Tyrus 1.3.3 User Guide

Tyrus 1.3.3 User Guide

Table of Contents

List of Tables

List of Examples

Preface

This is user guide for Tyrus 1.3.3. We are trying to keep it up to date as we add new features. Please use also our API documentation linked from the Tyrus [http://tyrus.java.net] and Java API for WebSocket [https://java.net/projects/websocket-spec] home pages as an additional source of information about Tyrus features and API. If you would like to contribute to the guide or have questions on things not covered in our docs, please contact us at users@tyrus.java.net [mailto:users@tyrus.java.net].

Chapter 1. Getting Started

This chapter provides a quick introduction on how to get started building WebSocket services using Java API for WebSocket and Tyrus. The example described here presents how to implement simple websocket service as JavaEE web application that can be deployed on any servlet container supporting Servlet 3.1 and higher. It also discusses starting Tyrus in standalone mode.

1.1. WebSocket Services Using Java API for WebSocket

First, to use the Java API for WebSocket in your project you need to depend on the following artifact:

```
<dependency>
    <groupId>javax.websocket</groupId>
    <artifactId>javax.websocket-api</artifactId>
    <version>1.0</version>
</dependency>
```

1.1.1. Creating Annotated Server Endpoint

In this section we will create a simple server side websocket endpoint which will echo the received message back to the sender. We will deploy this endpoint on the container.

In Java API for WebSocket and Tyrus, there are two basic approaches how to create an endpoint - either annotated endpoint, or programmatic endpoint. By annotated endpoint we mean endpoint constructed by using annotations (`javax.websocket.server.ServerEndpoint` for server endpoint and `javax.websocket.ClientEndpoint` for client endpoint), like in "Annotated Echo Endpoint".

Example 1.1. Annotated Echo Endpoint

```
1 @ServerEndpoint(value = "/echo")
2 public class EchoEndpointAnnotated {
3     @OnMessage
4     public String onMessage(String message, Session session) {
5         return message;
6     }
7 }
8
```

The functionality of the `EchoEndpointAnnotated` is fairly simple - to send the received message back to the sender. To turn a POJO (Plain Old Java Object) to WebSocket server endpoint, the annotation `@ServerEndpoint(value = "/echo")` needs to be put on the POJO - see line 1. The URI path of the endpoint is `"/echo"`. The annotation `@OnMessage` - line 3 on the method `public String onMessage(String message, Session session)` indicates that this method will be called whenever text message is received. On line 5 in this method the message is sent back to the user by returning it from the message.

The application containing only the `EchoEndpointAnnotated` class can be deployed to the container.

1.1.2. Client Endpoint

Let's create the client part of the application. The client part may be written in JavaScript or any other technology supporting WebSockets. We will use Java API for WebSocket and Tyrus to demonstrate how to develop programmatic client endpoint. The following code is used as a client part to communicate with the `EchoEndpoint` deployed on server using Tyrus and Java API for WebSocket.

The example "Client Endpoint" utilizes the concept of the programmatic endpoint. By programmatic endpoint we mean endpoint which is created by extending class `javax.websocket.Endpoint`. The example is standalone java application which needs to depend on some Tyrus artifacts to work correctly, see "Tyrus Standalone Mode". In the example first the `CountDownLatch` is initialized. It is needed as a bocking data structure - on line 31 it either waits for 100 seconds, or until it gets counted down (line 22). On line 9 the `javax.websocket.ClientEndpointConfig` is created - we will need it later to connect the endpoint to the server. On line 11 the `org.glassfish.tyrus.client.ClientManager` is created. it implements the `javax.websocket.WebSocketContainer` and is used to connect to server. This happens on next line. The client endpoint functionality is contained in the `javax.websocket.Endpoint` lazy instantiation. In the `onOpen` method new MessageHandler is registered (the received message is just printed on the console and the latch is counted down). After the registration the message is sent to tje servere (line 25).

Example 1.2. Client Endpoint

```
 1 public class DocClient {
 2     private CountDownLatch messageLatch;
 3     private static final String SENT_MESSAGE = "Hello World";
 4
 5     public static void main(String [] args){
 6         try {
 7             messageLatch = new CountDownLatch(1);
 8
 9             final ClientEndpointConfig cec = ClientEndpointConfig.Builder.crea
10
11             ClientManager client = ClientManager.createClient();
12             client.connectToServer(new Endpoint() {
13
14                 @Override
15                 public void onOpen(Session session, EndpointConfig config) {
16                     try {
17                         session.addMessageHandler(new MessageHandler.Whole<Str
18
19                             @Override
20                             public void onMessage(String message) {
21                                 System.out.println("Received message: "+messag
22                                 messageLatch.countDown();
23                             }
24                         });
25                         session.getBasicRemote().sendText(SENT_MESSAGE);
26                     } catch (IOException e) {
27                         e.printStackTrace();
28                     }
29                 }
30             }, cec, new URI("ws://localhost:8025/websockets/echo"));
31             messageLatch.await(100, TimeUnit.SECONDS);
32         } catch (Exception e) {
33             e.printStackTrace();
34         }
35     }
36 }
```

1.1.3. Creating Server Endpoint Programmatically

Similarly to "Client Endpoint" the server registered endpoint may also be the programmatic one:

Example 1.3. Programmatic Echo Endpoint

```
 1 public class EchoEndpointProgrammatic extends Endpoint {
 2     @Override
 3     public void onOpen(final Session session, EndpointConfig config) {
 4         session.addMessageHandler(new MessageHandler.Whole<String>() {
 5             @Override
 6             public void onMessage(String message) {
 7                 try {
 8                     session.getBasicRemote().sendText(message);
 9                 } catch (IOException e) {
10                     e.printStackTrace();
11                 }
12             }
13         });
14     }
15 }
```

The functionality of the `EchoEndpointProgrammatic` is fairly simple - to send the received message back to the sender. The programmatic server endpoint needs to extend `javax.websocket.Endpoint` - line 1. Mehod `public void onOpen(final Session session, EndpointConfig config)` gets called once new connection to this endpoin0t is opened. In this method the `MessageHandler` is registered to the `javax.websocket.Session` instance, which opened the connection. Method `public void onMessage(String message)` gets called once the message is received. On line 8 the message is sent back to the sender.

