# Developer Guide

**JWebSocket**

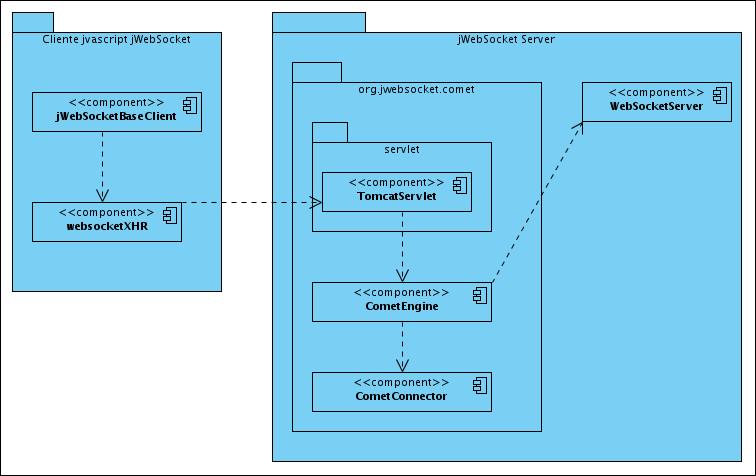
**jWebSocket Long-Polling fallback.**

# Overview

Currently if the users have older browsers and no flash support their client application cannot connect to the jWebSocket server. The goal of this project is resolve this problem using XHR/long-polling fallback as transport when we do not have websocket or flash support.

1. **Infrastructure, Model, Approach**

The solution has two mains packages that are integrated in to the client and server of jWebSocket framework. The package that is embed into jWebSocket client is called websocketXHR and it is a websocket implementation using XHR-Long-Polling as transport. In the server side the package has the org.jwebsocket.comet namespace and inside it are the CometEngine,CometConnector and the TomcatServlet to handler the XHR/AJAX request from the client. In the diagram below this approach is showed.

******

The servlet uses to receive the request from the client has a great importance in the development of the solution. This servlet implement the org.apache.catalina.comet.CometProcessor interface provided by Apache Tomcat to handler advance XHR/AJAX request, where the life cycle of each request passes through a series of events that give facilities to implement the long-polling technique. Using the event we can receive, read, write, hold and close each request from the client side, to get more information about CometProcessor please refers to http://tomcat.apache.org/tomcat-7.0-doc/aio.html. CometProcessor is implemented based on NIO, since this provides a great number of features to handle input and output asynchronously requests that are not supported by the standard used by Tomcat connector (HTTP / AJP). The first prerequisite to run this solution must be replace in the Tomcat configuration file the HTTP / AJP Connector set by default with the NIO connector. This could be done in the server.xml file shown in the example below.

<!-- Connector port="8080" protocol="HTTP/1.1"

connectionTimeout="20000"

URIEncoding="UTF-8"

redirectPort="8443" / -->

<Connector connectionTimeout="20000" port="8080"

protocol="org.apache.coyote.http11.Http11NioProtocol"

redirectPort="8443"/>

The above configuration is the only one that is mandatory but is still recommended increase the minimum size of RAM memory available for Tomcat altering the min values ​​for catalina.Xms(Minimum memory available to tomcat) and Xmx (maximum memory which can have tomcat).

In linux you could do this as follows.

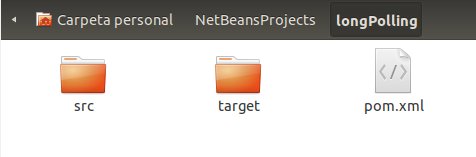
export CATALINA\_OPTS="-Xms512m -Xmx1024m"

In Windows:

set CATALINA\_OPTS=%CATALINA\_OPTS% -Xms512m -Xmx1024

You can get the same result by increasing the maximum memory size that can use the Java virtual machine, but in our case is much more advisable just increase the memory available for Tomcat.

# Modules, Structures

****

**src:**  All java.classes uses in the solution and a sample application. There is also a folder named "webapp" that contains a sample web application that uses the library XHRWebsocket to test the long-polling solution.

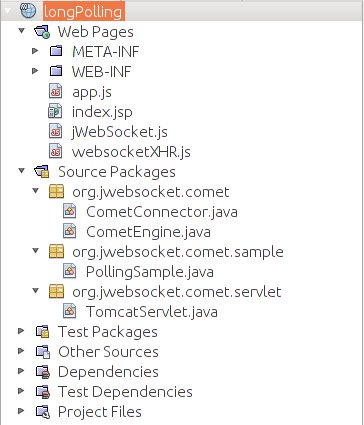
**target:** This directory stores the compiled source, also includes longPolling file-1.0.war with which you can deploy the sample application quickly and easily in tomcat server.

