IBM Software
WebSphere

An IBM Proof of Technology

WebSockets

Contents

Lab 6 WebSockets

In this lab we will learn how to use WebSocket via a sample application demonstrating five different patterns:

1. An annotated endpoint
2. Use of encoders and decoders
3. Programmatically configured extended endpoint
4. Ping and pong with client.
5. Parameter processing

Please refer to the following table for file and resource location references on different operating systems.

Location Ref.	OS	Absolute Path
{LAB_HOME}	Windows	`C:\WLP_<version>`
	Linux	`~/WLP_<version>`
	Mac OSX	

6.1 Prerequisites

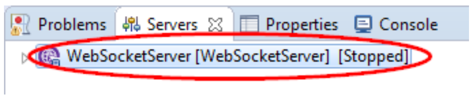
The following preparation must be completed prior to beginning this lab:

1. Complete the Getting Started lab to set up the lab environment, and learn how to create a server using Eclipse with WebSphere Developer Tools (WDT).
2. Optional: complete the Simple Development lab if you need a refresher on how to use Eclipse and WDT.

6.2 Create a new server

__1. Start Eclipse.

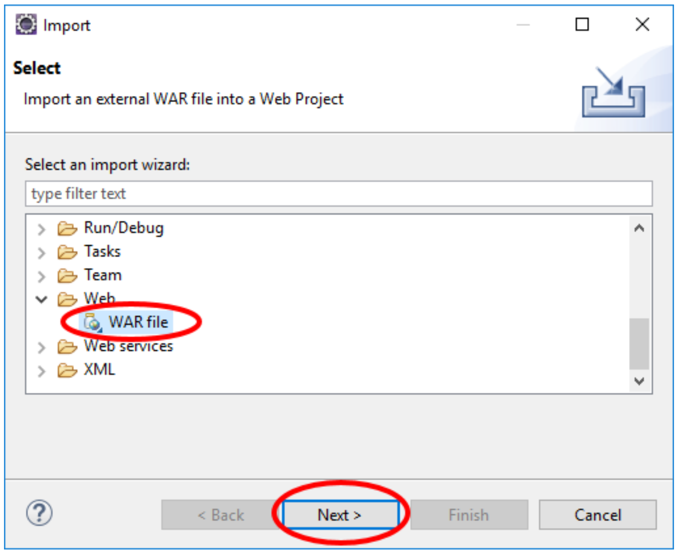
__2. Using the procedure learned in the Getting Started lab, create a new Liberty profile application server called **WebSocketServer**.



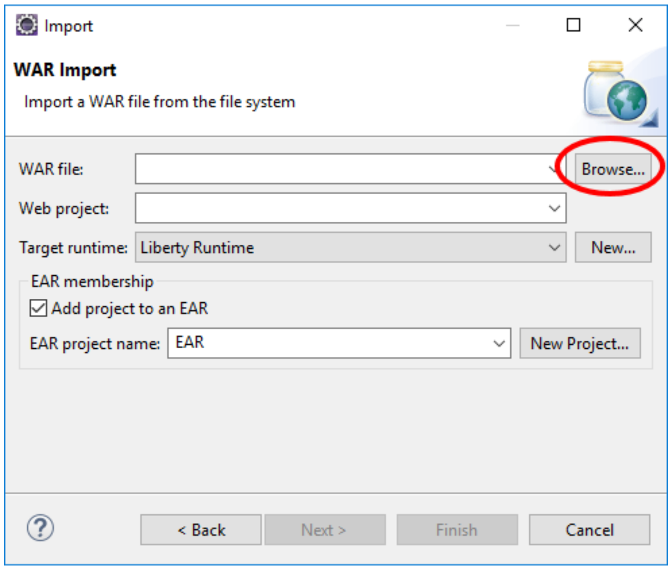
WebSocketServer [WebSocketServer] [Stopped]

6.3 Import the WebSocket sample application

__3. Click **File > Import** from the Eclipse menu. Expand **Web** and select **WAR file** on the Import window, then click **Next**.



__4. Click **Browse** on the WAR Import view.



WAR file:

Web project:

