

Chapter 13- Web Sockets

While there are quite a few WebSocket servers available we have chosen Netty to showcase assembling a custom server. And Jetty which is a prevalent standards based Servlet container which also supports WebSocket protocols.

Building a Multiplication WebSocket Server with Netty

Netty is a server framework using java.nio it is speculated that one netty instance can theoretically support upto 1,00,000 connections. The explanation of Netty is beyond the scope of this book. We will be assembling a multiplication WebSocket Server using Netty and JDK1.6 in this chapter. The concept for the example is very simple the client in our case an html page will open a websocket connection to the server and send a user entered integer number to it. The server will square the number and send it back which will be displayed on the browser.

We will be referring to the lone Netty jar and will create 3 classes in an eclipse command line java project. Let's create the classes one by one.

WebSocketServer.java Code

```
package com.ayna;

import java.net.InetSocketAddress;
import java.util.concurrent.Executors;

import org.jboss.netty.bootstrap.ServerBootstrap;
import org.jboss.netty.channel.socket.nio.NioServerSocketChannelFactory;

/**
 * An HTTP server which serves Web Socket requests at:
 *
 * http://localhost:9000/websocket
 */
public class WebSocketServer {
    public static void main(String[] args) {

        int port = 9000;

        // Configure the server.
        ServerBootstrap bootstrap = new ServerBootstrap(
            new NioServerSocketChannelFactory(
                Executors.newCachedThreadPool(),
                Executors.newCachedThreadPool()));

        // Set up the event pipeline factory.
```

```

bootstrap.setPipelineFactory(new WebSocketServerPipelineFactory());

// Bind and start to accept incoming connections.
bootstrap.bind(new InetSocketAddress(port));

System.out.println("WebSocket Server Started");
System.out.println("WebSocket Accessible on
http://localhost:"+port+WebSocketServerHandler.WEBSOCKET_PATH);
}
}

```

This is the main class. It will start a WebSocket server accessible on <http://localhost:9000/websocket>. Usually to make things secure we should implement same-origin constraints in the client-server web application, which means only those pages which are loaded from the same origin as the websocket server can call the websocket server. In this example we have not implemented any such mechanism so this server should ideally not be used as is for internet applications but rather intranet/enterprise applications.

WebSocketServerHandler.java Code

```

package com.ayna;

import static org.jboss.netty.handler.codec.http.HttpHeaders.*;
import static org.jboss.netty.handler.codec.http.HttpHeaders.Names.*;
import static org.jboss.netty.handler.codec.http.HttpHeaders.Values.*;
import static org.jboss.netty.handler.codec.http.HttpMethod.*;
import static org.jboss.netty.handler.codec.http.HttpResponseStatus.*;
import static org.jboss.netty.handler.codec.http.HttpVersion.*;

import org.jboss.netty.buffer.ChannelBuffer;
import org.jboss.netty.buffer.ChannelBuffers;
import org.jboss.netty.channel.ChannelFuture;
import org.jboss.netty.channel.ChannelFutureListener;
import org.jboss.netty.channel.ChannelHandlerContext;
import org.jboss.netty.channel.ChannelPipeline;
import org.jboss.netty.channel.ExceptionEvent;
import org.jboss.netty.channel.MessageEvent;
import org.jboss.netty.channel.SimpleChannelUpstreamHandler;
import org.jboss.netty.handler.codec.http.DefaultHttpResponse;
import org.jboss.netty.handler.codec.http.HttpHeaders;
import org.jboss.netty.handler.codec.http.HttpRequest;
import org.jboss.netty.handler.codec.http.HttpResponse;
import org.jboss.netty.handler.codec.http.HttpResponseStatus;
import org.jboss.netty.handler.codec.http.HttpHeaders.Names;
import org.jboss.netty.handler.codec.http.HttpHeaders.Values;
import org.jboss.netty.handler.codec.http.websocket.DefaultWebSocketFrame;
import org.jboss.netty.handler.codec.http.websocket.WebSocketFrame;
import org.jboss.netty.handler.codec.http.websocket.WebSocketFrameDecoder;

```

```

import org.jboss.netty.handler.codec.http.websocket.WebSocketFrameEncoder;
import org.jboss.netty.util.CharsetUtil;

public class WebSocketServerHandler extends SimpleChannelUpstreamHandler {

    public static final String WEBSOCKET_PATH = "/websocket";

    @Override
    public void messageReceived(ChannelHandlerContext ctx, MessageEvent e) throws Exception {
        Object msg = e.getMessage();
        if (msg instanceof HttpRequest) {
            handleHttpRequest(ctx, (HttpRequest) msg);
        } else if (msg instanceof WebSocketFrame) {
            handleWebSocketFrame(ctx, (WebSocketFrame) msg);
        }
    }

    private void handleHttpRequest(ChannelHandlerContext ctx, HttpRequest req) {
        // Allow only GET methods.
        if (req.getMethod() != GET) {
            sendHttpResponse(
                ctx, req, new DefaultHttpResponse(HTTP_1_1, FORBIDDEN));
            return;
        }

        // Serve the WebSocket handshake request.
        if (req.getUri().equals(WEBSOCKET_PATH) &&
            Values.UPGRADE.equalsIgnoreCase(req.getHeader(CONNECTION)) &&
            WEBSOCKET.equalsIgnoreCase(req.getHeader(Names.UPGRADE))) {

            // Create the WebSocket handshake response.
            HttpResponse res = new DefaultHttpResponse(
                HTTP_1_1,
                new HttpResponseStatus(101, "Web Socket Protocol Handshake"));
            res.addHeader(Names.UPGRADE, WEBSOCKET);
            res.addHeader(CONNECTION, Values.UPGRADE);
            res.addHeader(WEBSOCKET_ORIGIN, req.getHeader(ORIGIN));
            res.addHeader(WEBSOCKET_LOCATION, getWebSocketLocation(req));
            String protocol = req.getHeader(WEBSOCKET_PROTOCOL);
            if (protocol != null) {
                res.addHeader(WEBSOCKET_PROTOCOL, protocol);
            }

            // Upgrade the connection and send the handshake response.
            ChannelPipeline p = ctx.getChannel().getPipeline();
            p.remove("aggregator");
            p.replace("decoder", "wsdecoder", new WebSocketFrameDecoder());

            ctx.getChannel().write(res);

            p.replace("encoder", "wsencoder", new WebSocketFrameEncoder());
            return;
        }

        // Send an error page otherwise.
        sendHttpResponse(
            ctx, req, new DefaultHttpResponse(HTTP_1_1, FORBIDDEN));
    }

    /*
    * This is the meat of the server event processing
    */
}

