

Java API for Web Socket

Early Draft Review: version 006

September 21st 2012

Danny Coward

Comments to: users@websocket-spec.java.net

Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065

JSR-000356 Java API for Web Socket 1.0 Early Draft Review

ORACLE IS WILLING TO LICENSE THIS SPECIFICATION TO YOU ONLY UPON THE CONDITION THAT YOU ACCEPT ALL OF THE TERMS CONTAINED IN THIS LICENSE AGREEMENT ("AGREEMENT"). PLEASE READ THE TERMS AND CONDITIONS OF THIS AGREEMENT CAREFULLY. BY DOWNLOADING THIS SPECIFICATION, YOU ACCEPT THE TERMS AND CONDITIONS OF THIS AGREEMENT. IF YOU ARE NOT WILLING TO BE BOUND BY THEM, SELECT THE "DECLINE" BUTTON AT THE BOTTOM OF THIS PAGE AND THE DOWNLOADING PROCESS WILL NOT CONTINUE.

Specification: JSR-356 Java(tm) API for Web Socket ("Specification")
 Version: 1.0
 Status: Early Draft Review
 Release: 21 September 2012
 Copyright 2012 Oracle Corporation
 500 Oracle Parkway, Redwood Shores, California 94065, U.S.A
 All rights reserved.

NOTICE

The Specification is protected by copyright and the information described therein may be protected by one or more U.S. patents, foreign patents, or pending applications. Except as provided under the following license, no part of the Specification may be reproduced in any form by any means without the prior written authorization of Oracle America, Inc. ("the Specification Lead") and its licensors, if any. Any use of the Specification and the information described therein will be governed by the terms and conditions of this Agreement.

Subject to the terms and conditions of this license, including your compliance with Paragraphs 1 and 2 below, the Specification Lead hereby grants you a fully-paid, non-exclusive, non-transferable, limited license (without the right to sublicense) under the Specification Lead's intellectual property rights to:

1. Review the Specification for the purposes of evaluation. This includes:

- (i) developing implementations of the Specification for your internal, non-commercial use;
- (ii) discussing the Specification with any third party; and
- (iii) excerpting brief portions of the Specification in oral or written communications which discuss the Specification provided that such excerpts do not in the aggregate constitute a significant portion of the Technology.

2. Distribute implementations of the Specification to third parties for their testing and evaluation use, provided that any such implementation:

- (a) does not modify, subset, superset or otherwise extend the Licensor Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the Licensor Name Space other than those required/authorized by the Specification or Specifications being implemented;
- (b) is clearly and prominently marked with the word "UNTESTED" or "EARLY ACCESS" or "INCOMPATIBLE" or "UNSTABLE" or "BETA" in any list of available builds and in proximity to every link initiating its download, where the list or link is under Licensee's control; and
- (c) includes the following notice: "This is an implementation of an early-draft specification developed under the Java Community Process (JCP) and is made available for testing and evaluation purposes only. The code is not compatible with any specification of the JCP."

The grant set forth above concerning your distribution of implementations of the specification is contingent upon your agreement to terminate development and distribution of your "early draft" implementation as soon as feasible following final completion of the specification. If you fail to do so, the foregoing grant shall be considered null and void.

No provision of this Agreement shall be understood to restrict your ability to make and distribute to third parties applications written to the Specification. Other than this limited license, you acquire no right, title or interest in or to the Specification or any other intellectual property of the Specification Lead, and the Specification may only be used in accordance with the license terms set forth herein. This license will expire on the earlier of: (a) two (2) years from the date of Release listed above; (b) the date on which the final version of the Specification is publicly released; or (c) the date on which the Java Specification Request (JSR) to which the Specification corresponds is withdrawn. In addition, this license will terminate immediately without notice from the Specification Lead if you fail to comply with any provision of this license. Upon termination, you must cease use of or destroy the Specification.

"Licensor Name Space" means the public class or interface declarations whose names begin with "java", "javax", "com.oracle" or their equivalents in any subsequent naming convention adopted by Oracle through the Java Community Process, or any recognized successors or replacements thereof.

TRADEMARKS

No right, title, or interest in or to any trademarks, service marks, or trade names of Oracle or Oracle's licensors is granted hereunder. Oracle, the Oracle logo, Java are trademarks or registered trademarks of Oracle USA, Inc. in the U.S. and other countries.