**pom.xml:** Configuration file which specifies the characteristics and dependencies of the project maven.

**General Units:**

org.jwebsocket.comet package depends of several jWebSocket libraries, therefore we need to add org.jwebsocket dependencies. The other dependency is about CometProcessor that is inside org.apache.tomcat.embed package. Those Dependencies can be imported using maven as presented below.

1. </dependency>
2. <dependency>
3. <groupId>org.jwebsocket</groupId>
4. <artifactId>jWebSocketServer</artifactId>
5. <version>1.0</version>
6. </dependency>
7. <dependency>
8. <groupId>org.apache.tomcat.embed</groupId>
9. <artifactId>tomcat-embed-core</artifactId>
10. <version>7.0.21</version>
11. </dependency>
12. </dependencies>



# Source Code

The solution integrates components on the client and server of jWebSocket framework. In the server side all the solution is embed in org.jwebsocket.comet package. In the client side the solution is just the XHRWebSoscket component.

|  |  |
| --- | --- |
| **Package: org.jweboscket.comet** | |
| **Component** | **org.jweboscket.comet.CometEngine** |
| **Description** | It is an implementation of jWebSocket Engine. This class extends of base Engine, and overwrites some functionality to adjust to the long-polling solution. |
| **Class** | CometEngine |
| **Class CometEngine** | |
| **Extends** | BaseEngine |
| **Dependencies** | import java.util.Map;  import java.util.Queue;  import java.util.concurrent.ConcurrentLinkedQueue;  import javolution.util.FastMap;  import org.jwebsocket.api.EngineConfiguration;  import org.jwebsocket.api.WebSocketConnector;  import org.jwebsocket.comet.servlet.TomcatServlet;  import org.jwebsocket.engines.BaseEngine;  import org.jwebsocket.kit.CloseReason;  import org.jwebsocket.kit.WebSocketException;  import org.apache.log4j.Logger;  import org.jwebsocket.api.WebSocketPacket;  import org.jwebsocket.logging.Logging; |
| **Package: org.jweboscket.comet** | |
| **Component** | **org.jweboscket.comet.CometConnector** |
| **Description** | It is the connector to uses with this solution. |
| **Class** | CometConnector |
| **Class CometConnector** | |
| **Extends** | BaseConnector |
| **Dependencies** | import java.io.IOException;  import java.io.PrintWriter;  import java.net.InetAddress;  import java.net.UnknownHostException;  import java.util.Map;  import javolution.util.FastMap;  import org.apache.catalina.comet.CometEvent;  import org.jwebsocket.api.WebSocketEngine;  import org.jwebsocket.connectors.BaseConnector;  import org.jwebsocket.kit.CloseReason;  import org.apache.log4j.Logger;  import org.jwebsocket.api.WebSocketPacket;  import org.jwebsocket.comet.servlet.TomcatServlet;  import org.jwebsocket.kit.RawPacket;  import org.jwebsocket.logging.Logging;  import org.jwebsocket.packetProcessors.JSONProcessor; |
| **Package: org.jweboscket.comet.servlet** | |
| **Component** | **org.jweboscket.comet.servlet.****TomcatServlet** |
| **Description** | This component is used to handle incoming HTTP requests, to implement the long-polling technique, and the communication with the CometEngine component. |
| **Class** | TomcatServlet |
| **Class** TomcatServlet | |
| **Extends** | **HttpServlet** |
| **Implements** | **CometProcessor** |
| **Dependencies** | **java.io.IOException;**  **java.io.InputStream;**  **java.io.PrintWriter;**  **java.util.Map;**  **javax.servlet.ServletException;**  **javax.servlet.http.HttpServlet;**  **javolution.util.FastMap;**  **org.apache.catalina.comet.CometEvent;**  **org.apache.catalina.comet.CometEvent.EventType;**  **org.apache.catalina.comet.CometProcessor;**  **org.jwebsocket.comet.CometConnector;**  **org.jwebsocket.comet.CometEngine;**  **org.jwebsocket.factory.JWebSocketFactory;**  **org.jwebsocket.kit.RawPacket;**  **org.apache.log4j.Logger;**  **org.json.JSONObject;**  **org.jwebsocket.api.WebSocketConnector;**  **org.jwebsocket.kit.CloseReason;**  **org.jwebsocket.kit.RequestHeader;**  **org.jwebsocket.logging.Logging;**  **org.jwebsocket.packetProcessors.JSONProcessor;** |

**Client:** The XHRWebSocket component is embedded within the native jWebSocket JavaScript client and aims to be used automatically in the absence of WebSocket or flash suport.

|  |  |
| --- | --- |
| **Package: XHRWebSocket** | |
| **Component** | **XHRWebSocket** |
| **Description** | XHRWebSocket is an object that emulates the usual behavior of the object WebSocket. It uses long-polling to emulate a keep alive connection with several request, send and receive data at the same time. |

**Libraries and tools used:**

* Tomcat CometProccesor.
* Servidor Apache Tomcat 7,0
* NetBeans 7.0.1
* Maven
* jWebSocket framework