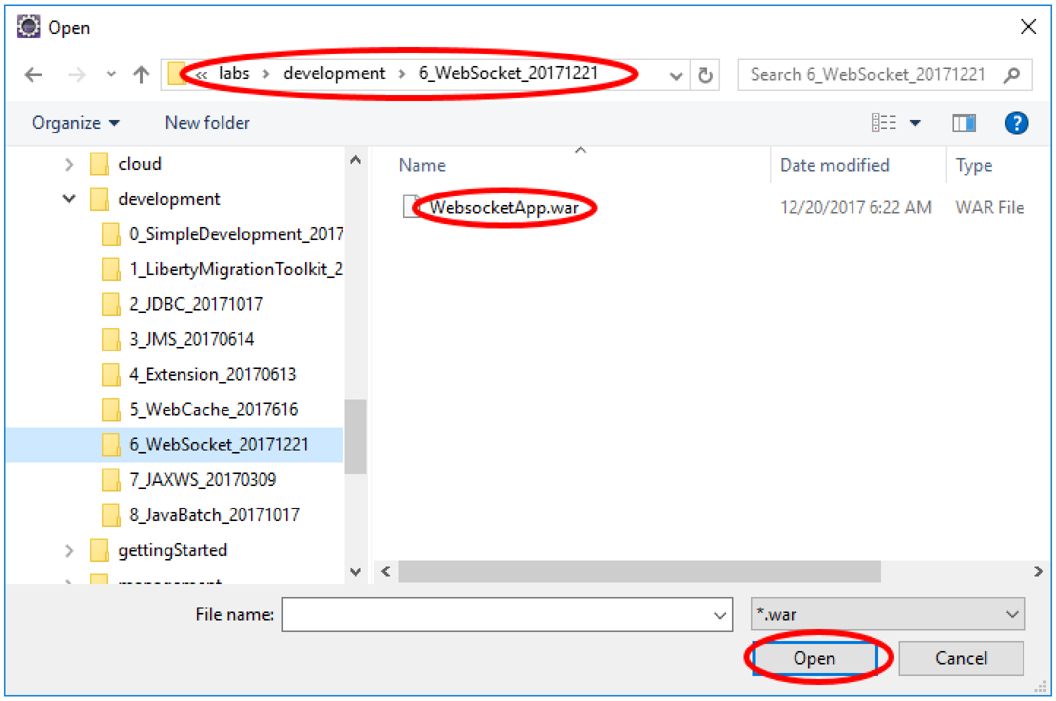
Target runtime: Liberty Runtime

EAR membership

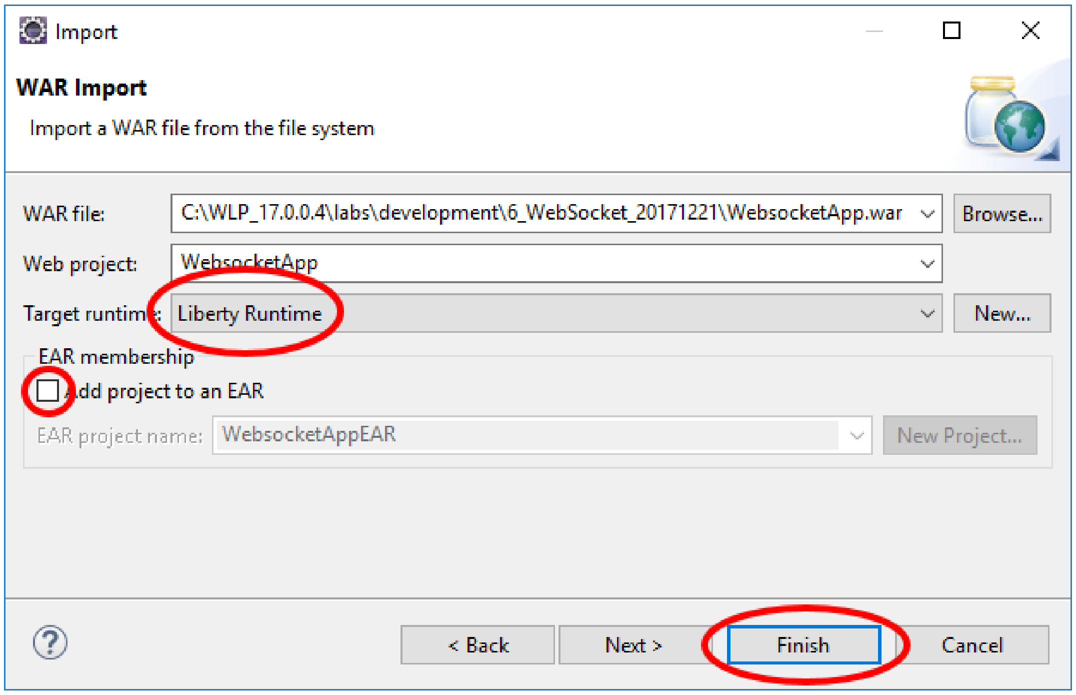
☑ Add project to an EAR

EAR project name: EAR

__5. Use the Open dialog to locate the `{LAB_HOME}\labs\development\6_WebSocket_<timestamp>\WebsocktApp.war` file and click **Open**.



__6. Clear the **Add project to an EAR** checkbox, and verify the **Target runtime** is set to **Liberty Runtime** or the appropriate Target runtime, and click **Finish**.