To see how both annotated and programmatic endpoints may be deployed please check the section Deployment. In short: you need to put the server endpoint classes into WAR, deploy on server and the endpoints will be scanned by server and deployed.

1.1.4. Tyrus in Standalone Mode

To use Tyrus in standalone mode it is necessary to depend on correct Tyrus artifacts. The following artifacts need to be added to your pom to use Tyrus:

```
<dependency>
    <groupId>org.glassfish.tyrus</groupId>
    <artifactId>tyrus-server</artifactId>
    <version>1.3.3</version>
</dependency>

<dependency>
    <groupId>org.glassfish.tyrus</groupId>
    <artifactId>tyrus-client</artifactId>
    <version>1.3.3</version>
</dependency>

<dependency>
    <groupId>org.glassfish.tyrus</groupId>
    <artifactId>tyrus-container-grizzly</artifactId>
    <version>1.3.3</version>
</dependency>
```

Let's use the very same example like for Java API for WebSocket and deploy the `EchoEndpointAnnotated` on the standalone Tyrus server on the hostname "localhost", port 8025 and path "/websocket", so the endpoint will be available at address "ws://localhost:8025/websockets/echo".

```
public void runServer() {
    Server server = new Server("localhost", 8025, "/websocket", EchoEndpoint.class

    try {
        server.start();
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in
        System.out.print("Please press a key to stop the server.");
        reader.readLine();
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        server.stop();
    }
}
```

Chapter 2. Tyrus Modules and Dependencies

Tyrus is built, assembled and installed using Maven. Tyrus is deployed to maven.org maven repository at the following location:http://search.maven.org/. Jars, jar sources, jar javadoc and samples are all available on the java.net maven repository.

All Tyrus components are built using Java SE 7 compiler. It means, you will also need at least Java SE 7 to be able to compile and run your application. Developers using maven are likely to find it easier to include and manage dependencies of their applications than developers using ant or other build technologies. The following table provides an overview of all Tyrus modules and their dependencies with links to the respective binaries.

Table 2.1. Tyrus core modules

Module	Dependencies	Description
tyrus-server [http:// search.maven.org/ #artifactdetails| org.glassfish.tyrus| tyrus-server| 1.3.3|jar]	tyrus-core, tyrus-spi, tyrus-websocket-core	Basic server functionality
tyrus-core [http:// search.maven.org/ #artifactdetails| org.glassfish.tyrus| tyrus-core| 1.3.3|jar]	tyrus-spi, tyrus-websocket-core	Core Tyrus functionality
tyrus-client [http:// search.maven.org/ #artifactdetails| org.glassfish.tyrus| tyrus-client| 1.3.3|jar]	tyrus-core, tyrus-spi, tyrus-websocket-core	Basic client functionality
tyrus-documentation [http:// search.maven.org/ #artifactdetails| org.glassfish.tyrus| tyrus-documentation| 1.3.3|jar]	[nothing]	Project documentation
tyrus-websocket-core [http://	[nothing]	The WebSocket protocol

Module	Dependencies	Description
search.maven.org/ #artifactdetails| org.glassfish.tyrus| tyrus- websocket- core|1.3.3| jar]		
tyrus- samples [http:// search.maven.org/ remotecontent? filepath=org/ glassfish/ tyrus/ bundles/ tyrus- samples/1.3.3/ tyrus- samples-1.3.3- all.zip]	tyrus-server, tyrus-client, tyrus- container-grizzly, tyrus-core, tyrus- spi, tyrus-websocket-core	Samples of using Java API for WebSocket and Tyrus
tyrus-spi [http:// search.maven.org/ #artifactdetails| org.glassfish.tyrus| tyrus-spi| 1.3.3|jar]	[nothing]	SPI

Table 2.2. Tyrus containers

Module	Dependencies	Description
tyrus- container- glassfish- cdi [http:// search.maven.org/ #artifactdetails| org.glassfish.tyrus| tyrus- container- glassfish-cdi| 1.3.3|jar]	tyrus-spi	CDI support
tyrus- container- glassfish- ejb [http:// search.maven.org/ #artifactdetails| org.glassfish.tyrus| tyrus-	tyrus-spi	EJB support

Module	Dependencies	Description
container-glassfish-ejb|1.3.3|jar]		
tyrus-container-grizzly [http://search.maven.org/#artifactdetails|org.glassfish.tyrus|tyrus-container-grizzly|1.3.3|jar]	tyrus-core, tyrus-spi, tyrus-websocket-core	Grizzly integration for Tyrus client and standalone server usage
tyrus-container-servlet [http://search.maven.org/#artifactdetails|org.glassfish.tyrus|tyrus-container-servlet|1.3.3|bundle]	tyrus-server, tyrus-core, tyrus-spi, tyrus-websocket-core	Servlet support for integration into web containers

Chapter 3. Deploying WebSocket Endpoints

Deploying WebSocket endpoints can be done in two ways. Either deploying via putting the endpoint in the WAR file, or using the ServerContainer methods to deploy the programmatic endpoint in the deployment phase.

3.1. Deploying Endpoints as a WAR file

The classes that are scanned for in WAR are the following ones:

- Classes that implement the `javax.websocket.ServerApplicationConfig`.

- Classes annotated with `javax.websocket.server.ServerEndpoint`.