```

```

* Here we square the number and return it
*/
private void handleWebSocketFrame(ChannelHandlerContext ctx, WebSocketFrame frame) {
    System.out.println(frame.getTextData());
    int number = 0;
    try {
        number = Integer.parseInt(frame.getTextData());

    } catch (Exception e){
        System.out.println("Invalid Number: Parsing Exception.");
    }
    //Square the number
    int result = number * number;

    // Send the square of the number back.
    ctx.getChannel().write(
        new DefaultWebSocketFrame(result+""));
}

private void sendHttpResponse(ChannelHandlerContext ctx, HttpRequest req, HttpResponse res)
{
    // Generate an error page if response status code is not OK (200).
    if (res.getStatus().getStatusCode() != 200) {
        res.setContent(
            ChannelBuffers.copiedBuffer(
                res.getStatus().toString(), CharsetUtil.UTF_8));
        setContentLength(res, res.getContent().readableBytes());
    }

    // Send the response and close the connection if necessary.
    ChannelFuture f = ctx.getChannel().write(res);
    if (!isKeepAlive(req) || res.getStatus().getStatusCode() != 200) {
        f.addListener(ChannelFutureListener.CLOSE);
    }
}

@Override
public void exceptionCaught(ChannelHandlerContext ctx, ExceptionEvent e)
    throws Exception {
    e.getCause().printStackTrace();
    e.getChannel().close();
}

private String getWebSocketLocation(HttpRequest req) {
    return "ws://" + req.getHeader(HttpHeaders.Names.HOST) + WEBSOCKET_PATH;
}
}

```

While we will not make an attempt to explain the code in this class the following code is the meat of this class i.e. the core business functionality of the server which is multiplication of integer numbers.

```

private void handleWebSocketFrame(ChannelHandlerContext ctx, WebSocketFrame frame) {
    System.out.println(frame.getTextData());
    int number = 0;

```

```

try {
    number = Integer.parseInt(frame.getTextData());

} catch (Exception e){
    System.out.println("Invalid Number: Parsing Exception.");
}
//Square the number
int result = number * number;

// Send the square of the number back.
ctx.getChannel().write(
    new DefaultWebSocketFrame(result+""));
}

```

This method handles the web socket frame. The number sent by the client is retrieved through 'frame.getTextData()' it parsed and squared and then sent back to the client using 'ctx.getChannel().write(new DefaultWebSocketFrame(result + ""));'

WebSocketServerPipelineFactory.java Code

```

package com.ayna;

import static org.jboss.netty.channel.Channels.*;

import org.jboss.netty.channel.ChannelPipeline;
import org.jboss.netty.channel.ChannelPipelineFactory;
import org.jboss.netty.handler.codec.http.HttpChunkAggregator;
import org.jboss.netty.handler.codec.http.HttpRequestDecoder;
import org.jboss.netty.handler.codec.http.HttpResponseEncoder;

public class WebSocketServerPipelineFactory implements ChannelPipelineFactory {
    public ChannelPipeline getPipeline() throws Exception {
        // Create a default pipeline implementation.
        ChannelPipeline pipeline = pipeline();
        pipeline.addLast("decoder", new HttpRequestDecoder());
        pipeline.addLast("aggregator", new HttpChunkAggregator(65536));
        pipeline.addLast("encoder", new HttpResponseEncoder());
        pipeline.addLast("handler", new WebSocketServerHandler());
        return pipeline;
    }
}

```

We will not attempt to explain this class but rather point you to the Netty documentation at <http://www.jboss.org/netty/documentation.html> for self reading.

Let's build the client side html page.

index.html Code

```

<!DOCTYPE html>
<!-- Tested In Chrome. -->
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

```

```

<title>WebSocket: Squaring Service</title>
<script type="text/javascript" charset="utf-8" src="js/jquery-1.3.2.js"></script>

<script type="text/javascript">
  var ws;

  $(document).ready(function () {
    ws = new WebSocket("ws://localhost:9000/websocket");
    ws.onopen = function(event) { $('#status').text("The WebSocket Connection Is Open."); }
    ws.onmessage = function(event) { $('#result').text("Result= "+event.data); }
    ws.onclose = function(event) { $('#status').text("The WebSocket Connection
Has Been Closed."); }

  });

  function sendNumber(){
    var number = $('#number').val();
    ws.send(number);
  }

</script>
</head>
<body>
  <h1>WebSocket: Squaring Service</h1>
  <div id="status"></div><br/>
  Enter The Number To Square <input id="number" value="10"/><input type="button"
value="Square It..." onclick="sendNumber();" /><br/>
  <div id="result"></div><br/>
</body>
</html>

```

```
<div id="status"></div><br/>
```

The above div will report the status of the connection.

```
Enter The Number To Square <input id="number" value="10"/><input type="button" value="Square It..."
onclick="sendNumber();" /><br/>
```

The above code renders the label 'Enter The Number To Square' followed by an input text box with the id 'number' and then a button with the label 'Square It...' which on being clicked will call the 'sendNumber' method.

```
<div id="result"></div><br/>
```

The above div will be used to display the result returned from the WebSocket server. It will be the square of the number entered by the user.

```
<script type="text/javascript" charset="utf-8" src="js/jquery-1.3.2.js"></script>
```