Import

WAR Import

Import a WAR file from the file system

WAR file: C:\WLP_17.0.0.4\labs\development\6_WebSocket_20171221\WebsocketApp.war

Web project: WebsocketApp

Target runtime: Liberty Runtime

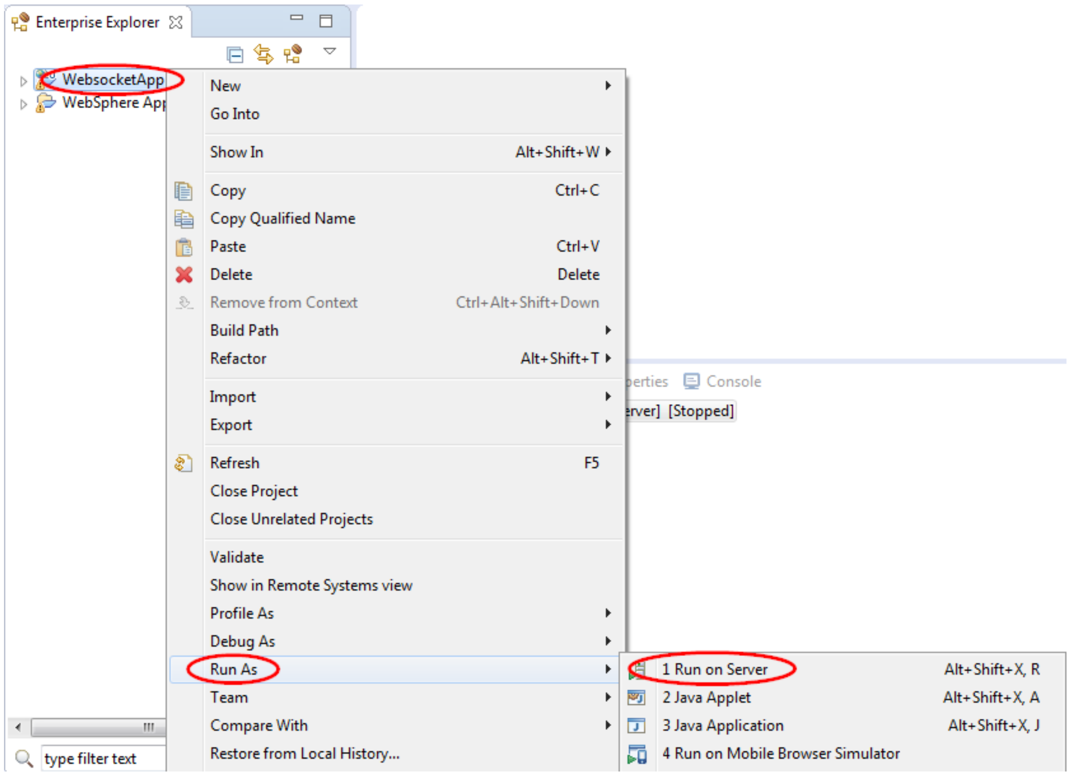
EAR membership

[ ] Add project to an EAR

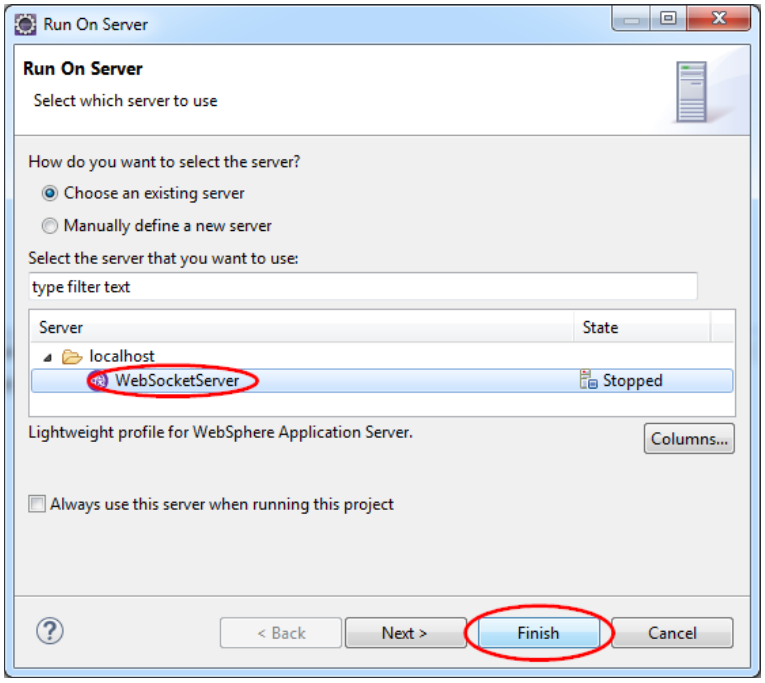
EAR project name: WebsocketAppEAR

6.4 Run the sample application

__1. In the Enterprise Explorer view, right click the **WebsocketApp** project, then click **Run As > Run on server**.

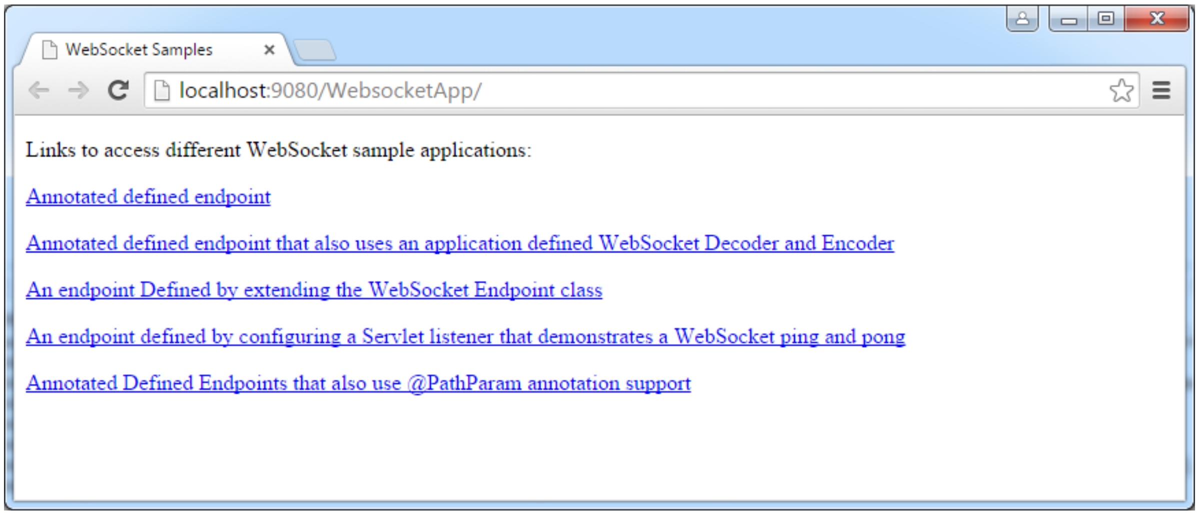


__2. On the Run On Server window, select **WebSocketServer** from the **Server** list, then click **Finish**.


Run On Server
Run On Server
Select which server to use
How do you want to select the server?
Choose an existing server
Manually define a new server
Select the server that you want to use:
type filter text
Server
State
localhost
WebSocketServer
Stopped
Lightweight profile for WebSphere Application Server.
Columns...
Always use this server when running this project
< Back
Next >
Finish
Cancel

__3. After the server starts running, Eclipse may automatically bring up a browser for you to interact with the application. However, the browser in Eclipse does not yet support `WebsocketApp`. You must switch to a browser that supports `WebsocketApp`, such as a recent version of Chrome, Firefox, or Internet Explorer.

__4. Start a browser that supports WebsocketApp, and go to http://localhost:9080/WebsocketApp.



Links to access different WebSocket sample applications:

Annotated defined endpoint

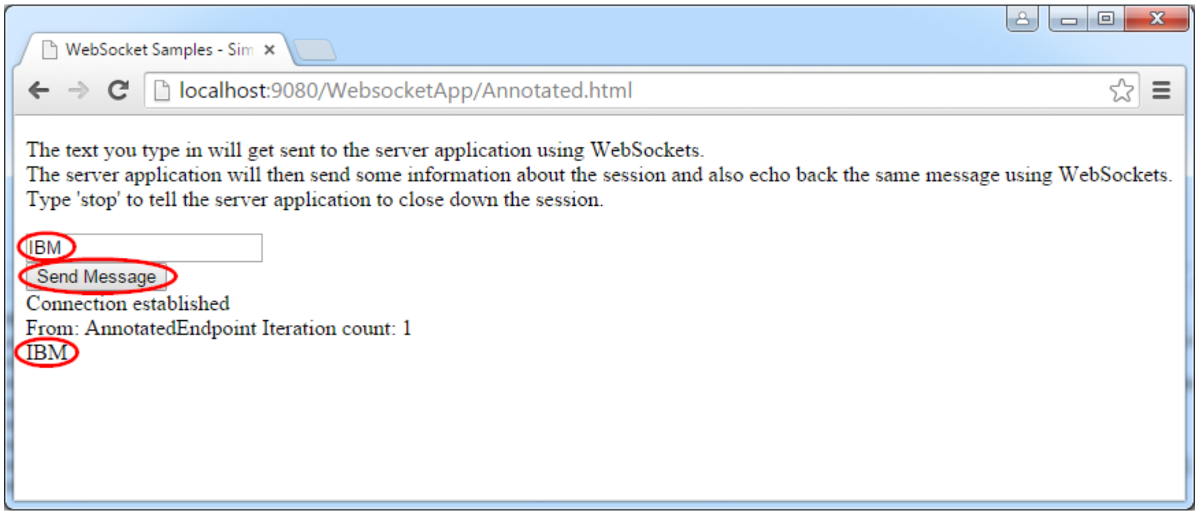
Annotated defined endpoint that also uses an application defined WebSocket Decoder and Encoder

An endpoint Defined by extending the WebSocket Endpoint class

An endpoint defined by configuring a Servlet listener that demonstrates a WebSocket ping and pong