- Classes that extend `javax.websocket.Endpoint`.

3.1.1. Deployment Algorithm

1. If one or more classes implementing ServerApplicationConfiguration are present in the WAR file, Tyrus deploys endpoints provided by all of these classes. Tyrus doesn't deploy any other classes present in the WAR (annotated by `javax.websocket.server.ServerEndpoint` or extending `javax.websocket.Endpoint`).

2. If no class implementing ServerApplicationConfiguration is present, Tyrus deploys all classes annotated with @ServerEndpoint or extending Endpoint present in the WAR.

Let's have the following classes in the WAR:

Example 3.1. Deployment of WAR containing several classes extending javax.websocket.server.ServerApplicationConfig

```
1  public class MyApplicationConfigOne implements ServerApplicationConfig {
2      public Set<ServerEndpointConfig> getEndpointConfigs(Set<Class<? extends En
3          Set<Class<? extends Endpoint>> s = new HashSet<Class<? extends Endpoin
4          s.add(ProgrammaticEndpointOne.class);
5          return s;
6      }
7
8      public Set<Class> getAnnotatedEndpointClasses(Set<Class<?>> scanned);
9         Set<Class<?>> s = new HashSet<Class<?>>;
10         s.add(AnnotatedEndpointOne.class);
11         return s;
12     }
13 }
14
15 public class MyApplicationConfigTwo implements ServerApplicationConfig {
16     public Set<ServerEndpointConfig> getEndpointConfigs(Set<Class<? extends En
17         Set<Class<? extends Endpoint>> s = new HashSet<Class<? extends Endpoin
18         s.add(ProgrammaticEndpointTwo.class);
19     return s;
20 }
21
22 public Set<Class> getAnnotatedEndpointClasses(Set<Class<?>> scanned);
23     Set<Class<?>> s = new HashSet<Class<?>>;
24         s.add(AnnotatedEndpointTwo.class);
25         return s;
26     }
27 }
28
29 @ServerEndpoint(value = "/annotatedone")
30 public class AnnotatedEndpointOne {
31     ...
32 }
33
34 @ServerEndpoint(value = "/annotatedtwo")
35     public class AnnotatedEndpointTwo {
36     ...
37 }
38
39 @ServerEndpoint(value = "/annotatedthree")
40 public class AnnotatedEndpointThree {
41     ...
42 }
43
44 public class ProgrammaticEndpointOne extends Endpoint {
45     ...
46 }
47
48 public class ProgrammaticEndpointTwo extends Endpoint {
49     ...
50 }
51
52 public class ProgrammaticEndpointThree extends Endpoint {
53     ...
54 }
```

According to the deployment algorithm classes `AnnotatedEndpointOne`, `AnnotatedEndpointTwo`, `ProgrammaticEndpointOne` and `ProgrammaticEndpointTwo` will be deployed. `AnnotatedEndpointThree` and `ProgrammaticEndpointThree` will not be deployed, as these are not returned by the respective methods of `MyApplicationConfigOne` nor `MyApplicationConfigTwo`.

3.2. Deploying endpoints via `javax.websocket.server.ServerContainer`

Endpoints may be deployed using `javax.websocket.server.ServerContainer` during the application initialization phase. For websocket enabled web containers, developers may obtain a reference to the ServerContainer instance by retrieving it as an attribute named `javax.websocket.server.ServerContainer` on the ServletContext, see the following example for annotated endpoint:

Example 3.2. Deployment of Annotated Endpoint Using ServerContainer

```
 1 @WebListener
 2 @ServerEndpoint("/annotated")
 3 public class MyServletContextListenerAnnotated implements ServletContextListen
 4
 5     @Override
 6     public void contextInitialized(ServletContextEvent servletContextEvent) {
 7         final ServerContainer serverContainer = (ServerContainer) servletConte
 8                                                 .getAttribute("javax.webso
 9
10         try {
11             serverContainer.addEndpoint(MyServletContextListenerAnnotated.clas
12         } catch (DeploymentException e) {
13             e.printStackTrace();
14         }
15     }
16
17     @OnMessage
18     public String onMessage(String message) {
19         return message;
20     }
21
22     @Override
23     public void contextDestroyed(ServletContextEvent servletContextEvent) {
24     }
25 }
```

Chapter 4. WebSocket API Endpoints, Sessions and MessageHandlers

This chapter presents an overview of the core WebSocket API concepts - endpoints, configurations and message handlers.

The JAVA API for WebSocket specification draft can be found online here [http://jcp.org/aboutJava/communityprocess/pfd/jsr356/index.html].

4.1. Endpoint Classes

Server endpoint classes are POJOs (Plain Old Java Objects) that are annotated with `javax.websocket.server.ServerEndpoint`. Similarly, *client endpoint classes* are POJOs annotated with javax.websocket.ClientEndpoint. This section shows how to use Tyrus to annotate Java objects to create WebSocket web services.

The following code example is a simple example of a WebSocket endpoint using annotations. The example code shown here is from echo sample which ships with Tyrus.

Example 4.1. Echo sample server endpoint.

```
 1 @ServerEndpoint("/echo")
 2 public class EchoEndpoint {
 3
 4     @OnOpen
 5     public void onOpen(Session session) throws IOException {
 6         session.getBasicRemote().sendText("onOpen");
 7     }
 8
 9     @OnMessage
10     public String echo(String message) {
11         return message + " (from your server)";
12     }
13
14     @OnError
15     public void onError(Throwable t) {
16         t.printStackTrace();
17     }
18
19     @OnClose
20     public void onClose(Session session) {
21
22     }
23 }
```

Let's explain the JAVA API for WebSocket annotations.

4.1.1. javax.websocket.server.ServerEndpoint

javax.websocket.server.ServerEndpoint has got one mandatory field - *value* and four optional fields. See the example below.