This page uses the JQuery javascript library.

```
var ws;
```

the 'ws' variable will be used to store a reference to the WebSocket.

```
$(document).ready(function () {  
    ws = new WebSocket("ws://localhost:9000/websocket");  
    ws.onopen = function(event) { $('#status').text("The WebSocket Connection Is Open."); }  
    ws.onmessage = function(event) { $('#result').text("Result= "+event.data); }  
    ws.onclose = function(event) { $('#status').text("The WebSocket Connection Has Been Closed."); }  
});
```

This statement calls the inline function when the document is ready. The inline function opens a new WebSocket to the 'ws://localhost:9000/websocket' url. Then it hooks 3 functions one each to the 'open', 'message' and 'close' events. The 'open' and 'close' event handlers are called when the WebSocket connection is opened and closed respectively, and the handlers update the status in the div with the id as 'status'. The 'message' event handler will be called when a message is received from the server and it displays the result in the div with the id 'result'.

```
function sendNumber(){  
    var number = $('#number').val();  
    ws.send(number);  
}
```

The above function gets called when the 'Square It...' buttons is clicked by the user after he enters an integer number. This function retrieves the number from the text box with the id 'number' and then calls the websocket 'send' method to send the number to the websocket.

We are all set. Start the WebSocketServer, it will display the following

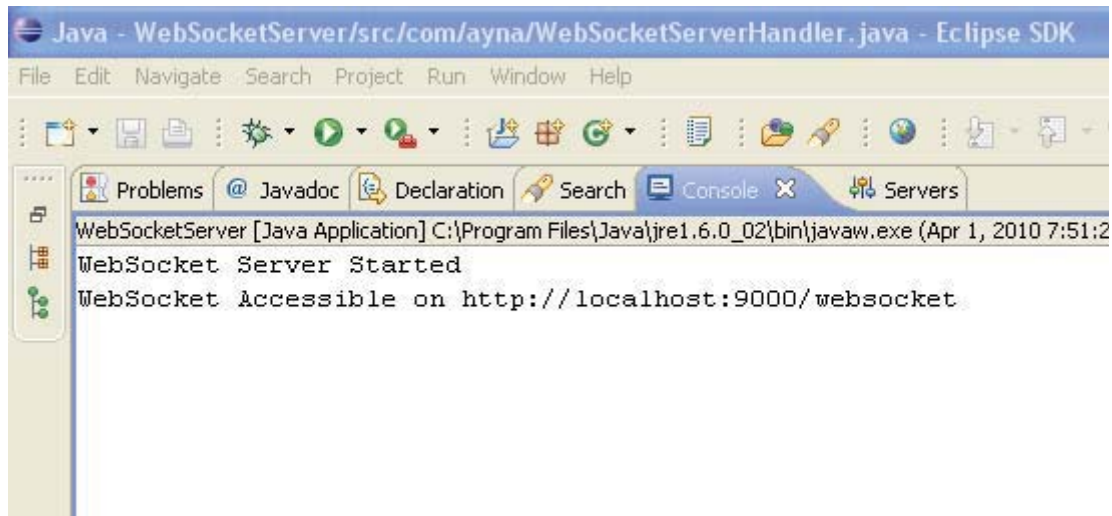
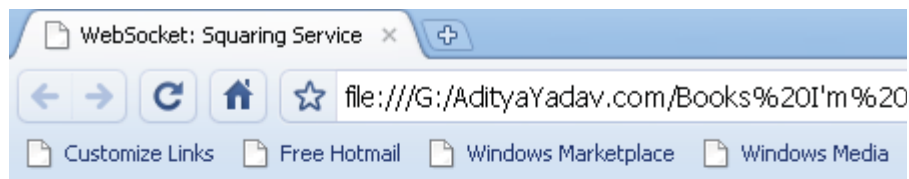


Figure 106 - Netty WebSocket Server Console Output

Double click the index.html file and open it in Chrome browser. Enter an integer number in the text box e.g. 12 and hit the 'Square It...' button. The page will make the call to the server and display the result it receives from the server i.e. 144 as shown below.



WebSocket: Squaring Service

The WebSocket Connection Is Open.

Enter The Number To Square
 Result= 144

Figure 107 - The Netty WebSocket Server Squares 12 and Returns 144

Building a Chat Application over a WebSocket Server using Jetty 7.0.1

Let's create a chat application using Jetty. Create a dynamic web application project in Eclipse and create the following class.

WebSocketChatServlet.java Code

```
package com.ayna;

import java.io.IOException;
import java.util.Set;
import java.util.concurrent.CopyOnWriteArraySet;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.eclipse.jetty.websocket.WebSocket;
import org.eclipse.jetty.websocket.WebSocketConnection;
import org.eclipse.jetty.websocket.WebSocketServlet;

public class WebSocketChatServlet extends WebSocketServlet
{
    private final Set<ChatWebSocket> users = new CopyOnWriteArraySet();

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException ,IOException
    {
        getServletContext().getNamedDispatcher("default").forward(request,response);
    }

    protected WebSocket doWebSocketConnect(HttpServletRequest request, String protocol)
    {
        return new ChatWebSocket();
    }

    class ChatWebSocket implements WebSocket
    {
        WebSocketConnection connection;

        public void onConnect(Outbound connection)
        {
            ChatWebSocket.this.connection= (WebSocketConnection) connection;
            users.add(this);
        }

        public void onMessage(byte frame, byte[] data,int offset, int length)
        {
            // binary communication not needed
        }

        public void onMessage(byte frame, String data)
        {
            for (ChatWebSocket user : users)
            {
                try
                {
                    user.connection.sendMessage(frame,data);
                }
                catch(Exception e) {}
            }
        }

        public void onDisconnect()
        {
            users.remove(this);
        }
    }
}
```

```
}
}
}
```

The `WebSocketServlet` is the base class which provides the `WebSocket` feature. The `doGet` method responds to a HTTP GET request which is when the `WebSocket` handshake happens. It then upgrades the protocol to `WebSocket` protocol this is when the `doWebSocketConnect` happens. The `ChatWebSocket` class is the heart of the businesslogic and it is also used to store a reference to the connection. A 'users' stores all the `WebSockets`. The `WebSocket` supports two kinds of content Text and Binary. For our purpose we are not using any binary payload so the method

```
public void onMessage(byte frame, byte[] data, int offset, int length)
```

is empty. The following method gets called when a Text message is received from Any chat user.

```
public void onMessage(byte frame, String data)
```

The method then loops over all the `WebSockets` and broadcasts the incoming message to all of them hence creating a chat like application.