Annotated Defined Endpoints that also use @PathParam annotation support

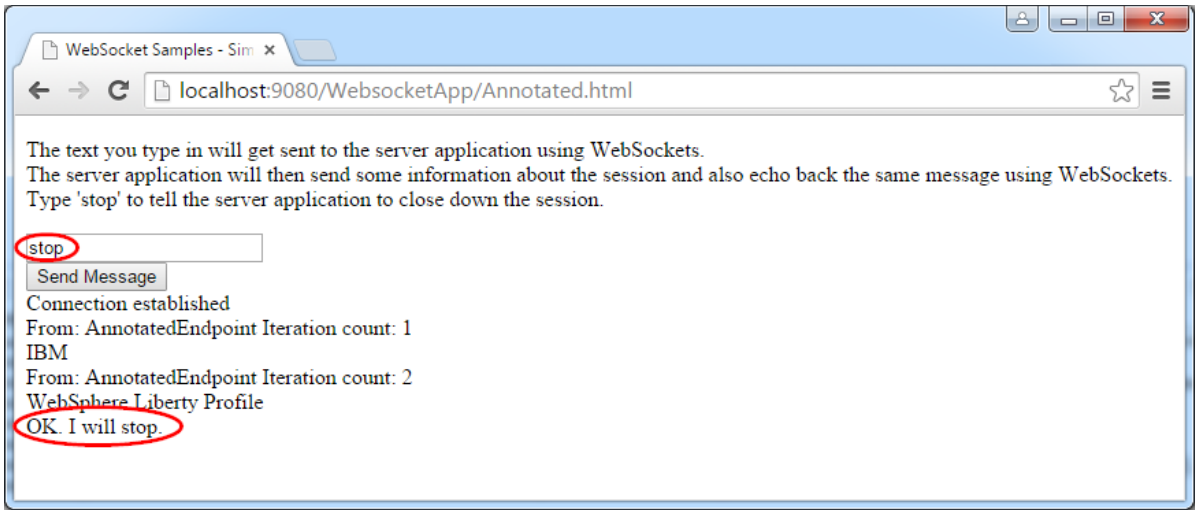
__5. Click on any of the links and follow the instructions for demonstration. For example, when you enter a message in the text box, the browser sends the message to the server via the WebSocket protocol. The server then echoes the message back.



The text you type in will get sent to the server application using WebSockets.
The server application will then send some information about the session and also echo back the same message using WebSockets.
Type 'stop' to tell the server application to close down the session.

Connection established
From: AnnotatedEndpoint Iteration count: 1
IBM

__6. Type `stop` in the text box in each section of the sample to shut down the session.



The text you type in will get sent to the server application using WebSockets.
The server application will then send some information about the session and also echo back the same message using WebSockets.
Type 'stop' to tell the server application to close down the session.

stop

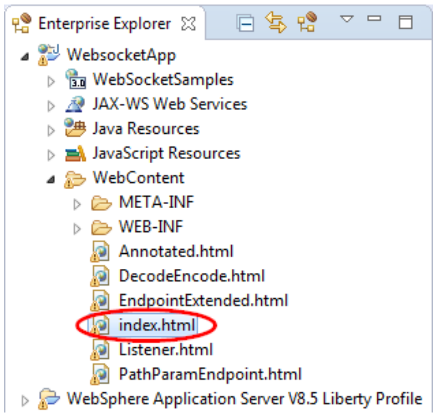
Send Message

```
Connection established
From: AnnotatedEndpoint Iteration count: 1
IBM
From: AnnotatedEndpoint Iteration count: 2
WebSphere Liberty Profile
OK. I will stop.
```

6.5 Code walk-through

6.5.1 `index.html`

The file `index.html` is the main entry point to the `WebsocketApp` application. It can found in the Enterprise Explorer view at **`WebsocketApp > WebContent > index.html`**.



Double click `index.html` to bring it up in the editor. Note that it contains links to the five different patterns of WebSocket protocol usage:

```
<!DOCTYPE html>
<html>
<head>
	<title>WebSocket Samples</title>
</head>
<body>
	<p />
	Links to access different WebSocket sample applications:
	<p />
	<a href="./Annotated.html">Annotated defined endpoint</a>
	<p />
	<a href="./DecodeEncode.html">Annotated defined endpoint that also uses an application defined
		WebSocket Decoder and Encoder</a>
	<p />
	<a href="./EndpointExtended.html">An endpoint Defined by extending the WebSocket Endpoint
		class</a>
	<p />
	<a href="./Listener.html">An endpoint defined by configuring a Servlet listener that
		demonstrates a WebSocket ping and pong</a>
	<p />
	<a href="./PathParamEndpoint.html">Annotated Defined Endpoints that also use @PathParam
		annotation support</a>
</body>
</html>
```

6.5.2 Simple Annotated endpoint

The simple annotated endpoint pattern demonstrates how to define a simple WebSocket endpoint to receive and send messages.

6.5.2.1 Annotated.html

Open the file `Annotated.html` in the editor. Review the source code:

```
<!DOCTYPE html>
<html>
<head>
	<title>WebSocket Samples - Simple Annotated Endpoint</title>
</head>
<body>
	<p />
	The text you type in will get sent to the server application using WebSockets.
	<br />The server application will then send some information about the session and also echo
	back the same message using WebSockets.
	<br />Type 'stop' to tell the server application to close down the session.
	<p />
	<div>
		<input id="inputmessage" type="text" />
	</div>
	<div>
		<input type="submit" value="Send Message" onclick="send()" />
	</div>
	<div id="messages"></div>
```

```
        <script language="javascript" type="text/javascript">
            var webSocket = new WebSocket('ws://' + window.document.location.host
                    + '/WebsocketApp/SimpleAnnotated');

            webSocket.onerror = function(event) {
                onError(event)
            };

            webSocket.onopen = function(event) {
                onOpen(event)
            };

            webSocket.onmessage = function(event) {
                onMessage(event)
            };


            function onMessage(event) {
                document.getElementById('messages').innerHTML += '<br />'
                    + event.data;
            }

            function onOpen(event) {
                document.getElementById('messages').innerHTML = 'Connection established';
            }

            function onError(event) {
                alert(event.data);
            }

            function send() {
                var txt = document.getElementById('inputmessage').value;
                webSocket.send(txt);
                return false;
            }
        </script>
</body>
</html>
```

Note that:

1 The HTML defines an `<input>` of type *`text`* and id *`inputmessage`* for you to type in a text string.

2 It defines an `<input>` of type *`submit`* that is displayed as a button with the label **Send Message**. When you click on the button, the method `send()` is called.

3 The supplied client side JavaScript

a Defines a variable named `webSocket` that is used to access the server via the URL ws://<host>/WebSocketApp/SimpleAnnotated.

b The are several callbacks defined for the WebSocket protocol:

i `onError`: for displaying any error message.

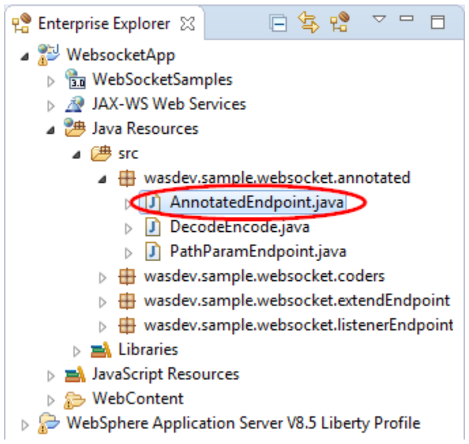
ii `onOpen`: to display a message when the WebSocket is opened. This is done during initialization of the page.

iii `onMessage`: to display any message returned by the server.

c The `send()` method sends the actual message that you entered in the text box to the server via the WebSocket protocol. Afterwards, the callbacks are called as the events occur. If nothing goes wrong, the `onMessage()` method is called when the server returns with a message.