Example 4.2. javax.websocket.server.ServerEndpoint with all fields specified

```
 1 @ServerEndpoint(
 2     value = "/sample",
 3     decoders = ChatDecoder.class,
 4     encoders = DisconnectResponseEncoder.class,
 5     subprotocols = {"subprtotocol1", "subprotocol2"},
 6     configurator = Configurator.class
 7 )
 8 public class SampleEndpoint {
 9
10     @OnMessage
11     public SampleResponse receiveMessage(SampleType message, Session session)
12         return new SampleResponse(message);
13     }
14 }
```

4.1.1.1. value

Denotes a relative URI path at which the server endpoint will be deployed. In the example "javax.websocket.server.ServerEndpoint with all fields specified", the Java class will be hosted at the URI path /sample. The field *value* must begin with a '/' and may or may not end in a '/', it makes no difference. Thus request URLs that end or do not end in a '/' will both be matched. WebSocket API for JAVA supports level 1 URI templates.

URI path templates are URIs with variables embedded within the URI syntax. These variables are substituted at runtime in order for a resource to respond to a request based on the substituted URI. Variables are denoted by curly braces. For example, look at the following @ServerEndpoint annotation:

```
@ServerEndpoint("/users/{username}")
```

In this type of example, a user will be prompted to enter their name, and then a Tyrus web service configured to respond to requests to this URI path template will respond. For example, if the user entered their username as "Galileo", the web service will respond to the following URL: http://example.com/users/Galileo

To obtain the value of the username variable the javax.websocket.server.PathParam may be used on method parameter of methods annotated with one of @OnOpen, @OnMessage, @OnError, @OnClose.

Example 4.3. Specifying URI path parameter

```
1 @ServerEndpoint("/users/{username}")
2 public class UserEndpoint {
3
4     @OnMessage
5     public String getUser(String message, @PathParam("username") String userNa
6         ...
7     }
8 }
```

4.1.1.2. decoders

Contains list of classes that will be used decode incoming messages for the endpoint. By decoding we mean transforming from text / binary websocket message to some user defined type. Each decoder needs to implement the Decoder interface.

`SampleDecoder` in the following example decodes String message and produces SampleType message
- see decode method on line 4.

Example 4.4. SampleDecoder

```
 1 public class SampleDecoder implements Decoder.Text<SampleType> {
 2
 3     @Override
 4     public SampleType decode(String s) {
 5         return new SampleType(s);
 6     }
 7
 8     @Override
 9     public boolean willDecode(String s) {
10         return s.startsWith(SampleType.PREFIX);
11     }
12
13     @Override
14     public void init(EndpointConfig config) {
15         // do nothing.
16     }
17
18     @Override
19     public void destroy() {
20         // do nothing.
21     }
22 }
```

4.1.1.3. encoders

Contains list of classes that will be used to encode outgoing messages. By encoding we mean transforming
message from user defined type to text or binary type. Each encoder needs to implement the Encoder
interface.

`SampleEncoder` in the following example decodes String message and produces SampleType message
- see decode method on line 4.

Example 4.5. SampleEncoder

```
 1 public class SampleEncoder implements Encoder.Text<SampleType> {
 2
 3     @Override
 4     public String encode(SampleType message) {
 5         return data.toString();
 6     }
 7
 8     @Override
 9     public void init(EndpointConfig config) {
10         // do nothing.
11     }
12
13     @Override
14     public void destroy() {
15         // do nothing.
16     }
17 }
```

4.1.1.4. subprotocols

List of names (Strings) of supported sub-protocols. The first protocol in this list that matches with sub-protocols provided by the client side is used.

4.1.1.5. configurator

Users may provide their own implementation of ServerEndpointConfiguration.Configurator. It allows them to control some algorithms used by Tyrus in the connection initialization phase:

- `public String getNegotiatedSubprotocol(List<String> supported, List<String> requested)` allows the user to provide their own algorithm for selection of used subprotocol.

- `public List<Extension> getNegotiatedExtensions(List<Extension> installed, List<Extension> requested)` allows the user to provide their own algorithm for selection of used Extensions.

- `public boolean checkOrigin(String originHeaderValue)`. allows the user to specify the origin checking algorithm.

- `public void modifyHandshake(ServerEndpointConfig sec, HandshakeRequest request, HandshakeResponse response)` . allows the user to modify the handshake response that will be sent back to the client.

- `public <T> T getEndpointInstance(Class<T> endpointClass) throws InstantiationException` . allows the user to provide the way how the instance of an Endpoint is created

```
 1 public class ConfiguratorTest extends ServerEndpointConfig.Configurator{
 2
 3     public String getNegotiatedSubprotocol(List<String> supported, List<String
 4         // Plug your own algorithm here
 5     }
```

```
 6
 7    public List<Extension> getNegotiatedExtensions(List<Extension> installed,
 8        // Plug your own algorithm here
 9    }
10
11    public boolean checkOrigin(String originHeaderValue) {
12        // Plug your own algorithm here
13    }
14
15    public void modifyHandshake(ServerEndpointConfig sec, HandshakeRequest req
16        // Plug your own algorithm here
17    }
18
19    public <T> T getEndpointInstance(Class<T> endpointClass) throws Instantiat
20        // Plug your own algorithm here
21    }
22 }
```

4.1.2. javax.websocket.ClientEndpoint

The @ClientEndpoint class-level annotation is used to turn a POJO into WebSocket client endpoint. In
the following sample the client sends text message "Hello!" and prints out each received message.