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED "AS IS" AND IS EXPERIMENTAL AND MAY CONTAIN DEFECTS OR DEFICIENCIES WHICH CANNOT OR WILL NOT BE CORRECTED BY THE SPECIFICATION LEADS. THE SPECIFICATION LEADS MAKE NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE OR THAT ANY PRACTICE OR IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADE SECRETS OR OTHER RIGHTS. This document does not represent any commitment to release or implement any portion of the Specification in any product. THE SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION THEREIN; THESE CHANGES WILL BE INCORPORATED INTO NEW VERSIONS OF THE SPECIFICATION, IF ANY. THE SPECIFICATION LEADS MAY MAKE IMPROVEMENTS AND/OR CHANGES TO THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THE SPECIFICATION AT ANY TIME. Any use of such changes in the Specification will be governed by the then-current license for the applicable version of the Specification.

LIMITATION OF LIABILITY

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL THE SPECIFICATION LEADS AND/OR THEIR LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO ANY FURNISHING, PRACTICING, MODIFYING OR ANY USE OF THE SPECIFICATION, EVEN IF ORACLE AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will hold the Specification Lead (and its licensors) harmless from any claims based on your use of the Specification for any purposes other than the limited right of evaluation as described above, and from any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND

If this Software is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions)

REPORT

You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your evaluation of the Specification ("Feedback"). To the extent that you provide the Specification Lead with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant the Specification Lead a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.

GENERAL TERMS

Any action related to this Agreement will be governed by California law and controlling U.S. federal law. The U.N. Convention for the International Sale of Goods and the choice of law rules of any jurisdiction will not apply. The Specification is subject to U.S. export control laws and may be subject to export or import regulations in other countries. Licensee agrees to comply strictly with all such laws and regulations and acknowledges that it has the responsibility to obtain such licenses to export, re-export or import as may be required after delivery to Licensee. This Agreement is the parties' entire agreement relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, conditions, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification to this Agreement will be binding, unless in writing and signed by an authorized representative of each party.

Introduction	7
<i>Purpose of this document</i>	7
<i>Goals of the specification</i>	8
<i>Terminology used throughout the specification</i>	8
<i>Specification Conventions</i>	8
<i>Expert group members</i>	9
<i>Acknowledgements</i>	9
Applications	10
<i>API overview</i>	10
<i>Endpoint lifecycle</i>	10
<i>Sessions</i>	10
<i>Receiving Messages</i>	11
<i>Sending Messages</i>	11
<i>Clients and Servers</i>	12
<i>Endpoints using WebSocket Annotations</i>	12
<i>POJO as Endpoint</i>	12
<i>Websocket lifecycle</i>	13
<i>Handling messages</i>	13
Configuration	14
<i>Server Configurations</i>	14
<i>URL mapping</i>	14
<i>Subprotocol negotiation</i>	14
<i>Extension Modification</i>	14
<i>Origin Check</i>	15

<i>Handshake Modification</i>	15
<i>Client Configuration</i>	15
<i>URI</i>	15
<i>Subprotocols</i>	15
<i>Extensions</i>	16
Annotations	17
<i>@WebSocketEndpoint</i>	17
<i>value</i>	17
<i>encoders</i>	18
<i>decoders</i>	18
<i>subprotocols</i>	18
<i>@WebSocketPathParam</i>	19
<i>@WebSocketOpen</i>	19
<i>@WebSocketClose</i>	20
<i>@WebSocketError</i>	20
<i>@WebSocketMessage</i>	20
Extensions	22
Packaging and Deployment	23
<i>Client Deployment on JDK</i>	23
<i>Server Deployment on Java EE</i>	23
<i>Platform Versions</i>	23
Java EE Environment	24
<i>Threading considerations</i>	24
<i>Bootstrapping Container Implementations</i>	24

<i>Java EE Environment</i>	25
<i>Creating Endpoints using Java EE components</i>	25
<i>Injectable components</i>	25
<i>Relationship with Http Session</i>	26
Server Security	27
Bibliography	29

1. Introduction

This specification defines a set of Java APIs for the development of web socket 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 [1]
- The Web Socket API for JavaScript [2]

1.1. Purpose of this document

This document in combination with the API documentation for the Java WebSocket API [link] is the specification of the Java WebSocket API. The specification defines the requirements that an implementation must meet in order to be an implementation of the Java WebSocket API. This specification has been developed under the rules of the Java Community Process. Together with the Test Compatibility Kit (TCK) which tests that a given implementation meets the requirements of the specification, and Reference Implementation (RI) that implements this specification and which passes the TCK, this specification defines the Java standard for WebSocket application development. While there is much useful information in this document for developers using the Java 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 Java 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

TODO: here are the goals

1.3. Terminology used throughout the specification

endpoint

connection

peer

session

client

server

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[10].

