# Manual de desarrollo

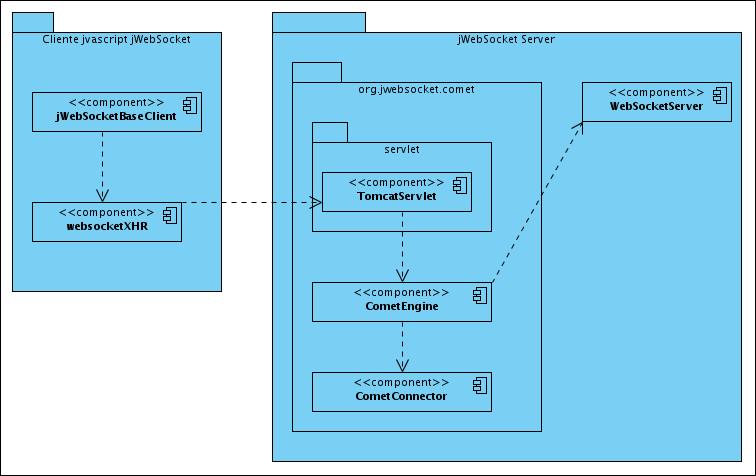
**jWebSocket**

**Mecanismo de respaldo a conexiones WebSocket para el marco de trabajo jWebSocket.**

# Control de versiones

|  |  |  |  |
| --- | --- | --- | --- |
| **Fecha** | **Versión** | **Descripción** | **Autor** |
| 28/03/2012 | 1.0 | Creación del documento | Osvaldo Aguilar Lauzurique |

# Infraestructura de la solución

El mecanismo de respaldo mediante la técnica de long-polling para jWebSocket permite a las aplicaciones desarrolladas con esta tecnología establecer la comunicación y funcionar de manera aceptable incluso cuando el navegador carece de soporte para websocket. La solución se distribuye en dos paquetes fundamentales que se integran al servidor y al cliente del marco de trabajo jWebSocket. **

*Ilustración 1: Diagrama con la estructura de la solución.*

Uno de los aspectos que más influye en el desarrollo de la solución desde el punto de vista arquitectónico y de implementación es el uso de la clase interfaces CometProcessor proporcionada por Apache Tomcat. La interface CometProcessor brinda una forma sutil de manejar las peticiones del cliente mediante una sucesión de eventos en el servidor, con el manejo de estos eventos se puede decidir cuándo recibir, leer, escribir, retener y cerrar una petición proveniente del cliente. CometProcessor está implementado sobre la base de NIO, dado que este brinda un gran número de funcionalidades para manejar la entrada y salida de peticiones de forma asíncrona que no son soportados por el conector estándar utilizado por Tomcat (HTTP/AJP). Por tanto como primer requisito para la solución se debe sustituir en las configuraciones de Tomcat el conector HTTP/AJP establecido por defecto con el conector de NIO. En la carpeta raíz de la instalación de Tomcat en el fichero server.xml se debe sustituir el conector como se muestra en el ejemplo a continuación.

<!-- Conector por defecto que se debe poner en modo comentario o eliminar -->

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

connectionTimeout="20000"

URIEncoding="UTF-8"

redirectPort="8443" /-->

<!-- Conector que permite el uso de NIO -->

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

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

redirectPort="8443"/>

La configuración anterior es la única que se presenta de forma obligatoria aunque es recomendable aumentar el tamaño mínimo de la memoria RAM disponible para Tomcat alterando los valores de las variables de configuracion catalina **Xms (memoria mínima disponible para Tomcat),** y **Xmx (Memoria máxima de la cual Tomcat puede disponer).**

Esto puede ser realizado sobre linux mediante una consola de la siguiente forma.

export CATALINA\_OPTS="-Xms512m -Xmx1024m"

En Windows:

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

Se puede obtener el mismo resultado aumentando el tamaño de memoria máximo que puede usar la máquina virtual de java, pero en nuestro caso es mucho más recomendable solo incrementar la memoria disponible para Tomcat.

# Modularización de la aplicación

La solución integra componentes en el cliente y servidor del framework jWebSocket. En el lado del servidor todo lo que concierne al módulo de respaldo se encuentra dentro del paquete org.jwebsocket.comet mientras en el cliente está compuesto con el componente XHRWebSoscket. A continuación se listaran todos los paquetes con sus componentes.

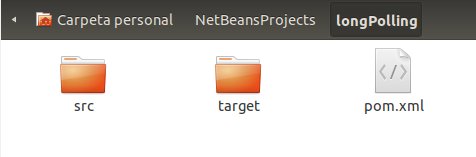
|  |  |
| --- | --- |
| **Paquete: org.jweboscket.comet** | |
| **Componente** | **org.jweboscket.comet.CometEngine** |
| **Descripción** | Contiene la implementación del motor de jWebSocket para comet. Puede ser utilizado como motor para otras soluciones basadas en las técnicas de comet. |
| **Clases que integran el componente.** | CometEngine |
| **Clase CometEngine** | |
| **Extiende** | BaseEngine |
| **Dependencias** | 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; |
| **Paquete: org.jweboscket.comet** | |
| **Componente** | **org.jweboscket.comet.CometConnector** |
| **Descripción** | Actúa como conector para las soluciones basadas en comet. |
| **Clases que integran el componente.** | CometConnector |
| **Clase CometConnector** | |
| **Extiende** | **CometConnector** |
| **Dependencias** | 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; |
| **Paquete: org.jweboscket.comet.servlet** | |
| **Componente** | **org.jweboscket.comet.servlet.****TomcatServlet** |
| **Descripción** | Este componente está diseñado para tratar con las peticiones http entrantes y comunicarse con el componente CometEngine. |
| **Clases que integran el componente.** | TomcatServlet |
| **Clase TomcatServlet** | |
| **Extiende** | **HttpServlet** |
| **Implementa** | **CometProcessor** |
| **Dependencias** | **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;** |

