol. Once established, the WebSocket protocol is symmetrical between the two parties in the conversation. The difference between a WebSocket *client* and a WebSocket *server* lies only in the means by which the two parties are connected. In this specification, we will say that a WebSocket *client* is a WebSocket endpoint that initiates a connection to a peer. We will say that a WebSocket *server* is a WebSocket endpoint that is published and awaits connections from peers. In most deployments, a WebSocket client will connect to only one WebSocket server, and a WebSocket server will accept connections from several clients.

Accordingly, the WebSocket API only distinguishes between endpoints that are WebSocket clients from endpoints that are WebSocket servers in the configuration and setup phase.

2.1.7. WebSocketContainers

The WebSocket implementation is represented to applications by instances of the **WebSocketContainer** class. Each **WebSocketContainer** instance carries a number of configuration properties that apply to endpoints deployed within it. In server deployments of WebSocket implementations, there is one unique **WebSocketContainer** instance per application per Java VM [WSC 2.1.7-1]. In client deployments of WebSocket implementations, applications obtain instances of the **WebSocketContainer** from the **ContainerProvider** class.

2.2. Endpoints using WebSocket Annotations

Java annotations have become widely used as a means to add deployment characteristics to Java objects, particularly in the Jakarta EE platform. The Jakarta WebSocket specification defines a small number of WebSocket annotations that allow developers to take Java classes and turn them into WebSocket endpoints. This section gives a short overview to set the stage for more detailed requirements later in this specification.

2.2.1. Annotated Endpoints

The class level **@ServerEndpoint** annotation indicates that a Java class is to become a WebSocket endpoint at runtime. Developers may use the value attribute to specify a URI mapping for the endpoint. The **encoders** and **decoders** attributes allow the developer to specify classes that encode application objects into WebSocket messages, and decode WebSocket messages into application objects.

2.2.2. WebSocket Lifecycle

The method level **@OnOpen** and **@OnClose** annotations allow the developers to decorate methods on their **@ServerEndpoint** annotated Java class to specify that they must be called by the implementation when the resulting endpoint receives a new connection from a peer or when a connection from a peer is closed, respectively [WSC 2.2.2-1].

2.2.3. Handling Messages

In order that the annotated endpoint can process incoming messages, the method level **@OnMessage** annotation allows the developer to indicate which methods the implementation must call when a message is received [WSC 2.2.3-1].

2.2.4. Handling Errors

In order that an annotated endpoint can handle errors that occur as a arising from external events, for example on decoding an incoming message, an annotated endpoint can use the **@OnError** annotation to mark one of its methods that must be called by the implementation with information about the error whenever such an error occurs [WSC 2.2.4-1].

2.2.5. Pings and Pongs

The ping/pong mechanism in the WebSocket protocol serves as a check that the connection is still active. Following the requirements of the protocol, if a WebSocket implementation receives a ping message from a peer, it must respond as soon as possible to that peer with a pong message containing the same application data [WSC 2.2.5-1]. Developers who wish to send a unidirectional pong message may do so using the **RemoteEndpoint** API. Developers wishing to listen for returning pong messages may either define a **MessageHandler** for them, or annotate a method using the **@OnMessage** annotation where the method stipulates a **PongMessage** as its message entity parameter. In either case, if the implementation receives a pong message addressed to this endpoint, it must call that MessageHandler or that annotated method [WSC 2.2.5-2].

2.3. Jakarta WebSocket Client API

This specification defines two configurations of the Jakarta WebSocket API. The Jakarta WebSocket API is used to mean the full functionality defined in this specification. This API is intended to be implemented either as a standalone WebSocket implementation, as part of a Jakarta Servlet container, or as part of a full Jakarta EE platform implementation. The APIs that must be implemented to conform

to the Jakarta WebSocket API are all the Java APIs in the packages **jakarta.websocket.*** and **jakarta.websocket.server.***. Some of the non-API features of the Jakarta WebSocket API are optional when the API is not implemented as part of the full Jakarta EE platform, for example, the requirement that WebSocket endpoints be non-contextual managed beans (see Chapter 7). Such Jakarta EE only features are clearly marked where they are described.

The Jakarta WebSocket API also contains a subset of its functionality intended for desktop, tablet or smartphone devices. This subset does not contain the ability to deploy server endpoints. This subset known as the Jakarta WebSocket Client API. The APIs that must be implemented to conform to the Jakarta WebSocket Client API are all the Java APIs in the package **jakarta.websocket.***.

Chapter 3. Configuration

WebSocket applications are configured with a number of key parameters: the path mapping that identifies a WebSocket endpoint in the URI-space of the container, the subprotocols that the endpoint supports, and the extensions that the application requires. Additionally, during the opening handshake, the application may choose to perform other configuration tasks, such as checking the hostname of the requesting client, or processing cookies. This section details the requirements on the container to support these configuration tasks.

Both client and server endpoint configurations include a list of application provided encoder and decoder classes that the implementation must use to translate between WebSocket messages and application defined message objects [WSC-3-1].

3.1. Server Configurations

In order to deploy a programmatic endpoint into the URI space available for client connections, the container requires a **ServerEndpointConfig** instance. This object holds configuration data and the default implementation provided algorithms needed by the implementation to configure the endpoint. The WebSocket API allow certain of these configuration operations to be overriden by developers by providing a custom **ServerEndpointConfig.Configurator** implementation with the **ServerEndpointConfig** [WSC-3.1-1].

These operations are laid out below.

3.1.1. URI Mapping

This section describes the the URI mapping policy for server endpoints.