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 in figure 1.1: Figure 1.1:

Example Java Code

```

1 package com.example.hello;
2
3 public class Hello {
4     public static void main(String args[]) {
5         System.out.println("Hello World");
6     }
7 }
```

URIs of the general form ‘[http://example.org/...](http://example.org/)’ and ‘[http://example.com/...](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. Expert group members

This specification was developed in the Java Community Process as part of JSR 356 [link]. It is the result of the collaborative work of the members of the JSR 356 expert group. The full public mail archive can be found here. [link] The following are the expert group members:-

Jean-Francois Arcand (Individual Member)

Scott Ferguson (Caucho Technology, Inc)

Joe Walnes (DRW Holdings, LLC)

Minehiko IIDA (Fujitsu Limited)

Wenbo Zhu (Google Inc.)

Bill Wigger (IBM)

Justin Lee (Individual Member)

Danny Coward (Oracle)

Rémy Maucherat (RedHat)

Moon Namkoong (TmaxSoft, Inc.)

Mark Thomas (VMware)

Wei Chen (Voxeo Corporation)

1.6. Acknowledgements

During the development of this specification we received many review comments, feedback and suggestions. Thanks in particular to:

Jitendra Kotamraju, Martin Matula, Stepan Kopriva, Jondean Healey (and more).

2. Applications

Java 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 Java WebSocket API in order to implement the required behavior to handle the endpoint lifecycle, consume and send messages and publish itself, or connect to a peer. The second means is to decorate a Plain Old Java Object (POJO) with certain of the annotations that are part of the Java WebSocket API. The implementation then takes these annotated classes and creates the appropriate objects at runtime to deploy the POJO as a websocket endpoint.

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

2.1. API overview

This section gives a brief overview of the Java 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 Java WebSocket API as an instance of the Endpoint class. Developers may subclass the Endpoint class in order to intercept lifecycle events of the endpoint: those of a peer connecting, a peer disconnecting and an error being raised during the lifetime of the endpoint.

The implementation must use at least one instance of the Endpoint class to represent the logical endpoint. Each instance of the Endpoint class may therefore handle connections to the endpoint from multiple peers. Some implementations may wish to use multiple instances of Endpoint to represent a logical endpoint, perhaps one instance per VM in a distributed server setting.

2.1.2. Sessions

The Java 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 web socket messages between the endpoint and peer, followed by a close notification or error terminal to the connection. For each peer that is interacting with an endpoint, there is one Session instance that represents that interaction. The Session instance corresponding to the peer is passed to the Endpoint instance representing the endpoint at the key events in its lifecycle.

2.1.3. Receiving Messages

The Java WebSocket API presents a variety of means for an endpoint to receive messages from its peers. Developers implement the subtype of the MessageHandler interface that suits 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.

2.1.4. Sending Messages

The Java WebSocket API models each peer in a session with an endpoint as an instance of the RemoteEndpoint class. This class contains a variety of methods for sending web socket messages from the endpoint to its peer.

Example

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 is repeated, first using only the API classes:-

```
public class HelloServer extends Endpoint {
    @Override
    public void onOpen(Session session) {
        final RemoteEndpoint remote = session.getRemote();
        session.addMessageHandler(new MessageHandler.Text() {
            public void onMessage(String text) {
                try {
                    remote.sendString("Got your message (" + text + ").
Thanks !");
                } catch (IOException ioe) {
                    ioe.printStackTrace();
                }
            }
        });
    }
}
```

The second using only the annotations in the API:-

```
@WebSocketEndpoint("/hello")
public class MyHelloServer {
    @WebSocketMessage
    public String doListen(String message) {
        return "Got your message (" + text + "). Thanks !";
    }
}
```

2.1.5. Clients and Servers

The websocket protocol is a two-way protocol. Once established, the web socket protocol is symmetrical between the two parties in the conversation. The difference between a websocket ‘client’ and a websocket ‘server’ is 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.