6.5.2.2 AnnotatedEndpoint.java

In the Enterprise Explorer view, navigate to **WebsocketApp > Java Resrouces > src> wasdev.sample.websocket.annotated > AnnotatedEndpoint.java**


Enterprise Explorer
WebsocketApp
WebSocketSamples
JAX-WS Web Services
Java Resources
src
wasdev.sample.websocket.annotated
AnnotatedEndpoint.java
DecodeEncode.java
PathParamEndpoint.java
wasdev.sample.websocket.coders
wasdev.sample.websocket.extendEndpoint
wasdev.sample.websocket.listenerEndpoint
Libraries
JavaScript Resources
WebContent
WebSphere Application Server V8.5 Liberty Profile

Double click `AnnotatedEndpoint.java` to bring it up in the editor.

```java
package wasdev.sample.websocket.annotated;

import java.io.IOException;

import javax.websocket.CloseReason;
import javax.websocket.EndpointConfig;
import javax.websocket.OnClose;
import javax.websocket.OnError;
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;

// The ServerEndpoint annotation value is the name of the WebSocket Endpoint for this
    application/endpoint.
// JavaScript to access from a WebSocket capable browser would be:  ws://<Host Name>:<port>/<Context-
    Root>/SimpleAnnotated
@ServerEndpoint(value = "/SimpleAnnotated")
public class AnnotatedEndpoint {
    Session currentSession = null;
    int count = 0;
    // OnOpen will get called by WebSockets when the connection has been established successfully using
        WebSocket handshaking with
    // the HTTP Request - Response processing.
    @OnOpen
    public void onOpen(Session session, EndpointConfig ec) {
        // Store the WebSocket session for later use.
        currentSession = session;
    }

    // using the OnMessage annotation for this method will cause this method to get called by
        WebSockets when this connection has received
    // a WebSocket message from the other side of the connection.
    // The message is derived from the WebSocket frame payloads of one, and only one, WebSocket
        message.
    @OnMessage
    public void receiveMessage(String message) {

        try {
            count++;

            if (message.toLowerCase().equals("stop")) {
                // send a WebSocket message back to the other endpoint that says we will stop.
                currentSession.getBasicRemote().sendText("OK. I will stop.");

                // Sleep to let the other side get the message before stopping - a bit kludgy, but this
                    is just a sample!
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                }

                currentSession.close();
            } else {
                // send the message back to the other side with the iteration count.  Notice we can send
                    multiple message without having
                // to receive messages in between.
                currentSession.getBasicRemote().sendText("From: " + this.getClass().getSimpleName() + "
                    Iteration count: " + count);
                currentSession.getBasicRemote().sendText(message);
            }
        } catch (IOException ex) {
            // no error processing will be done for this sample
        }
    }

    // Using the OnClose annotation will cause this method to be called when the WebSocket Session is
        being closed.
    @OnClose
    public void onClose(Session session, CloseReason reason) {
        // no clean up is needed here for this sample
    }

    // Using the OnError annotation will cause this method to be called when the WebSocket Session has
        an error to report. For the Alpha version
    // of the WebSocket implentation on Liberty, this will not be called on error conditions.
    @OnError
    public void onError(Throwable t) {
        // no error processing will be done for this sample
    }
}
```

Note that:

1. The `@ServerEndpoint` annotation defines the URI for this endpoint, which is ws://<host>:<port>/<context-root>/SimpleAnnotated.
2. The `onOpen()` method is called when a new session is established.
3. The `onMessage()` method is called when a message is received from the client.
   a. If the message is `stop`, the current session is closed.
   b. Otherwise, the message is echoed back to the caller with a prefix `From` followed by the class name and iteration count.
4. Two other callbacks `onClose()` and `onError()` are currently unimplemented.

6.5.3 Decode encode

The *decode encode* pattern demonstrates how to write your own decoder and encoder for the messages.

6.5.3.1 `DecodeEncode.html`

Open `DecodeEncode.html` in an editor and examine its source. Note that this HTML is the same as `Annotated.html` except that the URI ends in `/DecodeEncode`.

6.5.3.2 `DecodeEncode.java`

Open `DecodeEncode.java` in the editor:

```
package wasdev.sample.websocket.annotated;

import java.io.IOException;

import javax.websocket.CloseReason;
import javax.websocket.EncodeException;
import javax.websocket.EndpointConfig;
import javax.websocket.OnClose;
import javax.websocket.OnError;
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;

import wasdev.sample.websocket.coders.FormatIn;

//The ServerEndpoint annotation value is the name of the WebSocket Endpoint for this
    application/endpoint.
//JavaScript to access from a WebSocket capable browser would be:  ws://<Host Name>:<port>/<Context-
    Root>/DecodeEncode
// there is an application defined Decoder and an application defined Encoder for messages of the given
    Decoder and Encoder type for this endpoint.
```

```
@ServerEndpoint(value = "/DecodeEncode",
                  decoders = { wasdev.sample.websocket.coders.DecoderOne.class },
                  encoders = { wasdev.sample.websocket.coders.EncoderOne.class })
public class DecodeEncode {
    Session currentSession = null;
    int count = 0;

    // OnOpen will get called by WebSockets when the connection has been established successfully using
        WebSocket handshaking with
    // the HTTP Request - Response processing.
    @OnOpen
    public void onOpen(Session session, EndpointConfig ec) {
        // Store the WebSocket session for later use.
        currentSession = session;
    }

    // using the OnMessage annotation for this method will cause this method to get called by
        WebSockets when this connection has received
    // a WebSocket message from the other side of the connection that could be decoded into an
        application defined FormatIn object.
    // The message is derived from the WebSocket frame payloads of one, and only one, WebSocket
        message.
    @OnMessage
    public void decodeTextSendBackEncodedText(FormatIn decodedObject) {

        count++;
        String message = decodedObject.getDecodedString();

        try {
            if (message.equals("STOP")) {

                // use sendText to avoid the encoder that is looking for String types on sendObject to
                    encode.
                // send a WebSocket message back to the other endpoint that says we will stop.
                currentSession.getBasicRemote().sendText("ok, I will stop");

                // Sleep to let the other side get the message before stopping - a bit kludgy, but this
                    is just a sample!
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                }

                currentSession.close();

            } else {
                // WebSockets will identify the "String" type as a type to encode by EncoderOne class,
                    and call the encoder
                currentSession.getBasicRemote().sendObject(message);
            }
        } catch (IOException e) {
            System.out.println("Caught unexpected IOException: " + e);
        } catch (EncodeException e) {
            System.out.println("Caught unexpected EncodeException: " + e);
        }
    }

    // Using the OnClose annotation will cause this method to be called when the WebSocket Session is
        being closed.
    @OnClose
    public void onClose(Session session, CloseReason reason) {
        // no clean up is needed here for this sample
    }

    // Using the OnError annotation will cause this method to be called when the WebSocket Session has
        an error to report. For the Alpha version
    // of the WebSocket implentation on Liberty, this will not be called on error conditions.
    @OnError
    public void onError(Throwable t) {
        // no error processing will be done for this sample
    }
}
```