**Cliente:** En cuanto al cliente se refiere el componente XHRWebSocket es embebido dentro del cliente nativo de jWebSocket y su fin es ser utilizado de forma automática ante la carencia de websocket o flash.

|  |  |
| --- | --- |
| **Paquete: XHRWebSocket** | |
| **Componente** | **XHRWebSocket** |
| **Descripción** | En lado del cliente la librería XHRWebSocket es un objeto que emula el comportamiento habitual del objeto websocket. Este utiliza peticiones de tipo AJAX como medio de transporte en sustitución del protocolo websocket. |

# Estructura del código fuente

**Directorio raíz del proyecto**

****

**src:** Dentro de este directorio en la carpeta “java” se encuentran el código de las clases pertenecientes a la solución y un plugin de ejemplo. También se encuentra una carpeta nombrada “webapp”: contiene una aplicación web de ejemplo que se hace uso de la librería XHRWebsocket.

**target:** Este directorio almacena temporalmente el código fuente compilado, también incluye el archivo longPolling-1.0.war con el cual se puede desplegar la aplicación de ejemplo de forma rápida y sencilla en el servidor de tomcat.

**pom.xml:** Archivo de configuración en el cual se especifican las características y dependencias del proyecto maven.

**Dependencias Generales:**

El paquete **org.jwebsocket.comet** depende de varias librerías del framework jWebSocket por tanto presenta dependencias del paquete org.jwebsocket mientras el componente **TomcatServlet** presenta dependencias del paquete **org.apache.tomcat.embed**. Las dependencias pueden ser importadas usando maven como se presentan a continuación.

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>

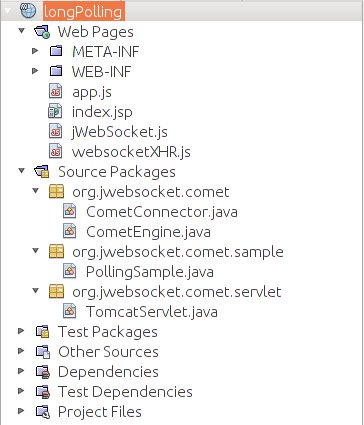


Ilustración 2: Código fuente de la solución en NetBeans IDE.

# Librerías y herramientas utilizadas

* Tomcat CometProccesor.
* Servidor Apache Tomcat 7,0

# Aspectos Relevantes para los desarrolladores.

Para la implementación del componente XHRWebSocket se tuvo en cuenta el límite que por defecto presenta el protocolo HTTP de dos conexiones abiertas permanentemente con el servidor. El límite de solo dos conexiones representa que si se tienen abierta dos conexiones y se inicia una tercera o un numero peticiones quedaran bloqueadas hasta que finalice una de las dos peticiones anteriores. Para simular la conexión con el servidor y recibir los mensajes en el instante en el que estos están disponibles en el servidor El componente XHRWebSocket mantiene ocupada una de las dos conexiones. Para enviar un mensaje al servidor con la palabra "Hello", el cliente debe hacer uso de otra petición HTTP. Por tanto en este instante se tiene una conexión abierta para la conexión y otra para enviar la palabra "Hello", quedando ocupadas las dos conexiones límites. Si el cliente trata de enviar otro o varios mensajes en este momento todas las peticiones HTTP que se necesitarían para enviar los mensajes quedarían bloqueadas. Después de finalizar la conexión que se utilizó para enviar la palabra "Hello", los mensajes bloqueados serian deliberados sin un orden en particular y en el peor de los casos el servidor por motivos de seguridad puede pensar que se trata de un intento de ataque y cerrar la sesión.

Para solucionar el problema de las dos conexiones límite, se implementó una cola de mensajes en el componente XHRWebSocket. Como se dijo anteriormente para simular la conexión y recibir mensajes desde el servidor siempre se mantiene ocupada una petición, por lo cual solo nos queda un canal para enviar. La otra petición que se utiliza para enviar los mensajes al servidor nunca es demorada en el servidor. El servidor recibe la petición con el mensaje y chequea si tiene algún mensaje disponible para el cliente, de ser así este pone el mensaje en la respuesta de la petición y la libera hacia el cliente, en caso de no tener mensajes de igual forma libera la petición. Si se intenta enviar un mensaje y la petición utilizada para enviar no ha sido liberada por el servidor, el mensaje es adicionado a la cola y cuando la petición regresa se chequea la cola y se envían los mensajes almacenados siguiendo el mismo procedimiento.

La conexión utilizada para la conexión también cumple el propósito de traer mensajes desde el servidor. El componente utiliza la técnica de long-polling para mantener una de las dos posibles peticiones abierta con el servidor. Cuando el servidor tiene un mensaje para enviar al cliente lo pone como respuesta en esta conexión que se utiliza para simular la conexión y la finaliza para que regrese al cliente. El cliente recibe el mensaje y vuelve a lanzar la petición para que el servidor la retenga y el proceso sigue sucediendo de esta forma siempre que el servidor necesita enviar un mensaje al cliente.