2.2. Endpoints using WebSocket Annotations

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

2.2.1. POJO as Endpoint

The class level `@WebSocketEndpoint` annotation indicates that a Java class is to become a websocket endpoint at runtime. Developers may use the path attribute to specify a URL mapping for the endpoint. The encoders and decoders attribute allow the developer to specify classes that encode application objects into web socket messages, and decode web socket messages into application objects.

2.2.2. Websocket lifecycle

The method level `@WebSocketOpen` and `@WebSocketClose` annotations allow the developers to decorate methods on their `@WebSocketEndpoint` annotated Java class to specify that they be called when the resulting endpoint receives a new connection from a peer or when a connection from a peer is closed respectively.

2.2.3. Handling messages

In order that the websocket POJO can process incoming messages, the method level `@WebSocketMessage` annotation allows the developer to indicate which methods the implementation should call when a message is received.

3. Configuration

WebSocket applications are configured with a number of key parameters: the URL mapping that identifies a web socket endpoint in the URI-space of the container, the subprotocols that the endpoint supports, 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.

[WSC-1] 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. Here follows the definition of the server-specific and client-specific

3.1. Server Configurations

In order to deploy a websocket Endpoint instance, the container requires an EndpointConfiguration instance. The WebSocket API provides default implementations of this interface, for the client, and for the server. [WSC-2] These configuration objects employ default policies for negotiating the opening handshake that will establish (or not) each web socket connection.

3.1.1. URL mapping

[WSC-3] The URL mapping policy of the default server configuration is such that a URI of the opening handshake matches the URI of an endpoint if and only if the match is exact. [WSC-4] The implementation must not establish the connection unless there is a match.

3.1.2. Subprotocol negotiation

The default server configuration must be provided a list of supported protocols in order of preference at creation time. [WSC-5] 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.

3.1.3. Extension Modification

In the opening handshake, the client supplies a list of extensions that it would like to use. [WSC-6] The default server configuration selects from those extensions the ones it supports, and places them in the same order as requested by the client.

3.1.4. Origin Check

[WSC-7] The default server configuration makes a check of the hostname provided in the Origin header, failing the handshake if the hostname cannot be verified.

3.1.5. Handshake Modification

[WSC-8] The default server configuration makes no modification of the opening handshake process other than that described above.

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 `ClientEndpointConfiguration` and `ServerEndpointConfiguration`, either by implementing the appropriate `*Configuration` interfaces, or subclassing the default implementations thereof.

For example, they 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 override the `modifyHandshake` method on the `ServerConfiguration`, wherein they have full access to the `HttpRequest` and `HttpResponse` of the handshake.

They may wish to provide a more sophisticated method for mapping URIs to endpoints, such as the URI-template based scheme defined by the web socket annotations see Chapter 4 Annotations).

3.2. Client Configuration

In order to connect a websocket endpoint to its partner websocket server, the implementation requires configuration information. Aside from the list of encoders and decoders, the Java `WebSocket` API needs the following attributes:-

3.2.1. URI

The default client configuration uses the developer provided URI to initiate the opening handshake it uses to establish a web socket session. [WSC-9]

3.2.2. 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. [WSC-10]

3.2.3. Extensions

The default client configuration uses the developer provided list of extensions to send, in order of preference, the extensions, including parameters, that it would like to use [WSC-11].

4. Annotations

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

4.1. @WebSocketEndpoint

This class level annotation signifies that the Java class it decorates is to be deployed as a WebSocket endpoint. The container must raise a deployment error if this annotation is located anywhere other than at the class level [WSC-12]. The class must have a public no-args constructor, and additionally may conform to one of the types listed in the Java EE Environment chapter.

4.1.1. value

The value parameter 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 [6]. The implementation uses value to deploy the endpoint to the URI space of the web socket implementation. The implementation treats the value as relative to the root URI of the web socket implementation in determining a match against the request URI of an incoming opening handshake request. If the value is a partial URI, the request URI matches if and only if the match is exact. If the value is a URI template (level-1) the request URI matches if and only if the request URI is an expanded version of the URI-template. The value attribute is mandatory; the implementation must reject a missing or malformed path at deployment time [WSC-14].