Example 4.6. SampleClientEndpoint

```
 1 @ClientEndpoint(
 2     decoders = SampleDecoder.class,
 3     encoders = SampleEncoder.class,
 4     subprotocols = {"subprtotocol1", "subprotocol2"},
 5     configurator = ClientConfigurator.class)
 6 public class SampleClientEndpoint {
 7
 8     @OnOpen
 9     public void onOpen(Session p) {
10         try {
11             p.getBasicRemote().sendText("Hello!");
12         } catch (IOException e) {
13             e.printStackTrace();
14         }
15     }
16
17     @OnMessage
18     public void onMessage(String message) {
19         System.out.println(String.format("%s %s", "Received message: ", messag
20     }
21 }
22
```

4.1.2.1. decoders

Contains list of classes that will be used decode incoming messages for the endpoint. By decoding we
mean transforming from text / binary websocket message to some user defined type. Each decoder needs
to implement the Decoder interface.

4.1.2.2. encoders

Contains list of classes that will be used to encode outgoing messages. By encoding we mean transforming message from user defined type to text or binary type. Each encoder needs to implement the Encoder interface.

4.1.2.3. subprotocols

List of names (Strings) of supported sub-protocols.

4.1.2.4. configurator

Users may provide their own implementation of ClientEndpointConfiguration.Configurator. It allows them to control some algorithms used by Tyrus in the connection initialization phase. Method *beforeRequest* allows the user to change the request headers constructed by Tyrus. Method *afterResponse* allows the user to process the handshake response.

```
 1 public class Configurator {
 2
 3     public void beforeRequest(Map<String, List<String>> headers) {
 4         //affect the headers before request is sent
 5     }
 6
 7     public void afterResponse(HandshakeResponse hr) {
 8         //process the handshake response
 9     }
10 }
```

4.2. Endpoint method-level annotations

4.2.1. @OnOpen

This annotation may be used on certain methods of @ServerEndpoint or @ClientEndpoint, but only once per endpoint. It is used to decorate a method which is called once new connection is established. The connection is represented by the optional Session parameter. The other optional parameter is EndpointConfig, which represents the passed configuration object. Note that the EndpointConfig allows the user to access the user properties.

Example 4.7. @OnOpen with Session and EndpointConfig parameters.

```
 1 @ServerEndpoint("/sample")
 2 public class EchoEndpoint {
 3
 4     private Map<String, Object> properties;
 5
 6     @OnOpen
 7     public void onOpen(Session session, EndpointConfig config) throws IOExcept
 8         session.getBasicRemote().sendText("onOpen");
 9         properties = config.getUserProperties();
10     }
11 }
```

4.2.2. @OnClose

This annotation may be used on any method of @ServerEndpoint or @ClientEndpoint, but only once per endpoint. It is used to decorate a method which is called once the connection is being closed. The method may have one Session parameter, one CloseReason parameter and parameters annotated with @PathParam.

Example 4.8. @OnClose with Session and CloseReason parameters.

```
1 @ServerEndpoint("/sample")
2 public class EchoEndpoint {
3
4     @OnClose
5     public void onClose(Session session, CloseReason reason) throws IOExceptio
6         //prepare the endpoint for closing.
7     }
8 }
```

4.2.3. @OnError

This annotation may be used on any method of @ServerEndpoint or @ClientEndpoint, but only once per endpoint. It is used to decorate a method which is called once Exception is being thrown by any method annotated with @OnOpen, @OnMessage and @OnClose. The method may have optional Session parameter and Throwable parameters.

Example 4.9. @OnError with Session and Throwable parameters.

```
1 @ServerEndpoint("/sample")
2 public class EchoEndpoint {
3
4     @OnError
5     public void onError(Session session, Throwable t) {
6         t.printStackTrace();
7     }
8 }
```

4.2.4. @OnMessage

This annotation may be used on certain methods of @ServerEndpoint or @ClientEndpoint, but only once per endpoint. It is used to decorate a method which is called once new message is received.

Example 4.10. @OnError with Session and Throwable parameters.

```
1 @ServerEndpoint("/sample")
2 public class EchoEndpoint {
3
4     @OnMessage
5     public void onMessage(Session session, Throwable t) {
6         t.printStackTrace();
7     }
8 }
```

4.3. MessageHandlers

Implementing the `javax.websocket.MessageHandler` interface is one of the ways how to receive messages on endpoints (both server and client). It is aimed primarily on programmatic endpoints, as the annotated ones use the method level annotation `javax.websocket.OnMessage` to denote the method which receives messages.

The MessageHandlers get registered on the Session instance:

Example 4.11. MessageHandler basic example

```
 1 public class MyEndpoint extends Endpoint {
 2
 3     @Override
 4     public void onOpen(Session session, EndpointConfig EndpointConfig) {
 5         session.addMessageHandler(new MessageHandler.Whole<String>() {
 6             @Override
 7             public void onMessage(String message) {
 8                 System.out.println("Received message: "+message);
 9             }
10         });
11     }
12 }
```

There are two orthogonal criterions which classify MessageHandlers. According the WebSocket Protocol (RFC 6455) the message may be sent either complete, or in chunks. In Java API for WebSocket this fact is reflected by the interface which the handler implements. Whole messages are processed by handler which implements `javax.websocket.MessageHandler.Whole` interface. Partial messages are processed by handlers that implement `javax.websocket.MessageHandler.Partial` interface. However, if user registers just the whole message handler, it doesn't mean that the handler will process solely whole messages. If partial message is received, the parts are cached by Tyrus until the final part is received. Then the whole message is passed to the handler. Similarly, if the user registers just the partial message handler and whole message is received, it is passed directly to the handler.