Note that:

1 The `@ServerEndpoint` annotation defines the URI that ends in `/Decodeencode`, and also the decoder and encoder classes.

```
@ServerEndpoint(value = "/DecodeEncode",
    decoders = { wasdev.sample.websocket.coders.DecoderOne.class },
    encoders = { wasdev.sample.websocket.coders.EncoderOne.class })
```

2 The `@OnMessage` annotation is applied to the `decodeTextSendBackEncodedText()` method to process the message from the client. The input parameter `decodeObject` is of type `FormatIn`, which is the type of the decoded message after the decoder `DecoderOne` decodes the input message.

6.5.3.3 DecodeOne.java

Open `DecoderOne.java` in the editor. Note that it is a very simple decoder that uses the `FormatIn.java` class to decode a message.

```
package wasdev.sample.websocket.coders;

import javax.websocket.DecodeException;
import javax.websocket.Decoder;
import javax.websocket.EndpointConfig;

// This is coded to be a Text type decoder, and it will decode incoming messages into object of type
    FormatIn.
public class DecoderOne implements Decoder.Text<FormatIn> {

    public DecoderOne() {
    }

    @Override
    public void destroy() {
    }

    @Override
    public void init(EndpointConfig config) {
    }

    @Override
    public FormatIn decode(String s) throws DecodeException {
        // For the incoming String, create a new FormatIn object and allow this object to decode the
            String data.
        FormatIn f = new FormatIn();
        f.doDecoding(s);
        return f;
    }

    @Override
    public boolean willDecode(String s) {
        // For the sample, we will always try to encode whatever String we receive
        return true;
    }
}
```

6.5.3.4 `EncodeOne.java`

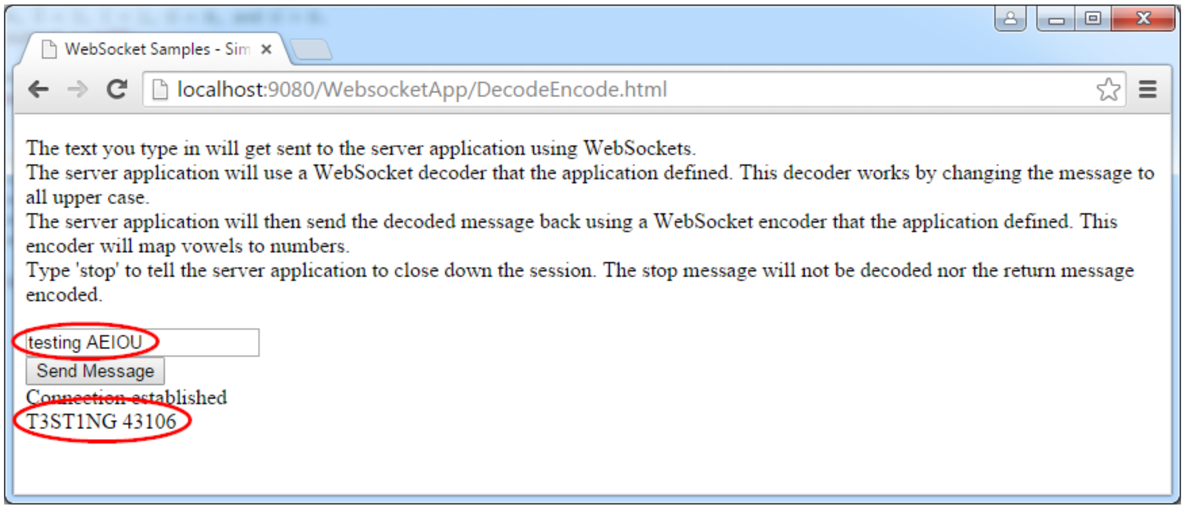
Open `EncodeOne.java` in the editor. Note that it is a very simple encoder that transforms vowels to numbers:

- `A` to `4`
- `E` to `3`
- `I` to `1`
- `O` to `0`
- `U` to `6`

```
package wasdev.sample.websocket.coders;

import javax.websocket.EncodeException;
import javax.websocket.Encoder;
import javax.websocket.EndpointConfig;

// This is coded to be a Text type encoder, and it will encode outgoing Strings that we sent using the
    sendObject method.
public class EncoderOne implements Encoder.Text<String> {

    @Override
    public void destroy() {
    }

    @Override
    public void init(EndpointConfig arg0) {
    }

    @Override
    public String encode(String s) throws EncodeException {
        // encoding will be to replace the upper case vowels with numbers.
        // A = 4, E = 3, I = 1, O = 0, and U = 6.
        String output = null;

        if (s == null) {
            return "";
        }

        output = s.replace("A", "4");
        output = output.replace("E", "3");
        output = output.replace("I", "1");
        output = output.replace("O", "0");
        output = output.replace("U", "6");

        return output;
    }
}
```

Rerun the sample using these variations for input text:

- Lower case input should be converted to upper case
- The vowels *AEIOU* should be converted to their numeric equivalent.



The text you type in will get sent to the server application using WebSockets.
The server application will use a WebSocket decoder that the application defined. This decoder works by changing the message to all upper case.
The server application will then send the decoded message back using a WebSocket encoder that the application defined. This encoder will map vowels to numbers.
Type 'stop' to tell the server application to close down the session. The stop message will not be decoded nor the return message encoded.

testing AEIOU

Connection established
T3ST1NG 43106

6.5.4 Extended Endpoint Pattern

Using the extended endpoint pattern enables you to define a WebSocket endpoint programmatically rather than via annotations.

6.5.4.1 `EndpointExtended.html`

This HTML is the same as `Annotated.html` except that the WebSocket URI ends in `/ExtendedEndpoint`.