For example,

```
@WebSocketEndpoint("/bookings/{guest-id}");
public class BookingServer {

    @WebSocketMessage
    public void processBookingRequest(@WebSocketPathParam("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 `@WebSocketEndpoint("/bookings/SallyBrown")`, then only a client request to `/bookings/SallyBrown` would be able to connect to this web socket endpoint.

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

TBD We may explore allowing higher levels of path templates in future drafts.

TBD Specification of this as a server only attribute. Also exploration of an equivalent client side attribute: `connectURI`: the uri this (client) endpoint will connect to.

4.1.2. 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, and have public no-arg constructors and be visible within the classpath of the application that this websocket endpoint is part of. The implementation must attempt to encode application objects of matching parametrized type as the encoder when they are attempted to be sent using the `RemoteEndpoint` API [WSC-15]. The implementation must use the first encoder that matches in the list [WSC-16].

4.1.3. 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, and have public no-arg constructors and be visible within the classpath of the application that this websocket endpoint is part of. The implementation must attempt to decode web socket messages using the first appropriate decoder in the list and pass the message in decoded object form to the websocket endpoint [WSC-17]. The implementation uses the `willDecode()` method on the decoder to determine if the `Decoder` will match the incoming message [WSC-18]

TBD Specify lifecycle of encoders/decoders new instance per...endpoint...message ?

TBD Does the implementation match multiple methods annotated with `@WebSocketMessage` ?

4.1.4. subprotocols

The `subprotocols` parameter contains a (possibly empty) list of string names of the sub protocols 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-19].

TBD: Perhaps add the ability to specify extensions in the annotation as well ?

4.2. @WebSocketPathParam

This annotation may be used to annotate one or more String parameters of methods on a POJO endpoint decorated with any of the annotations `@WebSocketMessage`, `@WebSocketError`, `@WebSocketOpen`, `@WebSocketClose`. The value attribute of this annotation must be present otherwise the implementation must throw an error. If the value attribute of this annotation matches the variable name of an element of the URI-template used in the `@WebSocketEndpoint` annotation that annotates this POJO endpoint, then the implementation must set the value of the parameter it annotates to the value of the path segment of the request URI to which the calling web socket frame is connected when the method is called. Otherwise, the value of the String parameter annotated by this annotation is null.

For example,

```
@WebSocketEndpoint("/bookings/{guest-id}");
public class BookingServer {

    @WebSocketMessage
    public void processBookingRequest(@WebSocketPathParam("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"`.

4.3. @WebSocketOpen

This annotation is a method level annotation. The implementation must reject at deployment time any Java class using this annotation anywhere except at this level [WSC-20]. 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-21]. The decorated method can only have optional `Session` parameter and zero to `n` String parameters annotated with a `@WebSocketPathSegment` annotation as parameters. If the `Session` parameter is present, the implementation must pass in the newly created `Session` corresponding to the new connection [WSC-22]. If the method throws an error, the implementation must pass this error to the `onError` method of the endpoint together with the session [WSC-23].

(**TBD** does this kill the session?).

4.3.1. @WebSocketClose

This annotation is a method level annotation. The implementation must reject at deployment time any Java class using this annotation anywhere except at this level [WSC-24]. The annotation defines that the decorated method be called whenever a new client is about to be disconnected from this endpoint. The container notifies the method before the connection is brought down [WSC-25]. The decorated method can only have optional Session parameter and zero to n String parameters annotated with a @WebSocketPathSegment 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-26]. If the method throws an error, the implementation must pass this error to the onError method of the endpoint together with the session [WSC-27].

4.3.2. @WebSocketError

This annotation is a method level annotation. The implementation must reject at deployment time any Java class using this annotation anywhere except at this level [WSC-24]. 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 String parameters annotated with a @WebSocketPathSegment 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-XX]. The container must pass the error as the Throwable parameter to this method.

4.3.3. @WebSocketMessage

This annotation is a method level annotation. The implementation must reject at deployment time any Java class using this annotation anywhere except at this level [WSC-28]. 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

- The first String parameter in its parameter list must be called by the container using the String form of the incoming message, if in text form [WSC-29].
- The first byte parameter in its parameter list must be called by the container using the byte array form of the incoming message, if in binary form [WSC-30].
- If the parameter list contains a Session parameter, the implementation must use the Session object corresponding to the connection on which the message arrived [WSC-31].