All server endpoint paths must:

- be a URI-template (level-1) or a partial URI
- start with a leading '/'
- not contain the sequences `/../`, `/./` or `//`

Additionally, URI-template server endpoint paths must:

- Only replace whole URI segments with variables
- Not use the same variable more than once in a path

For a definition of URI segments, see RFC 3986 (Berners-Lee et al. 2005). For a definition of URI-templates, see RFC 6570 (Gregorio et al. 2012).

The WebSocket implementation must compare the normalized - see section 6 of RFC 3986 (Berners-Lee et al. 2005) - incoming URI to the collection of all endpoint paths and determine the best match. The

incoming URI in an opening handshake request matches an endpoint path if either it is an exact match in the case where the endpoint path is a relative URI, and if it is a valid expansion of the endpoint path in the case where the endpoint path is a URI template [WSC-3.1.1-1].

An application that contains multiple endpoint paths that are the same relative URI is not a valid application. An application that contains multiple endpoint paths that are equivalent URI-templates is not a valid application [WSC-3.1.1-2].

However, it is possible for an incoming URI in an opening handshake request theoretically to match more than one endpoint path. For example, consider the following case:

- incoming URI: "/a/b"
- endpoint A is mapped to "/a/b"
- endpoint B is mapped to /a/{customer-name}

The WebSocket implementation will attempt to match an incoming URI to an endpoint path (URI or level 1 URI-template) in the application in a manner equivalent to the following: [WSC-3.1.1-3]

Since the endpoint paths are either relative URIs or URI templates level 1, the paths do not match if they do not have the same number of segments, using '/' as the separator. So, the container will traverse the segments of the endpoint paths with the same number of segments as the incoming URI from left to right, comparing each segment with the corresponding segment of the incoming URI. At each segment, the implementation will retain those endpoint paths that match exactly, or if there are none, those that are a variable segment, before moving to check the next segment. If there is an endpoint path at the end of this process there is a match.

Because of the requirement disallowing multiple endpoint paths and equivalent URI-templates, and the preference for exact matches at each segment, there can only be at most one path, and it is the best match.

Examples

i. suppose an endpoint has path /a/b/, the only incoming URI that matches this is /a/b/

ii. suppose an endpoint is mapped to /a/{var}
    - incoming URIs that do match:
        - /a/b (with var=b)
        - /a/apple (with var=apple)
    - URIs that do NOT match (because empty string and strings with reserved characters "/" are not valid URI-template level 1 expansions):
        - /a
        - /a/b/
        - /a/b/c

iii. suppose we have three endpoints and their paths:

- endpoint A: /a/{var}/c
- endpoint B: /a/b/c
- endpoint C: /a/{var1}/{var2}
- incoming URI: a/b/c matches B, not A or C, because an exact match is preferred.
- incoming URI: a/d/c matches A with variable var=d, because an exact matching segment is preferred over a variable segment
- incoming URI: a/x/y/ matches C, with var1=x, var2=y

iv. suppose we have two endpoints

- endpoint A: /{var1}/d
- endpoint B: /b/{var2}
- incoming URI: /b/d matches B with var2=d, not A with var1=b because the matching process works from left to right.

The implementation must not establish the connection unless there is a match [WSC-3.1.1-4].

3.1.2. Subprotocol Negotiation

The default server configuration must be provided a list of supported subprotocols in order of preference at creation time. During subprotocol negotiation, this configuration examines the client-supplied subprotocol list and selects the first subprotocol in the list it supports that is contained within the list provided by the client, or none if there is no match [WSC-3.1.2-1].

3.1.3. Extension Modification

In the opening handshake, the client supplies a list of extensions that it would like to use. The default server configuration selects from those extensions the ones it supports, and places them in the same order as requested by the client [WSC-3.1.3-1].

3.1.4. Origin Check

The default server configuration makes a check of the hostname provided in the Origin header, failing the handshake if the hostname cannot be verified [WSC-3.1.4-1].

3.1.5. Handshake Modification

The default server configuration makes no modification of the opening handshake process other than that described above [WSC-3.1.5-1].

Developers may wish to customize the configuration and handshake negotiation policies laid out above. In order to do so, they may provide their own implementations of **ServerEndpointConfig.Configurator**.

For example, developers may wish to intervene more in the handshake process. They may wish to use Http cookies to track clients, or insert application specific headers in the handshake response. In order to do this, they may implement the **modifyHandshake()** method on the **ServerEndpointConfig.Configurator**, wherein they have full access to the **HandshakeRequest** and **HandshakeResponse** of the handshake.

The user properties exposed during the **modifyHandshake()** method must be a per WebSocket connection (i.e. per **Session**) shallow copy of the user properties provided by **EndpointConfig.getUserProperties()**. When the **Session** object is created these user properties, including any modifications made during **modifyHandshake()**, must be used as the initial user properties for the **Session**.

3.1.6. Custom State or Processing Across Server Endpoint Instances

The developer may also implement **ServerEndpointConfig.Configurator** in order to hold custom application state or methods for other kinds of application specific processing that is accessible from all **Endpoint** instances of the same logical endpoint via the **EndpointConfig** object.

3.1.7. Customizing Endpoint Creation

The developer may control the creation of endpoint instances by supplying a **ServerEndpointConfig.Configurator** object that overrides the **getEndpointInstance()** call. The implementation must call this method each time a new client connects to the logical endpoint [WSC-3.1.7-1]. The platform default implementation of this method is to return a new instance of the endpoint class each time it is called [WSC-3.1.7-2].

In this way, developers may deploy endpoints in such a way that only one instance of the endpoint class is instantiated for all the client connections to the logical endpoints. In this case, developers are cautioned that such a 'singleton' instance of the endpoint class will have to program with concurrent calling threads in mind, for example, if two different clients send a message at the same time.