**Highlights for developers:**

To implement the XHRWebSocket component was necessary think in how to deal with the two HTTP connection limit issue. The limit of only two connections per server means that if you have two connections open and starts a third or several N requests, the N requests will be locked until one of the two previous requests finish. In this case to simulate the keep-alive connection to the server and receive instant messages in the moment that are available on the server side, the XHRWebSocket component keeps busy one of the two connections. So in order to send a simple message to the server with the word "Hello", the client must use another HTTP request, even when this second request is not delayed by the server, in the moment in which the server takes the message the two limit connections are busy. If the client tries to send one or more messages at this time all those HTTP requests will be blocked. Once the request used to send the word “Hello” is deliberate, the blocked messages could be send in no particular order and in the worst case the server for security reasons could destroy our session.

To solve the problem mentioned in the previous paragraph was implemented a message queue in the XHRWebSocket component. As mentioned above to simulate the connection and receive messages from the server one request is always busy, so we only have one channel to send. This request that is used to send messages to the server is never delayed by the server. The server receives the request with the message and checks if there is a message available for this client, if so this puts the message in response to the request and immediately release the request to the client, if there is no messages the request is release in the same way, enabling again the connection to send data at any time. But if the user try to send a message when the request used to send message is busy, the message is store in to the queue and always that the request is release by the server in the client side the queue is checked and the pending messages are sending.

The connection used to simulate the keep-alive connection also has the purpose of bringing messages from the server. The component uses long-polling technique to keep one of the two possible open requests to the server. When the server has a message to send to the client it put the message in the request response. Once this response arrived to the client the onMessage event is fired with the message, and the XHRWebsocket component re-launches the request in order to emulate the keep-alive connection gains.

**Details of the comunication:**

The goal here is provide the better undertanding of how the comunication works. To do this I explaing how emulate the connection. How use JSON to create each message to allow the agree between client and server side. What means each word using to create the message, and what ocurre with each message on the client and server side.

As mention in the higthliting for developer above, in this long-polling fallback solution I intend use the two open HTTP connection allowed per server by each browser. One of this HTTP connection always handler the message releat to the connection, and the other is uses to send message with information two the server. I will explain in details how this happens.

**Structure of messages**

Before beginning with the explanation of how the connection works I think thas is a good idea take a look of the message estructure. There are two type of message that allows the comunication. To explain the structure of the messages we will see first a “connection” message and analyze their terms.

Connection message:

{subPl:"json",cometType:"connection",readyState:0}

The **subPl** key is used to expecific which subprotocol use the connection, actualy just JSON is suport, but in the future this could be use to allow the comunication with XML or CSV.

The **cometType** keyis used to especific which kind of message is send. One message it is create with the “connection” type and the other with the “messages” type

cometType:"connection" - use for the connection messages.

cometType:"message" - use for message that send information.

The **readyState** propertie is uses to handle the state of the connection.

ReadyState: 0 – connection is not established

ReadyState: 1 – connection is already open

ReadyState: 3 – connection close

Example of the Message use to send data:

{subPl:"json",cometType:"message",data:"{ns:"jws.comet.sample",

type:"chat","message":"hello","utid":17}","readyState":1}

As can be seen the message above has the same structure of the connection message. The new here is the word “message” as the king of the message and the object **data**.

**Data:** Seethat in **data** is sent the message that would been sent by the normal websocket connection, I mean without use the long-polling fallback solution. The content of **data** is parse on the server side and given to the connector to execute the processPacket method.

The approach of the message above is the same that use the server to send information to the client. Remember the two channels or connection are use by the server to send information to the client as is show in the following messages.

Message receive by the **connection** channel:

{"cometType":"**connection**","data":{"type":"response","code":0,"msg":"ok","utid":22,"ns":"jws.comet.sample","reqType":"chat","data":"hello"},"readyState":1}

Message receive by the **message** channel:

{"cometType":"**message**","data":{"type":"response","code":0,"msg":"ok","utid":22,"ns":"jws.comet.sample","reqType":"chat","data":"hello"},

"readyState":1}

Wherever the channel in which one message is receive the content of **data** is fired into the on\_message event of websocket.

**Comunication step by step.**

Until now we already know what is a message and what kind of message handler this solution. Let see how it work.

When the jWebSocket client execute the open method to connect with the server the first HTTP XML-HttpReques is send with the following message to establish connection.

{"subPl":"json","cometType":"connection","readyState":0}

if the connection is established the server response with a similar message with the readyState in 1. The client responds again with this message

{"subPl":"json","cometType":"connection","readyState":1}

Until the moment that the connection is established, all the request that refers to the connection channel are delayed on the server as sets the long-polling style. At the same moment in which some data is able to the client, the data is put on the response and release to the client. If no data is able on the server side to send to the client, every 21 seconds the timeout event is fired and the server release the request, so the client send again the connection message to the server.