The second criterion is the data type of the message. WebSocket Protocol (RFC 6455) defines four message data type - text message, According to Java API for WebSocket the text messages will be processed by MessageHandlers with the following types:

- *java.lang.String*

- *java.io.Reader*

- any developer object for which there is a corresponding *javax.websocket.Decoder.Text* or *javax.websocket.Decoder.TextStream.*

The binary messages will be processed by MessageHandlers with the following types:

- *java.nio.ByteBuffer*

- *java.io.InputStream*

- any developer object for which there is a corresponding *javax.websocket.Decoder.Binary* or *javax.websocket.Decoder.BinaryStream.*

The Java API for WebSocket limits the registration of MessageHandlers per Session to be one MessageHandler per native websocket message type. 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. This rule holds for both whole and partial message handlers, i.e there may be one text MessageHandler - either whole, or partial, not both.

Chapter 5. Configurations

javax.websocket.server.ServerEndpointConfig and javax.websocket.ClientEndpointConfig objects are used to provide the user the ability to configure websocket endpoints. Both server and client endpoints have some part of configuration in common, namely encoders, decoders, and user properties. The user properties may developers use to store the application specific data. For the developer's convenience the builders are provided for both ServerEndpointConfig and ClientEndpointConfig.

5.1. `javax.websocket.server.ServerEndpointCon`

The `javax.websocket.server.ServerEndpointConfig` is used when deploying the endpoint either via implementing the `javax.websocket.server.ServerApplicationConfig`, or via registering the programmatic endpoint at the `javax.websocket.server.ServerContainer` instance. It allows the user to create the configuration programmatically.

The following example is used to deploy the EchoEndpoint programmatically. In the method `getEndpointClass()` the user has to specify the class of the deployed endpoint. In the example Tyrus will create an instance of `EchoEndpoint` and deploy it. This is the way how to tie together endpoint and it's configuration. In the method `getPath()` the user specifies that that the endpoint instance will be deployed at the path "/echo". In the method `public List<String> getSubprotocols()` the user specifies that the supported subprotocols are "echo1" and "echo2". The method `getExtensions()` defines the extensions the endpoint supports. Similarly the example configuration does not use any configurator. Method `public List<Class<? extends Encoder>> getEncoders()` defines the encoders used by teh endpoint. The decoders and user properties map are defined in similar fashion.

If the endpoint class which is about to be deployed is an annotated endpoint, note that the endpoint configuration will be taken from configuration object, not from the annotation on the endpoint class.

Example 5.1. Configuration for EchoEndpoint Deployment

```
 1 public class EchoEndpointConfig implements ServerEndpointConfig{
 2
 3     private final Map<String, Object> userProperties = new HashMap<String, Obj
 4
 5     @Override
 6     public Class<?> getEndpointClass() {
 7         return EchoEndpoint.class;
 8     }
 9
10     @Override
11     public String getPath() {
12         return "/echo";
13     }
14
15     @Override
16     public List<String> getSubprotocols() {
17         return Arrays.asList("echo1","echo2");
18     }
19
20     @Override
21     public List<Extension> getExtensions() {
22         return null;
23     }
24
25     @Override
26     public Configurator getConfigurator() {
27         return null;
28     }
29
30     @Override
31     public List<Class<? extends Encoder>> getEncoders() {
32         return Arrays.asList(SampleEncoder.class);
33     }
34
35     @Override
36     public List<Class<? extends Decoder>> getDecoders() {
37         return Arrays.asList(SampleDecoder.class);
38     }
39
40     @Override
41     public Map<String, Object> getUserProperties() {
42         return userProperties;
43     }
44 }
```

To make the development easy the *javax.websocket.server.ServerEndpointConfig* provides a builder to construct the configuration object:

Example 5.2. ServerEndpointConfigu built using Builder

```
1 ServerEndpointConfig config = ServerEndpointConfig.Builder.create(EchoEndpoint
2     decoders(Arrays.<Class<? extends Decoder>>asList(JsonDecoder.class)).
3     encoders(Arrays.<Class< extends Encoder>>asList(JsonEncoder.class)).build(
```

5.2. `javax.websocket.ClientEndpointConfig`

The `javax.websocket.ClientEndpointConfig` is used when deploying the programmatic client endpoint via registering the programmatic endpoint at the `WebSocketContainer` instance. Some of the configuration methods come from the `EndpointConfig`class, which is extended by both `javax.websocket.server.ServerEndpointConfig` and `javax.websocket.ClientEndpointConfig`. Then there are methods for configuring the preferred subprotocols the client endpoint wants to use and supported extensions. It is also possible to use the ClientEndpointConfig.Configurator in order to be able to affect the endpoint behaviour before and after request.

Similarly to the ServerEndpointConfig, there is a Builder provided to construct the configuration easily:

Example 5.3. ClientEndpointConfig built using Builder

```
1 ClientEndpointConfig.Builder.create().
2     decoders(Arrays.<Class<? extends Decoder>>asList(JsonDecoder.class)).
3     encoders(Arrays.<Class<? extends Encoder>>asList(JsonEncoder.class)).
4     preferredSubprotocols(Arrays.asList("echo1", "echo2")).build();
```

Chapter 6. Endpoint Lifecycle, Sessions, Sending Messages

6.1. Endpoint Lifecycle

As mentioned before, the endpoint in Java API for WebSocket is represented either by instance of `javax.websocket.Endpoint`, or by class annotated with either `javax.websocket.server.ServerEndpoint` or `javax.websocket.ClientEndpoint`. Unless otherwise defined by developer provided configurator (defined in instance of `javax.websocket.server.ServerEndpointConfig` or `javax.websocket.ClientEndpointConfig`, Tyrus uses one endpoint instance per VM per connected peer. Therefore one endpoint instance typically handles connections from one peer.

6.2. `javax.websocket.Session`

The sequence of interactions between an endpoint instance and remote peer is in Java API for WebSocket modelled by `javax.websocket.Session` instance. This interaction starts by mandatory open notification, continues by 0 - n websocket messages and is finished by mandatory closing notification.