3.2. Client Configuration

In order to connect a WebSocket client endpoint to its corresponding WebSocket server endpoint, the implementation requires configuration information. Aside from the list of encoders and decoders, the Jakarta WebSocket API needs the following attributes:

3.2.1. Subprotocols

The default client configuration uses the developer provided list of subprotocols, to send in order of preference, the names of the subprotocols it would like to use in the opening handshake it formulates [WSC-3.2.1-1].

3.2.2. Extensions

The default client configuration must use the developer provided list of extensions to send, in order of preference, the extensions, including parameters, that it would like to use in the opening handshake it formulates [WSC-3.2.2-1].

3.2.3. SSLContext

The default client configuration uses the developer provided `SSLContext` to establish a secure WebSocket (wss) connection or an insecure WebSocket (ws) connection if the provided `SSLContext` is `null`. If there is an existing connection to the server that uses the same SSLContext and that connection supports multiplexing WebSocket connections then the container may choose to re-use that connection rather than creating a new one. Containers may provide container specific configuration to control this behaviour.

3.2.4. Client Configuration Modification

Some clients may wish to adapt the way in which the client side formulates the opening handshake interaction with the server. Developers may provide their own implementations of ClientEndpointConfig.Configurator which override the default behavior of the underlying implementation in order to customize it to suit a particular application's needs.

Chapter 4. Annotations

This section contains a full specification of the semantics of the annotations in the Jakarta WebSocket API.

4.1. @ServerEndpoint

This class level annotation signifies that the Java class it decorates must be deployed by the implementation as a WebSocket server endpoint and made available in the URI-space of the WebSocket implementation [WSC-4.1-1]. The class must be public, concrete, and have a public no-args constructor. The class may or may not be final, and may or may not have final methods.

4.1.1. value

The **value** attribute must be a Java string that is a partial URI or URI-template (level-1), with a leading '/'. For a definition of URI-templates, see RFC 6570 (Gregorio et al. 2012). The implementation uses the value attribute to deploy the endpoint to the URI space of the WebSocket implementation. The implementation must treat the value as relative to the root URI of the WebSocket implementation in determining a match against the request URI of an incoming opening handshake request [WSC-4.1.1-2]. The semantics of matching for annotated endpoints is the same as was defined in the previous chapter. The value attribute is mandatory; the implementation must reject a missing or malformed path at deployment time [WSC-4.1.1-3].

For example,

```
@ServerEndpoint("/bookings/{guest-id}")
public class BookingServer {
    @OnMessage
    public void processBookingRequest(
        @PathParam("guest-id") String guestID,
        String message,
        Session session) {
        // process booking from the given guest here
    }
}
```

In this case, a client will be able to connect to this endpoint with any of the URIs

- **/bookings/JohnSmith**
- **/bookings/SallyBrown**
- **/bookings/MadisonWatson**

However, were the endpoint annotation to be **@ServerEndpoint("/bookings/SallyBrown")**, then only

a client request to **/bookings/SallyBrown** would be able to connect to this WebSocket endpoint.

If URI-templates are used in the value attribute, the developer may retrieve the variable path segments using the **@PathParam** annotation, as described below.

Applications that contain more than one annotated endpoint may inadvertently use the same relative URI. The WebSocket implementation must reject such an application at deployment time with an informative error message that there is a duplicate path that it cannot resolve [WSC-4.1.1-4].

Applications may contain an endpoint mapped to a path that is an expanded form of a URI template that is used by another endpoint in the same application. In this case, the application is valid. Please refer to the previous chapter for a definition of how to resolve the best match in this type of situation.

Future versions of the specification may allow higher levels of URI-templates.

4.1.2. encoders

The **encoders** attribute contains a (possibly empty) list of Java classes that are to act as encoder components for this endpoint. These classes must implement some form of the **Encoder** interface, have public no-arg constructors and be visible within the classpath of the application that this WebSocket endpoint is part of. The implementation must create a new instance of each encoder per connection per endpoint which guarantees no two threads are in the encoder at the same time. When sending an application object using the **RemoteEndpoint** API that is of a type that matches (same class or a sub-class) the parameterized type of the Encoder, the implementation must attempt to encode the object using the matching Encoder [WSC-4.1.2-1].

4.1.3. decoders

The **decoders** attribute contains a (possibly empty) list of Java classes that are to act as decoder components for this endpoint. These classes must implement some form of the **Decoder** interface, have public no-arg constructors and be visible within the classpath of the application that this WebSocket endpoint is part of. The implementation must create a new instance of each encoder per connection per endpoint. The implementation must attempt to decode WebSocket messages using the decoder in the list appropriate to the native WebSocket message type and pass the message in decoded object form to the WebSocket endpoint [WSC-4.1.3-1]. On **Decoder** implementations that have it, the implementation must use the **willDecode()** method on the decoder to determine if the **Decoder** will match the incoming message [WSC-4.1.3-2].

4.1.4. subprotocols

The **subprotocols** parameter contains a (possibly empty) list of string names of the subprotocols that this endpoint supports. The implementation must use this list in the opening handshake to negotiate the desired subprotocol to use for the connection it establishes [WSC-4.1.4-1].

4.1.5. configurator

The optional configurator attribute allows the developer to indicate that they would like the WebSocket implementation to use a developer provided implementation of **ServerEndpointConfig.Configurator**. If one is supplied, the WebSocket implementation must use this when configuring the endpoint [WSC-4.1.5-1]. The developer may use this technique to share state across all instances of the endpoint in addition to customizing the opening handshake.