- The method may have zero to n String parameters annotated with `@WebSocketPathParameter`.
- The method may or may not have a return type. If the method has a return type, the implementation must treat this return object as a web socket message to be immediately sent back to the sender of the incoming message [WSC-32]. It uses the usual mechanism of checking its supply of encoders in order to handle return types other than String or byte[].

TBD Future drafts of this specification may include ways to map methods with

- other messaging parameter types: String fragments, ByteBuffers, streams.
- other message level attributes

This will include attributes on this annotation and specification of more kinds of methods and parameters that can be decorated with the `@WebSocketMessage` annotation.

5. Extensions

The web socket protocol defines a mechanism by which the web socket protocol may be extended. Common examples include web socket extensions to compress data, and to multiplex web socket frame transmission.

Because applications may wish to define their own extensions, this specification includes an API to intercept the frame-level message transmission in the web socket implementation in order to customise the frame transmission interaction and to customize the frames themselves.

TBD Some requests to provide extension implementation of compression and maybe MUX.

An extension is represented by the Extension class. Each class has a name and set of extension parameters that identifies the extension. This representation is used for example during the opening handshake.

Each Extension acts as a factory of FrameHandler objects. The FrameHandler intercepts each frame that is either transmitted by the implementation or received from a peer by the implementation. During this interception, the FrameHandler for the extension has the opportunity to modify the Frame, or to coalesce or multiply the frames before passing the resulting Frame or Frames back to the implementation.

Note. There is no specified mapping between the Frames used at the extension level and the fragmented messages that are processed at the developer API, e.g. `MessageHandler.AsyncText.onMessagePart(String fragment, boolean isLast)`. The implementation may choose how to transform an incoming message in the form of partial text frames into the fragments it passes into this `MessageHandler`.

Extensions may be packaged along with the applications that use them as described in Chapter 6 [WSC-33].

6. Packaging and Deployment

Java WebSocket applications are packaged using the usual conventions of the Java Platform

6.1. Client Deployment on JDK

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

TBD Further API for client deployment ?

6.2. Server Deployment on Java EE

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

The Java WebSocket implementation must recognize any web socket endpoints contained within the WAR file at deployment time and correctly generate the corresponding configurations and endpoint instances such that it can handle incoming websocket requests from clients as soon as the web container is fully initialized.

[WSC-34]

Web Socket applications that contain endpoints that are created not using annotations but by extending the WebSocket API may use existing techniques in the web container to initialize and shut down their endpoints and configuration. For example, they may create a ServletContextListener to initiate the deployment of their Endpoint implementations at web container startup time.

TBD May look into providing more application lifecycle for non-POJO websocket endpoints in the web container.

6.3. Platform Versions

The minimum versions of the Java platforms are:

- Java SE version 7, for the Java WebSocket client API [WSC-35].
- Java EE version 6, for the Java WebSocket server API [WSC-36].

7. Java EE Environment

7.1. 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.

Each endpoint may receive multiple, concurrent lifecycle events (open, close). The container may use multiple threads to invoke the corresponding lifecycle methods on an endpoint [WSC-37]. The container may not invoke the close method on an endpoint until the open method has either completed, or the container has determined that it will not wait until it has completed and has removed it from service [WSC-39].

Each endpoint may receive messages from multiple peers. Developers use instances of the MessageHandler interface in order to process these messages.

The container may not use more than one thread at a time to notify the endpoint of an incoming message from the same peer [WSC-40].

If an incoming messages is received in fragments, the container must ensure that the streaming onMessage calls are called sequentially, and do not interleave [WSC-41].

TBD: Threading requirements on lifecycle events versus message handling callbacks.

TBD: Need to ensure that container created callback threads for onMessage, versus callback threads for (async) sends.

7.2. Bootstrapping Container Implementations

The Web Socket API allows different container implementations to be bootstrapped into the runtime.

In the client case, the WebSocket API has a ContainerProvider class that uses the service loaded mechanism of the Java SE platform to load suitable providers into the runtime.