The `javax.websocket.Session` instance is passed by Tyrus to the user in the following methods for programmatic endpoints:

- `public void onOpen(Session session, EndpointConfig config)`

- `public void onClose(Session session, CloseReason closeReason)`

- `public void onError(Session session, Throwable thr)`

The `javax.websocket.Session` instance is passed by Tyrus to the user in the methods annotated by following annotations for annotated endpoints:

- method annotated with `javax.websocket.OnOpen`

- method annotated with `javax.websocket.OnMessage`

- method annotated with `javax.websocket.OnClose`

- method annotated with `javax.websocket.OnError`

In each of the methods annotated with the preceeding annotations the user may use parameter of type `javax.websocket.Session`. In the following example the developer wants to send a message in the method annotated with `javax.websocket.OnOpen`. As we will demonstrate later, the developer needs the session instance to do so. According to Java API for WebSocket Session is one of the allowed parameters in methods annotated with `javax.websocket.OnOpen`. Once the annotated method gets called, Tyrus passes in the correct instance of `javax.websocket.Session`.

Example 6.1. Lifecycle echo sample

```
 1 @ServerEndpoint("/echo")
 2 public class EchoEndpoint {
 3
 4     @OnOpen
 5     public void onOpen(Session session) throws IOException {
 6         session.getBasicRemote().sendText("onOpen");
 7     }
 8
 9     @OnMessage
10     public String echo(String message) {
11         return message;
12     }
13
14     @OnError
15     public void onError(Throwable t) {
16         t.printStackTrace();
17     }
18 }
```

6.3. Sending Messages

Generally there are two ways how to send message to the peer endpoint. First one is usable for annotated endpoints only. The user may send the message by returning the message content from the method annotated with `javax.websocket.OnMessage`. In the following example the message m is sent back to the remote endpoint.

Example 6.2. Sending message in @OnMessage

```
1 @OnMessage
2 public String echo(String m) {
3     return m;
4 }
```

The other option how to send a message is to obtain the `javax.websocket.RemoteEndpoint` instance via the `javax.websocket.Session` instance. See the following example:

Example 6.3. Sending message via RemoteEndpoint.Basic instance

```
1 @OnMessage
2 public void echo(String message, Session session) {
3     session.getBasicRemote().sendText(message);
4 }
```

6.4. RemoteEndpoint

The interface `javax.websocket.RemoteEndpoint`, part of Java API for WebSocket, is designed to represent the other end of the communication (related to the endpoint), so the developer uses it to send the message. There are two basic interfaces the user may use - `javax.websocket.RemoteEndpoint` `$Basic` and `javax.websocket.RemoteEndpoint$Async`.

6.4.1. `javax.websocket.RemoteEndpoint.Basic`

is used to send synchronous messages The point of completion of the send is defined when all the supplied data has been written to the underlying connection. The methods for sending messages on the `javax.websocket.RemoteEndpoint$Basic` block until this point of completion is reached, except for `javax.websocket.RemoteEndpoint$Basic#getSendStream()` and `javax.websocket.RemoteEndpoint$Basic#getSendWriter()` which present traditional blocking I/O streams to write messages. See the example "Sending message via RemoteEndpoint.Basic instance" to see how the whole text message is send. The following example demonstrates a method which sends the partial text method to the peer:

Example 6.4. Method for sending partial text message

```
1 public void sendPartialTextMessage(String message, Boolean isLast, Session ses
2     try {
3         session.getBasicRemote().sendText(message, isLast);
4     } catch (IOException e) {
5         e.printStackTrace();
6     }
7 }
```

6.4.2. `javax.websocket.RemoteEndpoint.Async`

This representation of the peer of a web socket conversation has the ability to send messages asynchronously. The point of completion of the send is defined when all the supplied data has been written to the underlying connection. The completion handlers for the asynchronous methods are always called with a different thread from that which initiated the send.

Example 6.5. Sending mesage the async way using Future

```
1 public void sendWholeAsyncMessage(String message, Session session){
2     Future<Void> future = session.getAsyncRemote().sendText(message);
3 }
```

Chapter 7. Injection Support

As required in Java API for WebSocket, Tyrus supports full field, method and constructor injection using `javax.inject.Inject` into all websocket endpoint classes as well as the use of the interceptors on these classes. Except this, Tyrus also supports some of the EJB annotations. Currently `javax.ejb.Stateful`, `javax.ejb.Singleton` and `javax.ejb.Stateless` annotations are supported.

7.1. `javax.inject.Inject` sample

The following example presents how to inject a bean to the `javax.websocket.server.ServerEndpoint` annotated class using `javax.inject.Inject`. Class `InjectedSimpleBean` gets injected into class `SimpleEndpoint` on line 15.

Example 7.1. Injecting bean into `javax.websocket.server.ServerEndpoint`

```
 1 public class InjectedSimpleBean {
 2
 3     private static final String TEXT = " (from your server)";
 4
 5     public String getText() {
 6         return TEXT;
 7     }
 8 }
 9
10 @ServerEndpoint(value = "/simple")
11 public class SimpleEndpoint {
12
13     private boolean postConstructCalled = false;
14
15     @Inject
16     InjectedSimpleBean bean;
17
18     @OnMessage
19     public String echo(String message) {
20         return String.format("%s%s", message, bean.getText());
21     }
22 }
```

7.2. EJB sample

The following sample presents how to turn `javax.websocket.server.ServerEndpoint` annotated class into `javax.ejb.Singleton` and use interceptor on it.