6.5.4.2 `EEServerApplicationConfig.java`

This class implements the `ServerApplicationConfig` interface enabling you to define the endpoint configurations programmatically.

```
package wasdev.sample.websocket.extendEndpoint;

import java.util.HashSet;
import java.util.Set;

import javax.websocket.Endpoint;
import javax.websocket.server.ServerApplicationConfig;
import javax.websocket.server.ServerEndpointConfig;
```

```
public class EEServerApplicationConfig implements ServerApplicationConfig {

    @Override
    public Set<Class<?>> getAnnotatedEndpointClasses(Set<Class<?>> arg0) {
        // Accept for usage any endpoint classes that have been found for this application.
        return arg0;
    }

    @Override
    public Set<ServerEndpointConfig> getEndpointConfigs(Set<Class<? extends Endpoint>> arg0) {
        // Intialize and return the endpoint configuration object that will be used for our coded
            endpoint in this application.
        Set<ServerEndpointConfig> configs = new HashSet<ServerEndpointConfig>();
        EEServerEndpointConfig config = new EEServerEndpointConfig();
        configs.add(config);
        return configs;
    }
}
```

The `getAnnotatEndpointClasses()` method enables you decided which endpoint classes to be used. In our example, we allow all valid endpoint classes that are found by the runtime. The `getEndpointConfigs()` method allows you to decide the configurations for each endpoint.

6.5.4.3 EEServerEndpointConfig.java

This class implements the `ServerEndpointConfig` interface to programmatically define the configuration of an endpoint. Note the `getEndpointClass()` method returns a valid endpoint class `ExtendedEndpoint`. The `getPaht()` method returns the URI for this endpoint.

```
package wasdev.sample.websocket.extendEndpoint;

import java.util.List;
import java.util.Map;

import javax.websocket.Decoder;
import javax.websocket.Encoder;
import javax.websocket.Extension;
import javax.websocket.server.ServerEndpointConfig;

public class EEServerEndpointConfig implements ServerEndpointConfig {

    // the uri path to use to get to this endpoint.
    // JavaScript to access from a WebSocket capable browser would be:  ws://<Host
        Name>:<port>/<Context-Root>/ExtendedEndpoint
    String uriPath = "/ExtendedEndpoint";

    public EEServerEndpointConfig() {
        // no-arg constructor
    }

    @Override
    public Class<?> getEndpointClass() {
        return ExtendedEndpoint.class;
    }
    @Override
    public String getPath() {
        return uriPath;
    }

    @Override
    public Configurator getConfigurator() {
        ServerEndpointConfig.Configurator x = new ServerEndpointConfig.Configurator();
        return x;
    }

    @Override
    public Map<String, Object> getUserProperties() {
        return null;
    }

    @Override
    public List<Extension> getExtensions() {
        return null;
    }

    @Override
    public List<String> getSubprotocols() {
        return null;
    }

    @Override
    public List<Class<? extends Decoder>> getDecoders() {
        return null;
    }

    @Override
    public List<Class<? extends Encoder>> getEncoders() {
        return null;
    }
}
```

6.5.4.4 ExtendedEndpoint.java

This class is the actual endpoint, and defines the message handler used to handle message.

```
package wasdev.sample.websocket.extendEndpoint;

import javax.websocket.CloseReason;
import javax.websocket.Endpoint;
import javax.websocket.EndpointConfig;
import javax.websocket.MessageHandler;
import javax.websocket.Session;

public class ExtendedEndpoint extends Endpoint {

    // onOpen will get called by WebSockets when the connection has been established successfully using
        WebSocket handshaking with
    // the HTTP Request - Response processing.
    @Override
    public void onOpen(final Session session, EndpointConfig ec) {
        // Set up the object that will receive messages sent from the other side of this connection.
        MessageHandler.Whole<String> handler = new EEMessageHandler(session);
        session.addMessageHandler(handler);
    }

    // onClose will be called when the WebSocket Session is being closed.
    @Override
    public void onClose(Session session, CloseReason reason) {}

    // onError will be called when the WebSocket Session has an error to report. For the Alpha version
    // of the WebSocket implentation on Liberty, this will not be called on error conditions.
    @Override
    public void onError(Session session, Throwable thr) {}
}
```

6.5.4.5 EEMessageHandler.java

This class defines the message handler used to process `String` inputs. Note the `onMessage()` method uses `String` type input parameter.

```
package wasdev.sample.websocket.extendEndpoint;

import java.io.IOException;
import javax.websocket.MessageHandler;
import javax.websocket.Session;

public class EEMessageHandler implements MessageHandler.Whole<String> {
    int count = 0;
    Session currentSession = null;

    public EEMessageHandler(Session session) {
        // store the session so our onMessage method can use it later
        currentSession = session;
    }

    // onMessage will be called by WebSockets when this connection has received
    // a WebSocket message from the other side of the connection.
    // The message is derived from the WebSocket frame payloads of one, and only one, WebSocket
        message.
    @Override
    public void onMessage(String message) {
        try {
            count++;

            if (message.toLowerCase().equals("stop")) {
                // send a WebSocket message back to the other endpoint that says we will stop.
                currentSession.getBasicRemote().sendText("OK. I will stop.");

                // Sleep to let the other side get the message before stopping - a bit kludgy, but this
                    is just a sample!
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                }

                currentSession.close();
            } else {
                // send the message back to the other side with the iteration count.  Notice we can send
                    multiple message without having
                // to receive messages in between.
                currentSession.getBasicRemote().sendText("From: " + this.getClass().getSimpleName()
                + "  Iteration count: " + count);
            currentSession.getBasicRemote().sendText(message);
         }
      } catch (IOException ex) {
         // no error processing will be done for this sample
      }
   }
}
```

6.5.5 Listener Pattern

The listener pattern demonstrates the server pinging the web browser. This is useful for keep-alive. It also demonstrates how to programmatically add a new endpoint during web application servlet context initialization.

6.5.5.1 `Listener.html`

This HTML is similar to `Annotated.html` except that the WebSocket URI ends in `/listenerEndpoint`. Read the introduction text which explains how the sample is supposed to work:

1. After you enter some text and click on the **Request Ping** button, a message is sent to the server.
2. The server sends a *ping* to the browser, and upon receipt the browser immediately sends back a *pong* automatically. This is a feature of a browser that supports WebSocket.
3. After the server receives the *pong*, the reply HTML is sent back to the browser for you to verify the *pong* was successfully received. Note that there is currently no API in the browser for JavaScript to intercept pings or pongs.

6.5.5.2 `ServletListener.java`

This class programmatically creates a new endpoint during servlet context initialization in the `contextInitialized()` method.

6.5.5.3 `ListenerEndpointConfig.java`

This class implements the endpoint configuration for the `ListenerEndpoint` endpoint class.

6.5.5.4 `ListenerEndpoint.java`

This class implements the WebSocket endpoint for the sample application. Note that in the `onOpen()` method, two message handlers are created.

1. The first message handler processes a `PongMessage` from the browser, and returns the verification that a *pong* message was received via HTML to the browser.