4.2. @ClientEndpoint

This class level annotation signifies that the Java class it decorates is to be deployed as a WebSocket client endpoint that will connect to a WebSocket endpoint residing on a WebSocket server. The class must have a public no-args constructor, and additionally may conform to one of the types listed in Chapter 7.

4.2.1. encoders

The **encoders** parameter contains a (possibly empty) list of Java classes that are to act as encoder components for this endpoint. These classes must implement some form of the **Encoder** interface, have public no-arg constructors and be visible within the classpath of the application that this WebSocket endpoint is part of. The implementation must create a new instance of each encoder per connection per endpoint which guarantees no two threads are in the encoder at the same time. When sending an application object using the **RemoteEndpoint** API that is of a type that matches (same class or a sub-class) the parameterized type of the Encoder, the implementation must attempt to encode the object using the matching Encoder[WSC-4.2.1-1].

4.2.2. decoders

The **decoders** parameter contains a (possibly empty) list of Java classes that are to act as decoder components for this endpoint. These classes must implement some form of the Decoder interface, have public no-arg constructors and be visible within the classpath of the application that this WebSocket endpoint is part of. The implementation must create a new instance of each encoder per connection per endpoint. The implementation must attempt to decode WebSocket messages using the first appropriate decoder in the list and pass the message in decoded object form to the WebSocket endpoint [WSC-4.2.2-1]. If the Decoder implementation has the method, the implementation must use the **willDecode()** method on the decoder to determine if the **Decoder** will match the incoming message [WSC-4.2.2-2].

4.2.3. configurator

The optional **configurator** attribute allows the developer to indicate that they would like the WebSocket implementation to use a developer provided implementation of **ClientEndpointConfig.Configurator**. If one is supplied, the WebSocket implementation must use this when configuring the endpoint [4.2.3-1]. The developer may use this technique to share state across all instances of the endpoint in addition to customizing the opening handshake.

4.2.4. subprotocols

The **subprotocols** parameter contains a (possibly empty) list of string names of the subprotocols that this endpoint is willing to support. The implementation must use this list in the opening handshake to negotiate the desired subprotocol to use for the connection it establishes [WSC-4.2.4-1].

4.3. @PathParam

This annotation is used to annotate one or more parameters of methods on an annotated endpoint class decorated with any of the annotations **@OnMessage**, **@OnError**, **@OnOpen**, **@OnClose**. The allowed types for these parameters are String, any Java primitive type, or boxed version thereof. Any other type annotated with this annotation is an error that the implementation must report at deployment time [WSC-4.3-1]. The **value** attribute of this annotation must be present otherwise the implementation must throw an error [WSC-4.3-2]. If the **value** attribute of this annotation matches the variable name of an element of the URI-template used in the **@ServerEndpoint** annotation that annotates this annotated endpoint, then the implementation must associate the value of the parameter it annotates with the value of the path segment of the request URI to which the calling WebSocket frame is connected when the method is called [WSC-4.3-3]. Otherwise, the value of the String parameter annotated by this annotation must be set to **null** by the implementation. The association must follow these rules:

- if the parameter is a **String**, the container must use the value of the path segment [WSC-4.3-4].
- if the parameter is a Java primitive type or boxed version thereof, the container must use the path segment string to construct the type with the same result as if it had used the public one argument String constructor to obtain the boxed type, and reduced to its primitive type if necessary [WSC-4.3-5].

If the container cannot decode the path segment appropriately to the annotated path parameter, then the container must raise an **DecodeException** to the error handling method of the WebSocket containing the path segment [WSC-4.3-6].

For example,

```
@ServerEndpoint("/bookings/{guest-id}")
public class BookingServer {
    @OnMessage
    public void processBookingRequest(
        @PathParam("guest-id") String guestID,
        String message,
        Session session) {
        // process booking from the given guest here
    }
}
```

In this example, if a client connects to this endpoint with the URI **/bookings/JohnSmith**, then the value of the **guestID** parameter will be **"JohnSmith"**.

Here is an example where the path parameter is an Integer:

```
@ServerEndpoint("/rewards/{vip-level}")
public class RewardServer {
    @OnMessage
    public void processReward(
        @PathParam("vip-level") Integer vipLevel,
        String message, Session session) {
        // process reward here
    }
}
```

4.4. @OnOpen

This annotation may be used on certain methods of a Java class annotated with **@ServerEndpoint** or **@ClientEndpoint**. The annotation defines that the decorated method be called whenever a new client has connected to this endpoint. The container notifies the method after the connection has been established [WSC-4.4-1]. The decorated method can only have an optional **Session** parameter, an optional **EndpointConfig** parameter and zero to n parameters (of type **String**, any Java primitive type, or boxed version thereof) annotated with a **@PathParam** annotation as parameters. If the **Session** parameter is present, the implementation must pass in the newly created **Session** corresponding to the new connection [WSC-4.4-2].

Any Java class using this annotation on a method that does not follow these rules, or that uses this annotation on more than one method must not be deployed by the implementation and the error must be reported to the deployer [WSC-4.4-3].

4.5. @OnClose

This annotation may be used on certain methods of a Java class annotated with **@ServerEndpoint** or **@ClientEndpoint**. The annotation defines that the decorated method be called whenever a remote peer is about to be disconnected from this endpoint, whether that process is initiated by the remote peer, by the local container or by a call to **session.close()**. The container notifies the method before the connection is brought down [WSC-4.5-1]. The decorated method can only have optional **Session** parameter, optional **CloseReason** parameter and zero to n parameters (of type **String**, any Java primitive type, or boxed version thereof) annotated with a **@PathParam** annotation as parameters. If the **Session** parameter is present, the implementation must pass in the about-to-be ended **Session** corresponding to the connection [WSC-4.5-2]. If the method itself throws an error, the implementation must pass this error to the **onError()** method of the endpoint together with the session [WSC-4.5-3].