Example 7.2. Echo sample server endpoint.

```
 1 @ServerEndpoint(value = "/singleton")
 2 @Singleton
 3 @Interceptors(LoggingInterceptor.class)
 4 public class SingletonEndpoint {
 5
 6     int counter = 0;
 7     public static boolean interceptorCalled = false;
 8
 9     @OnMessage
10     public String echo(String message) {
11         return interceptorCalled ? String.format("%s%s", message, counter++) :
12     }
13 }
14
15 public class LoggingInterceptor {
16
17     @AroundInvoke
18     public Object manageTransaction(InvocationContext ctx) throws Exception {
19         SingletonEndpoint.interceptorCalled = true;
20         Logger.getLogger(getClass().getName()).info("LOGGING.");
21         return ctx.proceed();
22     }
23 }
24
```

Chapter 8. Tyrus proprietary configuration

Following settings do have influence on Tyrus behaviour and are *NOT* part of WebSocket specification. If you are using following configurable options, your application might not be easily transferable to other WebSocket API implementation.

8.1. Client-side SSL configuration

When accessing "wss" URLs, Tyrus client will pick up whatever keystore and truststore is actually set for current JVM instance, but that might not be always convenient. WebSocket API does not have this feature (yet, seeWEBSOCKET_SPEC-210 [https://java.net/jira/browse/WEBSOCKET_SPEC-210]), so Tyrus exposed SSLEngineConfigurator [https://grizzly.java.net/docs/2.3/apidocs/org/glassfish/grizzly/ssl/SSLEngineConfigurator.html] class from Grizzly which can be used for specifying all SSL parameters to be used with current client instance. Additionally, WebSocket API does not have anything like a client, only WebSocketContainer and it does not have any properties, so you need to use Tyrus specific class - ClientManager [https://tyrus.java.net/apidocs/1.3.3/org/glassfish/tyrus/client/ClientManager.html].

```
 1 final ClientManager client = ClientManager.createClient();
 2
 3 System.getProperties().put("javax.net.debug", "all");
 4 System.getProperties().put(SSLContextConfigurator.KEY_STORE_FILE, "...");
 5 System.getProperties().put(SSLContextConfigurator.TRUST_STORE_FILE, "...");
 6 System.getProperties().put(SSLContextConfigurator.KEY_STORE_PASSWORD, "...");
 7 System.getProperties().put(SSLContextConfigurator.TRUST_STORE_PASSWORD, "...")
 8 final SSLContextConfigurator defaultConfig = new SSLContextConfigurator();
 9
10 defaultConfig.retrieve(System.getProperties());
11     // or setup SSLContextConfigurator using its API.
12
13 SSLEngineConfigurator sslEngineConfigurator =
14     new SSLEngineConfigurator(defaultConfig, true, false, false);
15 client.getProperties().put(GrizzlyEngine.SSL_ENGINE_CONFIGURATOR,
16     sslEngineConfigurator);
17 client.connectToServer(... , ClientEndpointConfig.Builder.create().build(),
18     new URI("wss://localhost:8181/sample-echo/echo"));
19 }
```

8.2. Asynchronous connectToServer methods

WebSocketContainer.connectToServer(...) methods are by definition blocking - declared exceptions needs to be thrown after connection attempt is made and it returns Session instance, which needs to be ready for sending messages and invoking other methods, which require already estabilished connection.

Existing connectToServer methods are fine for lots of uses, but it might cause issue when you are designing application with highly responsible user interface. Tyrus introduces asynchronous variants to each connectToServer method (prefixed with "async"), which returns Future<Session>. These methods do only simple check for provided URL and the rest is executed in separate thread. All exceptions thrown during this phase are reported as cause of ExecutionException thrown when calling Future<Session>.get().

Asynchronous connect methods are declared on Tyrus implementation of WebSocketContainer called ClientManager.

```
1 ClientManager client = ClientManager.createClient();
2   final Future<Session> future = client.asyncConnectToServer(ClientEndpoint.cl
3   try {
4     future.get();
5   } catch (...) {
6 }
```

ClientManager contains async alternative to each connectToServer method.

8.3. Optimized broadcast

One of the typical usecases we've seen so far for WebSocket server-side endpoints is broadcasting messages to all connected clients, something like:

```
1 @OnMessage
2 public void onMessage(Session session, String message) throws IOException {
3   for (Session s : session.getOpenSessions()) {
4     s.getBasicRemote().sendText(message);
5   }
6 }
```

Executing this code might cause serious load increase on your application server. Tyrus provides optimized broadcast implementation, which takes advantage of the fact, that we are sending exactly same message to all clients, so dataframe can be created and serialized only once. Furthermore, Tyrus can iterate over set of opened connections faster than Session.getOpenSession().

```
1 @OnMessage
2 public void onMessage(Session session, String message) {
3   ((TyrusSession) session).broadcast(message);
4 }
```

Unfortunately, WebSocket API forbids anything else than Session in @OnMessage annotated method parameter, so you cannot use TyrusSession there directly and you might need to perform instanceof check.

8.4. Incoming buffer size

Sevlet container buffers incoming WebSocket frames and there must a size limit to precede OutOfMemory Exception and potentially DDoS attacks.

Configuration property is named `"org.glassfish.tyrus.servlet.incoming-buffer-size"` and you can set it in web.xml (this particular snipped sets the buffer size to 17000000 bytes (~16M payload):

```
1 <web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="ht
2   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xm
3
4   <context-param>
5     <param-name>org.glassfish.tyrus.servlet.incoming-buffer-size</param-name>
6     <param-value>17000000</param-value>
7   </context-param>
8 </web-app>
```

Default value is 4194315, which correspond to 4M plus few bytes to frame headers, so you should be able to receive up to 4M long message without the need to care about this property.

Same issue is present on client side. There you can set this property via ClientManager:

```
1 ClientManager client = ClientManager.createClient();
2 client.getProperties().put("org.glassfish.tyrus.incomingBufferSize", 6000000);
3 client.connectToServer( ... )
```