The `web.xml` file is as follows.

web.xml content

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.4"
xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <display-name>ChatWebSocketServer</display-name>

  <servlet>
    <servlet-name>WebSocketChat</servlet-name>
    <servlet-class>com.ayna.WebSocketChatServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
```

```

        <servlet-name>WebSocketChat</servlet-name>
<url-pattern>/wsc/*</url-pattern>
</servlet-mapping>

<welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.htm</welcome-file>
    <welcome-file>default.jsp</welcome-file>
</welcome-file-list>
</web-app>

```

The above web.xml file configures the WebSocket Servlet. Let's create the chat front end html file as follows.

index.html Code

```

<!DOCTYPE html>
<!-- Tested In Chrome. -->
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>WebSocket: Chat </title>
<script type="text/javascript" charset="utf-8" src="js/jquery-1.3.2.js"></script>

<script type="text/javascript">
    var ws;

    $(document).ready(function () {
        var loc = window.location;
        var host = loc.host;
        ws = new WebSocket("ws://" + host + "/ChatWebSocketServer/wsc/anything");
        ws.onopen = function(event) { $('#status').text("The Chat Connection Is Open."); }
        ws.onmessage = function(event) {
            var $textarea = $('#messages');
            $textarea.val( $textarea.val() + event.data + "\n");
            $textarea.animate({ scrollTop: $textarea.height() }, 1000);
        }
        ws.onclose = function(event) { $('#status').text("The Chat Connection Has Been Closed."); }
    });

    function sendMessage(){
        var message = $('#username').val()+"-"+$('#message').val();
        ws.send(message);
        $('#message').val("");
    }
</script>
</head>
<body>
<h1>WebSocket: Chat</h1>

```

```

<div id="status"></div><br/>
Username <input id="username" value="anonymous"/><br/>
<textarea id="messages" rows="20" cols="60" readonly="readonly" ></textarea><br/>
<input id="message" type="text"/><input type="button" value="Send..."
onclick="sendMessage();"/><br/>

</body>
</html>

```

We are using the JQuery Javascript library in this application.

```
var ws;
```

This variable is used to store the reference to the WebSocket.

```

$(document).ready(function () {
    var loc = window.location;
    var host = loc.host;
    ws = new WebSocket("ws://" + host + "/ChatWebSocketServer/wsc/anything");
    ws.onopen = function(event) { $('#status').text("The Chat Connection Is Open."); }
    ws.onmessage = function(event) {
        var $textarea = $('#messages');
        $textarea.val( $textarea.val() + event.data + "\n");
        $textarea.animate({ scrollTop: $textarea.height() }, 1000);
    }
    ws.onclose = function(event) { $('#status').text("The Chat Connection Has Been Closed."); }
});

```

The above code links the inline function to the document ready event. The inline function retrieves the 'host' string from the url which in our case should be 'localhost:9000' as we have configured the Jetty server to start on port 9000. It then opens a WebSocket to 'ws://localhost:9000/ChatWebSocketServer/wsc/anything' ChatWebSocketServer is the context path of the web application and the Chat WebSocket Servlet is mapped to /wsc/* and hence the /wsc/anything part of the url.

The 'open' and 'close' event handlers are used to update the connection status in the status div.

The onmessage event handler adds the chat message received to the textarea and scrolls the textarea down to the maximum.

```

function sendMessage(){
    var message = $('#username').val()+":"+$('#message').val();
    ws.send(message);
}

```

```
    $('#message').val("");  
}
```

The above function gets called when the 'Send...' button is clicked and it sends the chat message to the Chat WebSocket Server. The message is a text message of the following format '<username>:<message>' where the username is retrieved from the username text box which the user can change at anytime.

```
<div id="status"></div><br/>
```

The above div is used to display the connection status.

```
Username <input id="username" value="anonymous"/><br/>
```

The above line renders the 'Username' label and a textbox with the id 'username' which can be used by the chat user to enter his chat alias. There is no validation or authentication in the system. And there is only one chat room i.e. if you are connected to the server you are in the chatroom.