Any Java class using this annotation on a method that does not follow these rules, or that uses this

annotation on more than one method must not be deployed by the implementation and the error must be reported to the deployer [WSC-4.5-4].

4.6. @OnError

This annotation may be used on certain methods of a Java class annotated with **@ServerEndpoint** or **@ClientEndpoint**. The annotation defines that the decorated method be called whenever an error is generated on any of the connections to this endpoint. The decorated method can only have optional **Session** parameter, mandatory **Throwable** parameter and zero to n parameters (of type **String**, any Java primitive type, or boxed version thereof) annotated with a **@PathParam** annotation as parameters. If the **Session** parameter is present, the implementation must pass in the **Session** in which the error occurred to the connection [WSC-4.6-1]. The container must pass the error as the **Throwable** parameter to this method [WSC-4.6-2].

Any Java class using this annotation on a method that does not follow these rules, or that uses this annotation on more than one method must not be deployed by the implementation and the error must be reported to the deployer [WSC-4.6-3].

4.7. @OnMessage

This annotation may be used on certain methods of a Java class annotated with **@ServerEndpoint** or **@ClientEndpoint**. The annotation defines that the decorated method be called whenever an incoming message is received. The method it decorates may have a number of forms for handling text, binary or pong messages, and for sending a message back immediately that are defined in detail in the API documentation for **@OnMessage**.

Any method annotated with **@OnMessage** that does not conform to the forms defied therein is invalid. The WebSocket implementation must not deploy such an endpoint and must raise a deployment error if an attempt is made to deploy such an annotated endpoint [WSC-4.7-1].

If the method uses a class equivalent of a Java primitive as a method parameter to handle whole text messages, the implementation must use the single String parameter constructor to attempt construct the object. If the method uses a Java primitive as a method parameter to handle whole text messages, the implementation must attempt to construct its class equivalent as described above, and then convert it to its primitive value [WSC-4.7-2].

If the method uses a Java primitive as a return value, the implementation must construct the text message to send using the standard Java string representation of the Java primitive. If the method uses a class equivalent of a Java primitive as a return value, the implementation must construct the text message from the Java primitive equivalent as just described [WSC-4.7-3].

Each WebSocket endpoint may only have one message handling method for each of the native WebSocket message formats: text, binary and pong. Any WebSocket endpoint that defines more than one message handling method for any of the native WebSocket message formats is invalid. The WebSocket implementation must not deploy such an endpoint and must raise a deployment error if an

attempt is made to deploy such an annotated endpoint [WSC-4.7-4].

4.7.1. maxMessageSize

The maxMessageSize attribute allows the developer to specify the maximum size of message in bytes that the method it annotates will be able to process, or `-1` to indicate that there is no maximum. The default is `-1`.

If an incoming message exceeds the maximum message size, the implementation must formally close the connection with a close code of `1009` (Too Big) [WSC-4.7.1-1].

4.8. WebSockets and Inheritance

The WebSocket annotation behaviors defined by this specification are not passed down the Java class inheritance hierarchy. They apply only to the Java class on which they are marked. For example, a Java class that inherits from a Java class annotated with class level WebSocket annotations does not itself become an annotated endpoint, unless it itself is annotated with a class level WebSocket annotation. Similarly, subclasses of an annotated endpoint may not use method level WebSocket annotations unless they themselves use a class level WebSocket annotation. Subclasses that override methods annotated with WebSocket method annotations do not obtain WebSocket callbacks unless those subclass methods themselves are marked with a method level WebSocket annotation.

Implementations should not deploy Java classes that mistakenly mix Java inheritance with WebSocket annotations in these ways [WSC-4.8.1].

Implementations that use archive scanning techniques to deploy endpoints on startup must filter out subclasses of annotated endpoints, in addition to other errent endpoint definitions such as annotated classes that are non-public when they build the list of annotated endpoints to deploy [WSC-4.8.2].

Chapter 5. Exception handling and Threading

5.1. Threading Considerations

Implementations of the WebSocket API may employ a variety of threading strategies in order to provide a scalable implementation. The specification aims to allow a range of strategies. However, the implementation must fulfill certain threading requirements in order to provide the developer a consistent threading environment for their applications.

Unless backed by a Jakarta EE component with a different lifecycle (See Chapter 7), the container must use a unique instance of the endpoint per peer [WSC-5.1-1]. In all cases, the implementation must not invoke an endpoint instance with more than one thread per peer at a time [WSC-5.1-2]. The implementation may not invoke the close method on an endpoint until after the open method has completed [WSC-5.1-3].

This guarantees that a WebSocket endpoint instance is never called by more than one container thread at a time per peer [WSC-5.1-4].

If the implementation decides to process an incoming message in parts, it must ensure that the corresponding **onMessage()** calls are called sequentially, and do not interleave either with parts of the same message or with other messages [WSC-5.1.5].

5.2. Error Handling

There are three categories of errors (checked and unchecked Java exceptions) that this specification defines.

5.2.1. Deployment Errors

These are errors raised during the deployment of an application containing WebSocket endpoints. Some of these errors arise as the result of a container malfunction during the deployment of the application. For example, the container may not have sufficient computing resources to deploy the application as specified. In this case, the container must provide an informative error message to the developer during the deployment process [WSC-5.2.1-1]. Other errors arise as a result of a malformed WebSocket application. Chapter 4 provides several examples of WebSocket endpoints that are malformed. In such cases, the container must provide an informative error message to the deployer during the deployment process [WSC-5.2.1-2].