2. The second message handler processes the string from the text box entered by the user. It initiates a *ping* message back to the browser. When the browser returns with a *pong*, the first message handler will handle the *pong*.

6.5.6 Path Parameter Pattern

This pattern demonstrates how to process path parameters via annotations.

6.5.6.1 `PathParamEndpoint.html`

This HTML sends a WebSocket URL containing path parameters:
ws:<host>:<port>/WebsocketApp/SimplePathParam/rentals/LakeViewProperties/C/111/2/299999.99/true/199.45

6.5.6.2 `PathParamEndpoint.java`

Note the annotation that defines the path parameters:

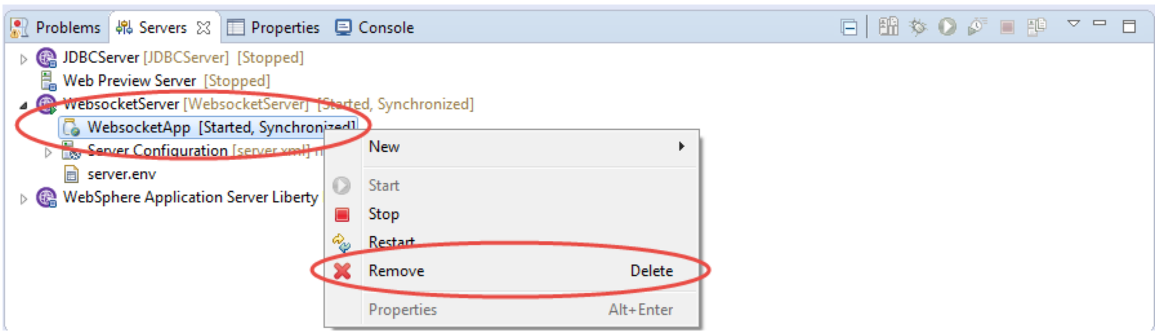
```
@ServerEndpoint(value = "/SimplePathParam/rentals/{name}/{building}/{home-number}/{no-of-
rooms}/{property-val}/{pets-allowed}/{maintenance-fee}")
```

Also note how the parameters appear in the method that processes the message:

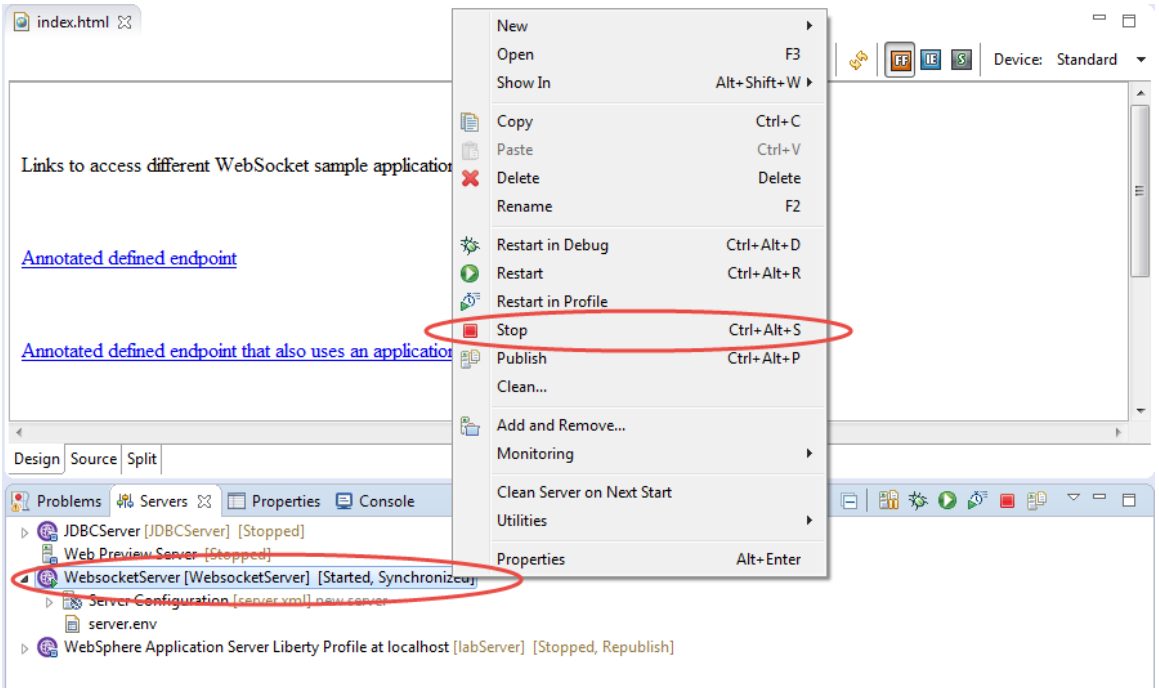
```
@OnMessage
public String echoText(String message,
       @PathParam("pets-allowed") Boolean isPetsAllowed,
       @PathParam("property-val") Double propertyValue,
       @PathParam("name") String name,
       @PathParam("building") char building,
       @PathParam("home-number") Integer homeNumber,
       @PathParam("no-of-rooms") short noOfRooms,
       @PathParam("maintenance-fee") float maintFee) {

       . . .

}
```

6.6 Clean up after Lab

__1. Remove Websocket Application from the server. Right click on the **WebsocketApp** application and select **Remove**


Problems
Servers
Properties
Console
JDBCServer [JDBCServer] [Stopped]
Web Preview Server [Stopped]
WebsocketServer [WebsocketServer] [Started, Synchronized]
WebsocketApp [Started, Synchronized]
Server Configuration [server.xml]
server.env
WebSphere Application Server Liberty
New
Start
Stop
Restart
Remove Delete
Properties Alt+Enter

__2. Stop the WebsocketServer. Right click on the **WebsocketServer** and select **Stop** to stop the server


index.html
Links to access different WebSocket sample application
Annotated defined endpoint
Annotated defined endpoint that also uses an application
Design
Source
Split
Device: Standard
New
Open F3
Show In Alt+Shift+W
Copy Ctrl+C
Paste Ctrl+V
Delete Delete
Rename F2
Restart in Debug Ctrl+Alt+D
Restart Ctrl+Alt+R
Restart in Profile
Stop Ctrl+Alt+S
Publish Ctrl+Alt+P
Clean...
Add and Remove...
Monitoring
Clean Server on Next Start
Utilities
Properties Alt+Enter
Problems
Servers
Properties
Console
JDBCServer [JDBCServer] [Stopped]
Web Preview Server [Stopped]
WebsocketServer [WebsocketServer] [Started, Synchronized]
Server Configuration [server.xml] new server
server.env
WebSphere Application Server Liberty Profile at localhost [labServer] [Stopped, Republish]

Appendix A. The file Notices

Appendix B. Trademarks and copyrights

NOTES

NOTES

WebSphere® software