```
<input id="message" type="text"/><input type="button" value="Send..." onclick="sendMessage();"/><br/>
```

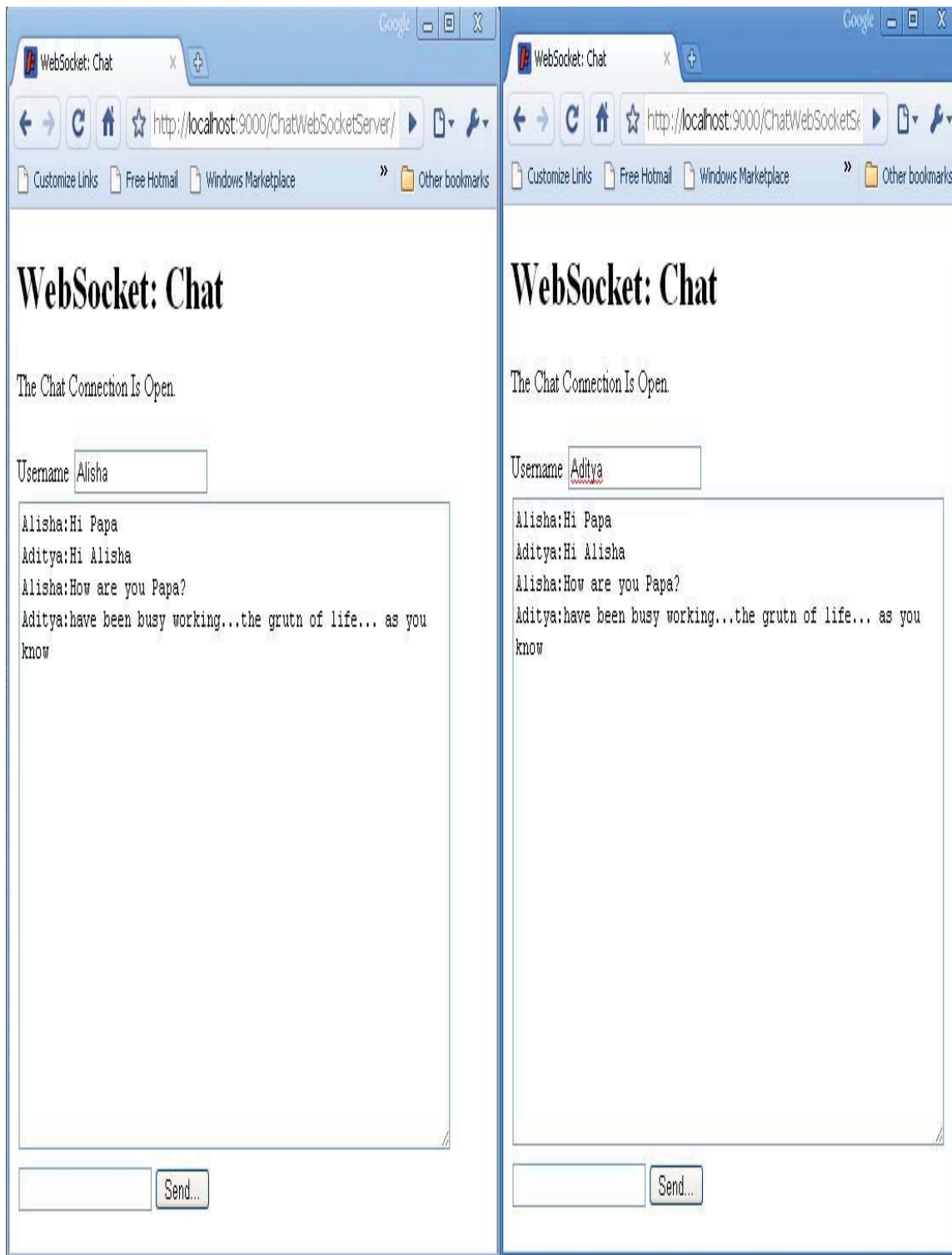
The above line renders a text box with the id 'message' which the user can use to enter his chat message into. It also renders a button with the label 'Send...' which when clicked calls the sendMessage function which will send the message to the server.

Let's take it for a spin. Deploy the web application into the webapps folder of the jetty installation. The source code bundle contains a preconfigured Jetty 7.0.1 server with the web application pre-installed (which is the minimum version of Jetty which supports the WebSocket feature)

If JDK 1.6 is installed on your system and the JAVA_HOME and PATH configured properly you can double click start.jar in the jetty folder to run the server. Or you can type 'java -jar start.jar' from the command line in the Jetty folder.