In both cases, a deployment error raised during the deployment process must halt the deployment of the application, any well formed endpoints deployed prior to the error being raised must be removed from service and no more WebSocket endpoints from that application may be deployed by the container, even if they are valid [WSC-5.2.1-3].

If the deployment error occurs under the programmatic control of the developer, for example, when

using the WebSocketContainer API to deploy a client endpoint, deployment errors must be reported by the container to the developer by using an instance of the DeploymentException [WSC-5.2.1-4]. Containers may choose the precise wording of the error message in such cases.

If the deployment error occurs while deployment is managed by the implementation, for example, as a result of deploying a WAR file where the endpoints are deployed by the container as a result of scanning the WAR file, the deployment error must be reported to the deployer by the implementation as part of the container specific deployment process [WSC-5.2.1-5].

If the deployment error occurs while the web application is running (i.e. if the programmatic API is used to add a server endpoint after the deployment phase has completed), the error must be reported to the caller using an instance of the DeploymentException.

5.2.2. Errors Originating in WebSocket Application Code

All errors arising during the functioning of a WebSocket endpoint must be caught by the WebSocket implementation [WSC-5.2.2-1]. Examples of these errors include checked exceptions generated by **Decoders** used by the endpoint and runtime errors generated in the message handling code used by the endpoint. If the WebSocket endpoint has provided an error handling method, either by implementing the **onError()** method in the case of programmatic endpoints, or by using the @OnError annotation in the case of annotated endpoints, the implementation must invoke the error handling method with the error [WSC-5.2.2-2].

If the developer has not provided an error handling method on an endpoint that is generating errors, this indicates to the implementation that the developer does not wish to handle such errors. In these cases, the container must make this information available for later analysis, for example by logging it [WSC-5.2.2-3].

If the error handling method of an endpoint itself is generating runtime errors, the container must make this information available for later analysis [WSC-5.2.2-4].

5.2.3. Errors Originating in the Container and/or Underlying Connection

A wide variety of runtime errors may occur during the functioning of an endpoint. These may including broken underlying connections, occasional communication errors handling incoming and outgoing messages, or fatal errors communicating with a peer. Implementations or their administrators judging such errors to be fatal to the correct functioning of the endpoint may close the endpoint connection, making an attempt to informing both participants using the **onClose()** method. Containers judging such errors to be non-fatal to the correct functioning of the endpoint may allow the endpoint to continue functioning, but must report the error in message processing either as a checked exception returned by one of the send operations, or by delivering a SessionException to the endpoint's error handling method, if present, or by logging the error for later analysis [WSC-5.2.3-1].

Chapter 6. Packaging and Deployment

Jakarta WebSocket applications are packaged using the usual conventions of the Jakarta and Java platforms.

6.1. Client Deployment on JRE

The class files for the WebSocket application and any application resources such as Jakarta WebSocket client applications are packaged as JAR files, along with any resources such as text or image files that it needs.

The client container is not required to automatically scan the JAR file for WebSocket client endpoints and deploy them.

Obtaining a reference to the **WebSocketContainer** using the **ContainerProvider** class, the developer deploys both programmatic endpoints and annotated endpoints using the **connectToServer()** APIs on the **WebSocketContainer**.

6.2. Application Deployment on Web Containers

The class files for the endpoints and any resources they need such as text or image files are packaged into the Jakarta EE-defined WAR file, either directly under **WEB-INF/classes** or packaged as a JAR file and located under **WEB-INF/lib**.

Jakarta EE containers are not required to support deployment of WebSocket endpoints if they are not packaged in a WAR file as described above.

The Jakarta WebSocket implementation must use the web container scanning mechanism defined in Servlet 3.0 to find annotated and programmatic endpoints contained within the WAR file at deployment time [WSC-6.2-1]. This is done by scanning for classes annotated with **@ServerEndpoint** and classes that extend **Endpoint**. See also Section 4.8 for potential extra steps needed after the scan for annotated endpoints. Further, the WebSocket implementation must use the WebSocket scanning mechanism to find implementations of the **ServerApplicationConfig** interface packaged within the WAR file (or in any of its sub-JAR files) [WSC-6.2-2].

If scan of the WAR file locates one or more **ServerApplicationConfig** implementations, the WebSocket implementation must instantiate each of the **ServerApplicationConfig** classes it found. For each one, it must pass the results of the scan of the archive containing it (top level WAR or contained JAR) to its methods [WSC-6.2-4]. The set that is the union of all the results obtained by calling the **getEndpointConfigs()** and **getAnnotatedEndpointClasses()** on the **ServerApplicationConfig** classes (that is to say, the annotated endpoint classes and configuration objects for programmatic endpoints) is the set that the WebSocket implementation must deploy [WSC-6.2-5].

If the WAR file contains no **ServerApplicationConfig** implementations, it must deploy all the annotated endpoints it located as a result of the scan [WSC-6.2-3]. Because programmatic endpoints

cannot be deployed without a corresponding **ServerEndpointConfig**, if there are no **ServerApplicationConfig** implementations to provide these configuration objects, no programmatic endpoints can be deployed.