In the Java EE environment, it is TBD how the bootstrapping will occur. We may define the ServerContainer to be a CDI managed bean with Application scope, thereby allowing a single instance to be injected into developer code, for example into a ServletContainer initializer. Or we may define an annotation to mark a web socket container implementation that the web container must scan for for at startup and load an instance thereof.

7.3. Java EE Environment

When supported on the Java EE platform, there are some additional requirements to support Web Socket applications.

7.3.1. Creating Endpoints using Java EE components

In addition to the ability to apply the WebSocket annotations to POJOs, the WebSocket implementation must support the creation of WebSocket endpoints using the following Java EE components [WSC-42]:

- a stateless session EJB
- a singleton EJB
- a CDI managed bean

7.3.2. Injectable components

A Java EE WebSocket implementation must support the injection of certain objects into web socket endpoints as associated objects [WSC-43]

Into web socket API endpoints and websocket POJO endpoints:

- `@ServletContext` - injects the ServletContext (Servlet API) of the web container where the endpoint is deployed.
- `@ServerContainer` - injects the ServerContainer (WebSocket API) into the endpoint.
- `@EndpointConfiguration` - injects the (Server)EndpointConfiguration (WebSocket API) into the endpoint.
- `@ActiveSessions` - injects the list of active web socket Sessions (WebSocket API) into the endpoint

Into a MessageHandler

- `@Session` - injects the current Session (WebSocketAPI) into the message handler.
- `@RemoteEndpoint` - injects the current RemoteEndpoint (WebSocketAPI) into the MessageHandler.

- **TBD** The above section lists the object we wish to inject. The annotation names are illustrative until we investigate how best to use the existing Java EE CDI framework techniques to inject them

7.4. Relationship with Http Session

It is often useful for developers who embed web socket applications 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 web socket endpoints servicing that client. Because web socket connections are initiated with an Http Request, there is an association between the Http Session under which a client is operating and any web sockets that are established within that HttpSession. The API allows access from each web socket Session to the unique HttpSession corresponding to that same client. Here the specification defines the relationship between a websocket connection with a client and the HttpSession of the opening handshake of the client that initiated it.

if the web socket is a protected resource in the web application, that is to say, required an authorized user to access it, and the user explicitly invalidates the HttpSession, the websocket implementation must close the web socket connection immediately [WSC-44]

if the user of the web application is actively using the web sockets within the web application, but does not access any of the web resources, the web socket implementation must keep the HttpSession from timing out (TBD This a request to the servlet specification). [WSC-45]

8. Server Security

Web Socket endpoints are secured using the web container security model. The goal of this is to make it easy for a web socket developer to declare whether access to a web socket server endpoint needs to be authenticated, and who can access it, and if it needs an encrypted connection or not. A web socket which is mapped to a given ws:// URI (as described in Chapters 3 and 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, web socket developers may assign an authentication scheme, user roles granted access and transport guarantee to their web socket endpoints.

Authentication of web sockets

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

If a client sends an unauthenticated opening handshake request for a web socket that is protected by the security mechanism, the web socket runtime must return a 401 (Unauthorized) response to the opening handshake request and may not initiate a web socket connection [WSC-81].

TBD Add/reuse security annotations to define authorization for web sockets.

Authorization of web sockets

A web socket'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 web socket [WSC-82]. The container must interpret any http-method other than GET (or the default, missing) as not applying to the web socket. [WSC-83]

Transport Guarantee

A transport guarantee of NONE must be interpreted by the container as allowing unencrypted ws:// connections to the web socket [WSC-84]. A transport guarantee of CONFIDENTIAL must be interpreted by the container as only allowing access to the web socket over an encrypted (wss://) connection [WSC-85]. This may require a pre-authenticated request.

ToDo Add example

9. Bibliography

[1] The WebSocket Protocol RFC at the Internet Engineering Task Force (IETF) See <http://tools.ietf.org/html/rfc6455>

[2] The WebSocket API specification at the World Wide Web Consortium (W3C) See <http://dev.w3.org/html5/websockets/>

[3] Expert group mailing archive <http://java.net/projects/websocket-spec/lists/jsr356-experts/archive>

[4] Servlet specification 3.1 <http://jcp.org/en/jsr/detail?id=340>.

[5] Key words for use in RFCs to Indicate Requirement Levels <http://www.ietf.org/rfc/rfc2119.txt>

[6] URI-Templates <http://tools.ietf.org/html/rfc6570>