Launch two browser tabs and open the following URL in both <http://localhost:9000/ChatWebSocketServer/>

Enter different chat aliases in both the tabs and wait for the connection to establish. And start chatting. See below.



Let's try the Chat WebSocket Application in other browsers.

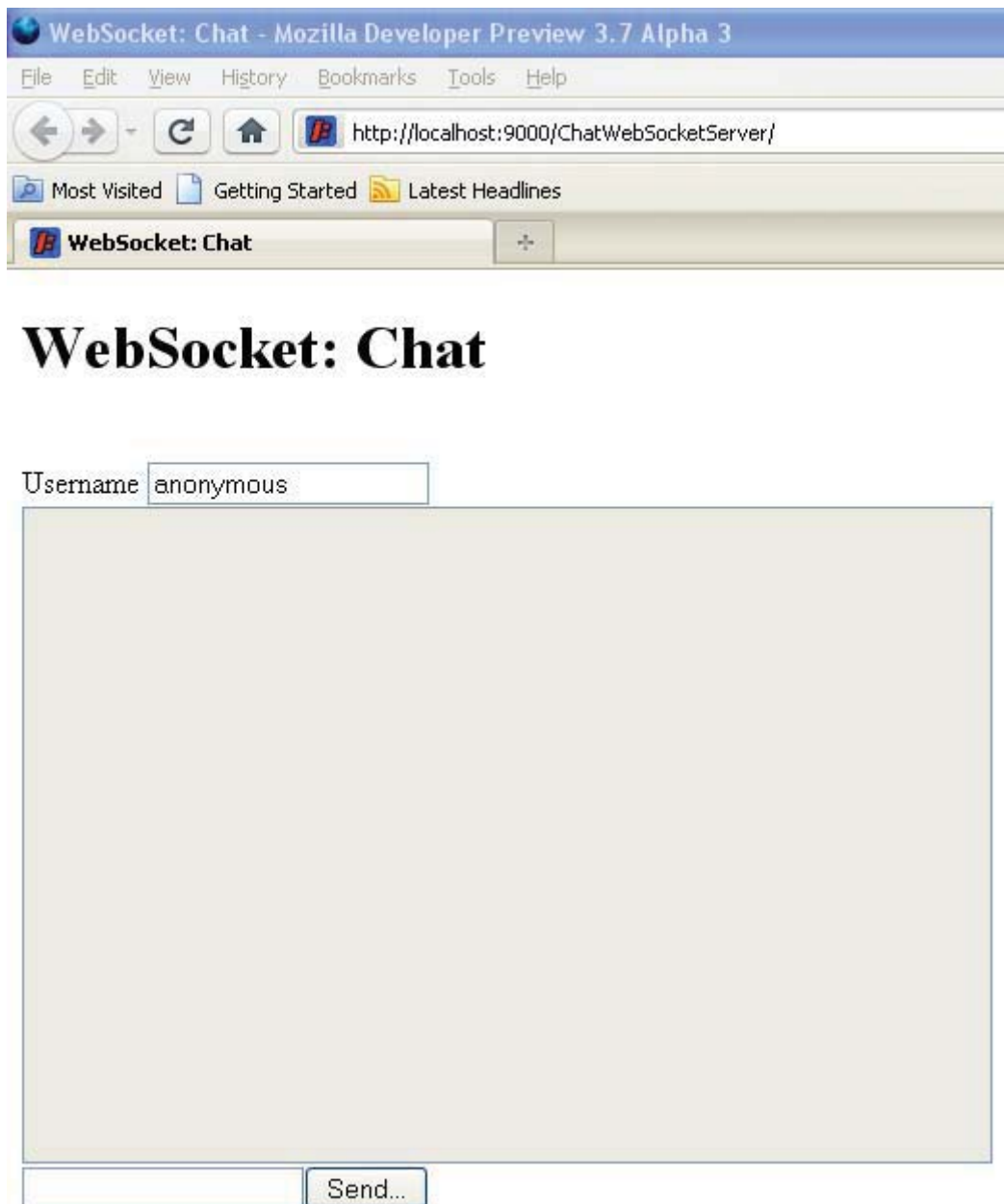


Figure 108 - WebSocket Chat Doesn't Work/Connect in Firefox

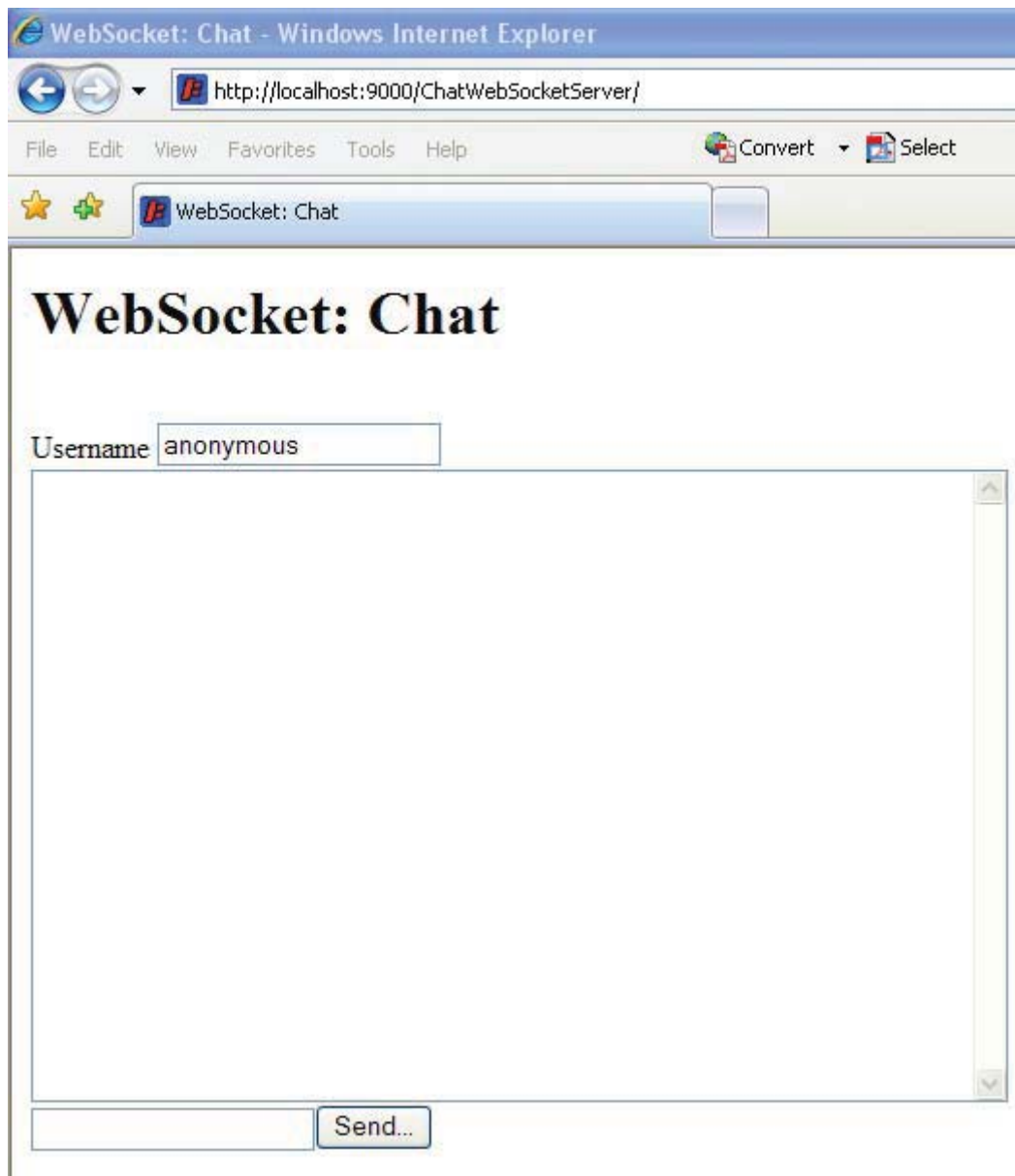


Figure 109 - WebSocket Chat Doesn't Work/Connect in IE

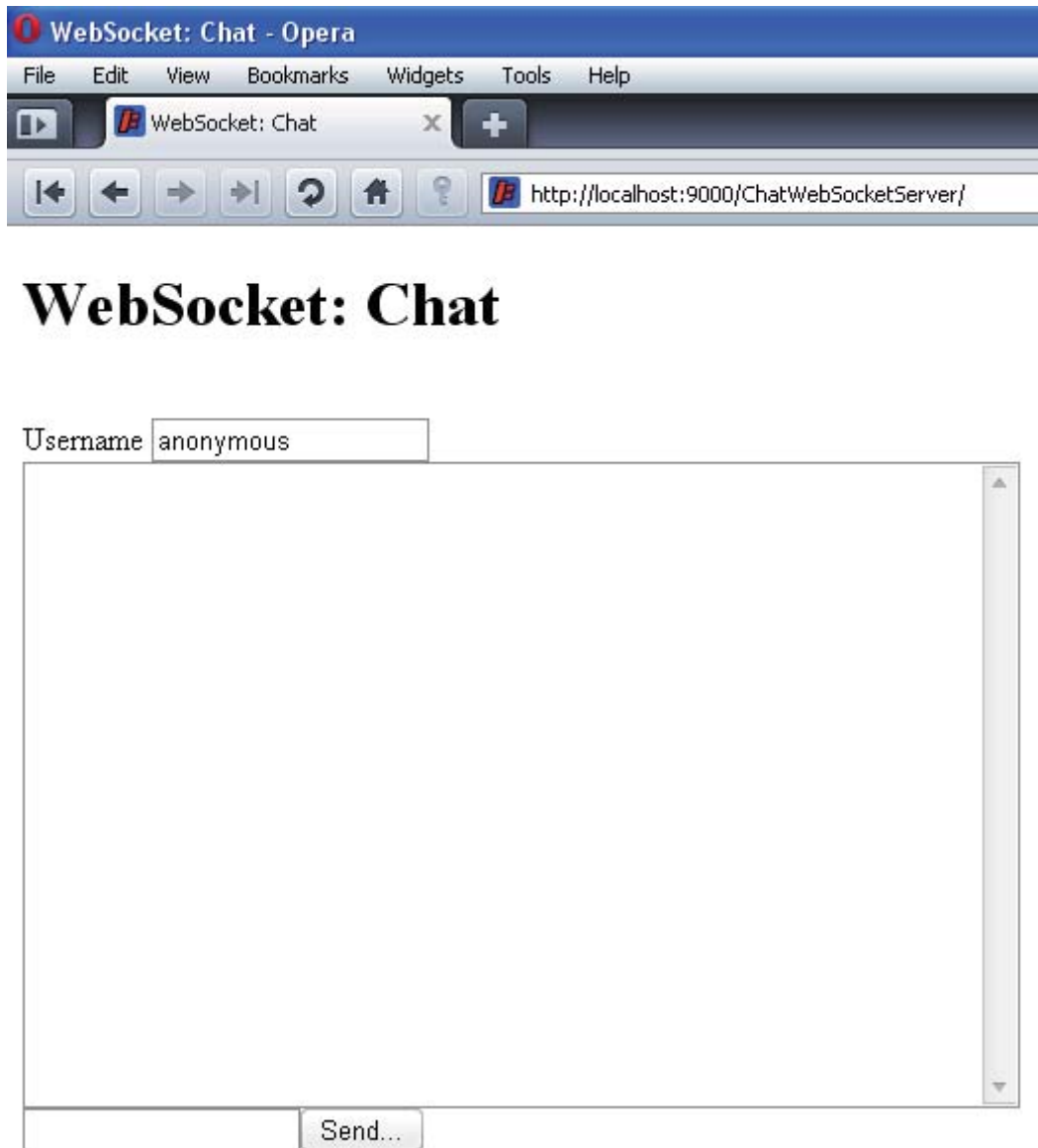


Figure 110 - WebSocket Chat Doesn't Work/Connect in Opera

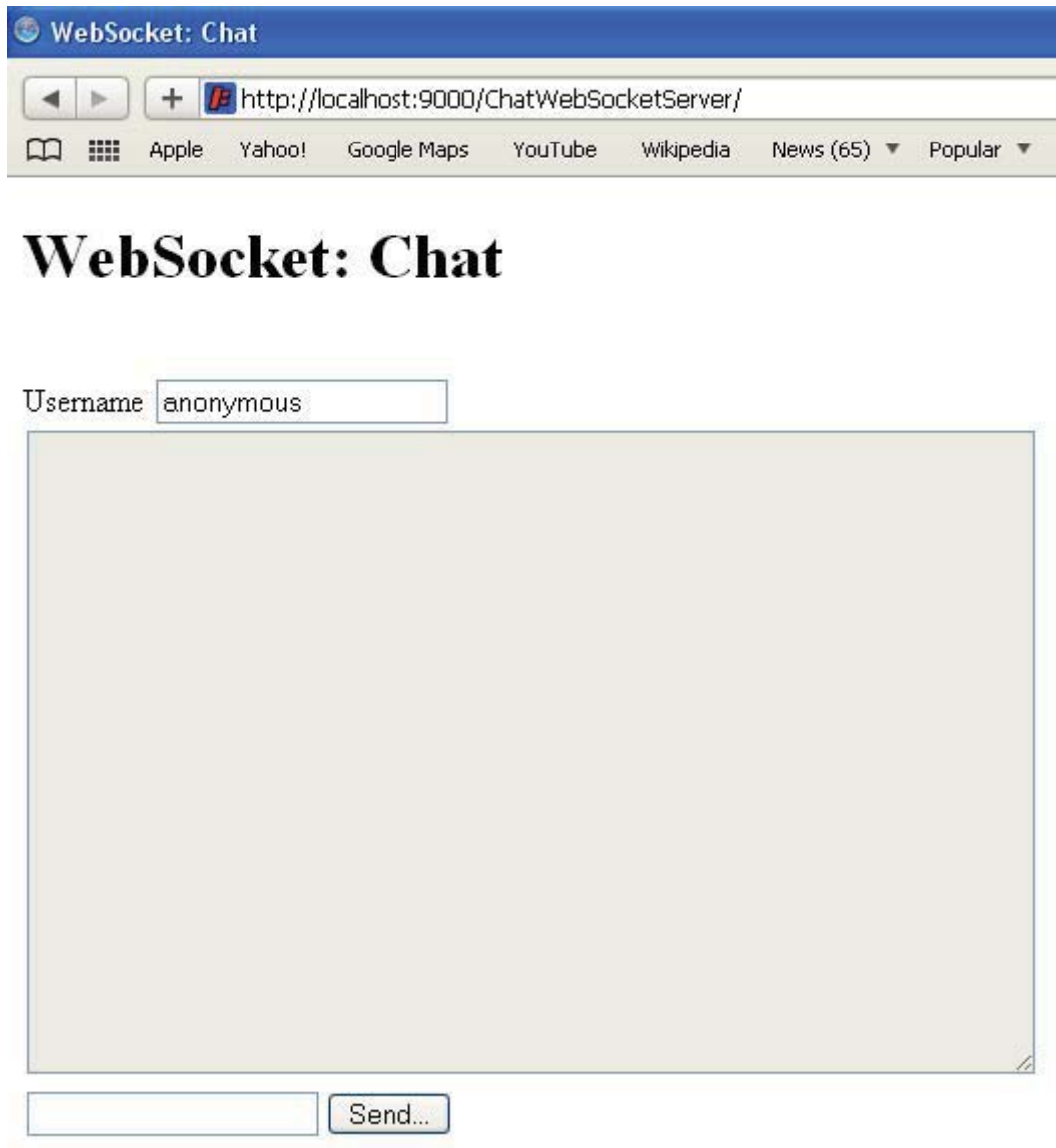


Figure 111 - WebSocket Chat Doesn't Work/Connect in Safari

A Quick Walkthrough of the Specifications

The web sockets specification is very short and simple. Web Sockets is a way of establishing a duplex communication channel with a backend server. It is different from earlier adhoc non-standard methods used by developers like Reverse Ajax, Comet and Pushlets. Which were limited by the 2 connection browser limit. Which doesn't apply to WebSockets or Server Sent Events for that matter. But on the other hand WebSocket based applications are prone to too many connections overwhelming the server.

Security over internet is provided by the application of same-origin policy between the web page and the WebSocket server.

WebSocket protocol starts with an HTTP request which then upgrades to the WebSocket protocol. The protocol is very simple and allows UTF-8 characters in text messages followed by a delimiter. And for binary messages the length is specified. Unlike HTTP it is not a request-response protocol and once the connection is established both sides can send as many messages as and when they need to send them as long as the connection is open.

There are 5 things to be done.