Note: *This means developers can easily deploy all the annotated endpoints in a WAR file by simply bundling the class files for them into the WAR. This also means that programmatic endpoints cannot be deployed using this scanning mechanism unless a suitable **ServerApplicationConfig** is supplied. This also means that the developer can have precise control over which endpoints are to be deployed from a WAR file by providing one or more **ServerApplicationConfig** implementation classes. This also allows the developer to limit a potentially lengthy scanning process by excluding certain JAR files from the scan (see Servlet 3.0, section 8.2.1). This last case may be desirable in the case of a WAR file containing many JAR files that the developer knows do not contain any WebSocket endpoints.*

6.3. Application Deployment in Standalone WebSocket Server Containers

This specification recommends standalone WebSocket server containers (i.e. those that do not include a Servlet container) locate any WebSocket server endpoints and **ServerApplicationConfig** classes in the application bundle and deploy the set of all the server endpoints returned by the configuration classes. However, standalone WebSocket server containers may employ other implementation techniques to deploy endpoints if they wish.

6.4. Programmatic Server Deployment

This specification also defines a mechanism for deployment of server endpoints that does not depend on Servlet container scanning of the application. Developers may deploy server endpoints programmatically by using one of the **addEndpoint** methods of the **ServerContainer** interface.

Versions of this specification before 2.1 restricted the programmatic deployment of server endpoints to the application deployment phase of an application. As of version 2.1, this restriction no longer applies.

When running on the web container, the **addEndpoint** methods may be called from a **jakarta.servlet.ServletContextListener** provided by the developer and configured in the deployment descriptor of the web application. The WebSocket implementation must make the **ServerContainer** instance corresponding to this application available to the developer as a **ServletContext** attribute registered under the name **jakarta.websocket.server.ServerContainer**.

When running on a standalone container, the developer will need to utilize whatever proprietary hooks the particular container has to offer to make a **ServerContainer** instance available to the developer.

It is recommended that developers use either the programmatic deployment API, or base their application on the scanning and **ServerApplicationConfig** mechanism, but not mix both methods. Developers can suppress a deployment by scan of the endpoints in the WAR file by providing a

ServerApplicationConfig that returns empty sets from its methods.

If however, the developer does mix both modes of deployment, it is possible in the case of annotated endpoints, for the same annotated endpoint to be submitted twice for deployment, once as a result of a scan of the WAR file, and once by means of the developer calling the programmatic deployment API. In this case of an attempt to deploy the same annotated endpoint class more than once, the WebSocket implementation must only deploy the annotated endpoint once, and ignore the duplicate submission.

6.5. WebSocket Server Paths

WebSocket implementations that include server functionality must define a root or the URI space for WebSockets. Called the the WebSocket root, it is the URI to which all the relative WebSocket paths in the same application are relative. If the WebSocket server does not include the Servlet API, the WebSocket server may choose WebSocket root itself. If the WebSocket server includes the Jakarta Servlet API, the WebSocket root must be the same as the Servlet context root of the web application [WSC-6.4-1].

6.6. Platform Versions

The minimum versions of the platforms are:

- Java SE version 8, for the Jakarta WebSocket Client API [WSC-6.5-1].
- Jakarta EE version 9, for the Jakarta WebSocket Server API [WSC-6.5-2].

Chapter 7. Jakarta EE Environment

7.1. Jakarta EE Environment

When supported on the Jakarta EE platform, there are some additional requirements to support WebSocket applications.

7.1.1. WebSocket Endpoints and Dependency Injection

WebSocket endpoints running in the Jakarta EE platform must have full dependency injection support as described in the Jakarta Contexts and Dependency Injection specification (Jakarta CDI Team, 2020). WebSocket implementations part of the Jakarta EE platform are required to support field, method, and constructor injection using the `jakarta.inject.Inject` annotation into all WebSocket endpoint classes, as well as the use of interceptors for these classes [WSC-7.1.1-1]. The details of this requirement are laid out in the Jakarta EE Platform Specification (Jakarta EE Platform Team, 2020), section EE.5.2.5, and a useful guide for implementations to meet the requirement is location in section EE.5.24.

7.2. Relationship with Http Session and Authenticated State

It is often useful for developers who embed WebSocket server endpoints into a larger web application to be able to share information on a per client basis between the web resources (JSPs, JSFs, Servlets for example) and the WebSocket endpoints servicing that client. Because WebSocket connections are initiated with an http request, there is an association between the HttpSession under which a client is operating and any WebSockets that are established within that **HttpSession**. The API allows access in the opening handshake to the unique **HttpSession** corresponding to that same client [WSC-7.2-1].

Similarly, if the opening handshake request is already authenticated with the server, the opening handshake API allows the developer to query the user **Principal** of the request. If the connection is established with the requesting client, the WebSocket implementation considers the user **Principal** for the associated WebSocket **Session** to be the user **Principal** that was present on the opening handshake [WSC-7.2-2].

In the case where a WebSocket endpoint is a protected resource in the web application (see Chapter 8), that is to say, requires an authorized user to access it, then the WebSocket implementation must ensure that the WebSocket endpoint does not remain connected to its peer after the underlying implementation has decided the authenticated identity is no longer valid [WSC-7.2-3]. This may happen, for example, if the user logs out of the containing web application, or if the authentication times out or is invalidated for some other reason. In this situation, the WebSocket implementation must immediately close the connection using the WebSocket close status code 1008 [WSC-7.2-3].

On the other hand, if the WebSocket endpoint is not a protected resource in the web application, then the user identity under which an opening handshake established the connection may become invalid

or change during the operation of the WebSocket without the WebSocket implementation needing to close the connection.

Chapter 8. Server Security

WebSocket endpoints are secured using the web container security model. The goal of this is to make it easy for a WebSocket developer to declare whether access to a WebSocket server endpoint needs to be authenticated, who can access it, and if it needs an encrypted connection or not. A WebSocket which is mapped to a given **ws://** URI (as described in Chapter 3 and Chapter 4) is protected in the deployment descriptor with a listing to a **http://** URI with same hostname, port and path since this is the URL of its opening handshake. Accordingly, WebSocket developers may assign an authentication scheme, user roles granted access and transport guarantee to their WebSocket endpoints.

8.1. Authentication of Websockets

This specification does not define a mechanism by which WebSockets themselves can be authenticated. Rather, by building on the Servlet defined security mechanism, a WebSocket that requires authentication must rely on the opening handshake request that seeks to initiate a connection to be previously authenticated. Typically, this will be performed by an HTTP authentication (perhaps basic or form-based) in the web application containing the WebSocket prior to the opening handshake to the WebSocket.

If a client sends an unauthenticated opening handshake request for a WebSocket that is protected by the security mechanism, the WebSocket implementation must return a **401 (Unauthorized)** response to the opening handshake request and may not initiate a WebSocket connection [WSC-8.1-1].

8.2. Authorization of Websockets

A WebSocket's authorization may be set by adding a <**security-constraint**> element to the **web.xml** of the web application in which it is packaged. The <**url-pattern**> used in the security constraint must be used by the container to match the request URI of the opening handshake of the WebSocket [WSC-8.2-1]. The implementation must interpret any http-method other than GET (or the default, missing) as not applying to the WebSocket [WSC-8.2-2].

8.3. Transport Guarantee

A transport guarantee of **NONE** must be interpreted by the container as allowing unencrypted **ws://** connections to the WebSocket [WSC-8.3-1]. A transport guarantee of **CONFIDENTIAL** must be interpreted by the implementation as only allowing access to the WebSocket over an encrypted (**wss://**) connection [WSC-8.3-2]. This may require a pre-authenticated request.

8.4. Example

This example snippet from a larger web.xml deployment descriptor shows a security constraint for a WebSocket endpoint. In the example, the WebSocket endpoint which matches on an incoming request URI of **'quotes/live'** relative to the context root of the containing web application is protected such that

it may only be accessed using **wss://**, and is available only to authenticated users who belong either to the **GOLD_MEMBER** or **PLATINUM_MEMBER** roles.

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>
            LiveQuoteWebSocket
        </web-resource-name>
        <description>
            Security constraint for
            live quote WebSocket endpoint
        </description>
        <url-pattern>/quotes/live</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <description>
            definition of which roles
            may access the quote endpoint
        </description>
        <role-name>GOLD_MEMBER</role-name>
        <role-name>PLATINUM_MEMBER</role-name>
    </auth-constraint>
    <user-data-constraint>
        <description>WSS required</description>
        <transport-guarantee>
            CONFIDENTIAL
        </transport-guarantee>
    </user-data-constraint>
</security-constraint>
```

Appendix A: Changes

This appendix lists the changes in the WebSocket specification. This appendix is non-normative.

A.1. Changes Between 2.1 and 2.0

- Issue 190 and Issue 192 Clarify that once the container has identified a MessageHandler for a message, the MessageHandler is used for the entirety of the message irrespective of any subsequent changes to the MessageHandlers configured for the Session.
- Issue 207 Add a getter for the default platform configurator.
- Issue 210 Provide an API for client-side TLS configuration.
- Issue 211 Remove the restriction that, in a Jakarta web container environment, endpoints can only registered during the deployment of the web application. Also add a new method, `ServerContainer.upgradeHttpToWebSocket()` that allows a web application to programmatically dispatch a request to a WebSocket endpoint.
- Issue 228 Clarify the expected behaviour for `Session.getRequestURI()`. The full URI should be returned.
- Issue 235 Clarify the expected handling of user properties.
- Issue 382 Clarify that a zero or negative value disables the session idle timeout and improve the language used in the Javadoc for the other timeouts.
- Removed the copy of the `jakarta.websocket.*` classes from the `jakarta.websocket-api` jar and replaced the copy with a dependency on the `jakarta.websocket-client-api` jar.
- Added JPMS module descriptors that define the client module name as `jakarta.websocket.client` and the server module name as `jakarta.websocket` with the server module depending on the client module.

A.2. Changes Between 2.0 and JSR-356

- Pull Request 312 Convert from `javax.*` to `jakarta.*`.
- Pull Request 315 Update specification document for move to Jakarta EE plus a large number of smaller tweaks and editorial improvements.

Bibliography

[1] I. Fette and A. Melnikov. RFC 6455: The WebSocket Protocol. RFC, IETF, December 2011. See http://www.ietf.org/rfc/rfc6455.txt.

[2] Ian Hickson. The WebSocket API. Note, W3C, December 2012. See http://dev.w3.org/html5/websockets/.

[3] S. Bradner. RFC 2119: Keywords for use in RFCs to Indicate Requirement Levels. RFC, IETF, March 1997. See http://www.ietf.org/rfc/rfc2119.txt.

[4] J. Gregorio, R. Fielding, M. Hadley, M. Nottingham, and D. Orchard. RFC 6570: URI Template. RFC, IETF, March 2012. See http://www.ietf.org/rfc/rfc6570.txt.

[5] Jakarta CDI Team. Jakarta Contexts and Dependency Injection 3.0. Eclipse Foundation, 2020. See https://jakarta.ee/specifications/cdi/3.0/.

[6] Jakarta EE Platform Team. Jakarta EE Platform Specification 9. Eclipse Foundation, 2020. See https://jakarta.ee/specifications/platform/9/.

[7] T. Berners-Lee, R. Fielding and L. Masinter. RFC 3986: Uniform Resource Identifier (URI): Generic Syntax, IETF, January 2005. See https://tools.ietf.org/rfc/rfc3986.txt