```
var ws = new WebSocket("ws://www.mydomain.com/endpointpath")
```

We open a new WebSocket connection to the backend server. And then we hook onto the 'open', 'message', 'error' and 'close' events.

```
ws.onopen = function(event) { alert("We are connected to the backend server"); }
```

```
ws.onmessage = function(event) { alert("Recieved: " + event.data); }
```

```
ws.onerror = function(event){ alert("error encountered"); }
```

```
ws.onclose = function(event) { alert("Connection to the backend server closed"); }
```

The open event is fired when we get connected to the backend server. The read event is fired when something is received from the backend server. And the close event is fired when the server connection is closed/lost.

```
ws.send("string message payload")
```

The send method sends a message to the server. That's all there is to it apart from the details.

The WebSocket constructor can take a protocol parameter apart from the URL 'new WebSocket(url, protocol)' The protocol parameter is a sub-protocol that the server must support for the connection to be successful. The constructor is a non-blocking call. The URL will start with 'ws://' or 'wss://', the latter is for the secure protocol.

The 'URL' attribute of a WebSocket returns the resolved URL which may or may not be the same as the URL passed in the constructor.

The 'readyState' attribute can be

- CONNECTING (numeric value 0) - The connection has not yet been established.
- OPEN (numeric value 1) - The WebSocket connection is established and communication is possible.
- CLOSING (numeric value 2) - The connection is going through the closing handshake.
- CLOSED (numeric value 3) - The connection has been closed or could not be opened.

The 'send(data)' method sends the data to the server if the state is OPEN. Else it throws an error.

The 'close' method of the websocket closes the connection. The 'bufferedAmount' attribute returns the number of bytes that have been queued but not yet sent to the server.

Retrospective

Instead of explaining WebSockets with proprietary server side software we showed how to create a simple multiplication service with a two way channel using Netty. We also created a simple chat application using the implementation of WebSockets in Jetty 7.0.1. We leave it to the readers to explore other proprietary servers available and also to explore the same origin constraints applicable in their use which will be required when creating an internet facing application.



“This book is a Classic Collector’s item, easy read, short and simple, the first book for everyone to read when learning HTML 5. And in fact the first book in the world available on this topic. We had been struggling with HTML 5 for months and then suddenly our team used this book over the weekend and got started with designing and coding the next Monday. The one book every engineer should have” - Chetan Kumar, VP. Engg, Mobiporter

Aditya has been involved with Software Engineering before the time Internet came to India in 1992-1993. During his more than a decade of industry stint he has played the roles of developer, analyst, product manager through CTO. He is the founder of one of the top 25 startups in India. Before he took his sabbatical to write this book he was a Sr. Architect at Thoughtworks. And is a well known Agile coach. Leads an independent consulting company Aditya Yadav & Associates which offers Technology and Technology Strategy consulting services to Fortune companies. He is a contrarian when it comes to SOA & EAI on which he believes there is no need for Governance or COE's and has proven his approach over the years successfully. Aditya is a Cloud Computing evangelist and has been following this space since Amazon launched AWS. He is technology, platform and domain agnostic and specializes in Internet Scale Architectures and organizational architecture.

When he is not doing something related to software development. He is busy composing music, learning guitar and swimming. Aditya has been a long distance 24 hour swimmer since his college days. He has a B.Tech and M.Tech degree from Indian Institute Of Technology, Bombay, India.

The author can be reached via <http://adityayadav.com>