Example of Creating a WebSocket Server in Java

Posted on **January 28, 2018** by **Bruno**

This post shows how to implement a WebSocket server in Java using the @ServerEndpoint annotation. A WebSocket server application can be deployed to Tomcat 7 or higher, or to any other Java EE servlet container that supports WebSockets. There are two packages for WebSocket programming:

javax.websocket – APIs common to both the client and server side
javax.websocket.server – APIs used only by server side applications

WebSocket is a technology for establishing a persistent, low-latency, full-duplex communication channel over a single http connection for real-time data exchange between a server endpoint (*Java, .NET, PHP etc.*) and a client (*HTML5 / JavaScript, iOS*).

- The WebSocket protocol is an IETF proposed standard as described in RFC6455.
- The WebSocket protocol defines 2 new URI schemes: **`ws`** and **`wss`** for TLS encryption.
- WebSocket server URIs support query parameters as in:
  `ws://hostname:8080/AppContext/endpoint?userId=12345&location=London`
- The Java WebSocket API runs on Servlet containers such as Tomcat, JBoss and Websphere.
- See Oracle's JSR 356 for specification details of the Java API for WebSocket.
- The W3C maintains the WebSocket JavaScript API Specification and defines the WebSocket interface.
- HTML5 compliant web browsers provide an implementation of the specification to enable clients to connect to a WebSocket server and to send and receive data (*IE10+, Chrome 16+, Firefox 11+, Safari 6+*).

For an example of how to create a WebSocket client in JavaScript and HTML 5, see the post below:

Create a WebSocket Client in JavaScript

Summary

CATEGORIE

1 Implementation of WebSocket Server in Java

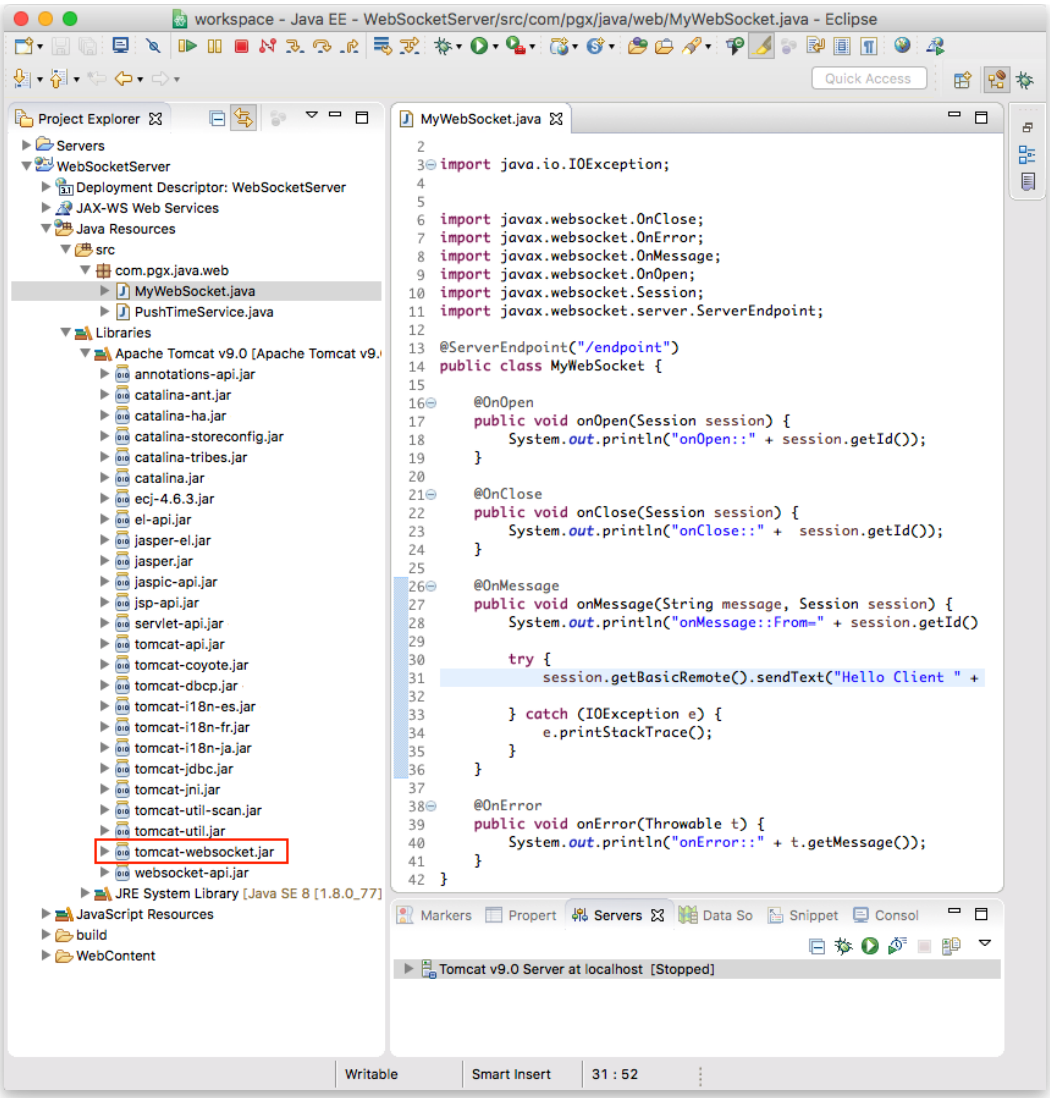
Below is the Java source code for the WebSocket server endpoint implementation. In ***line 10***, the annotation @ServerEndpoint is used to decorate a class that implements a WebSocket server endpoint. Four more method annotations are used to decorate event handlers for WebSocket client connections.

@OnOpen – Called when a client connects
@OnClose – Called when a client connection is closed
@OnMessage – Called when a message is received by the client
@OnError – Called when an error for this endpoint occurred

These four methods are invoked by the container.

```
1. package com.pgx.java.web;
2.
3. import java.io.IOException;
4. import java.util.List;
5. import java.util.Map;
6.
7. import javax.websocket.OnClose;
8. import javax.websocket.OnError;
9. import javax.websocket.OnMessage;
10. import javax.websocket.OnOpen;
11. import javax.websocket.Session;
12. import javax.websocket.server.ServerEndpoint;
13.
14. @ServerEndpoint("/endpoint")
15. public class MyWebSocket {
16.
17.     private static PushTimeService pst;
18.
19.     @OnOpen
20.     public void onOpen(Session session) {
21.         System.out.println("onOpen::" + session.getId());
22.     }
23.
24.     @OnClose
25.     public void onClose(Session session) {
26.         System.out.println("onClose::" +  session.getId());
27.     }
28.
29.     @OnMessage
30.     public void onMessage(String message, Session session) {
31.         System.out.println("onMessage::From=" + session.getId() + " Message=" + message);
32.
33.         try {
34.             session.getBasicRemote().sendText("Hello Client " + session.getId() + "!");
35.
36.         } catch (IOException e) {
37.             e.printStackTrace();
38.         }
39.     }
40.
41.     @OnError
42.     public void onError(Throwable t) {
43.         System.out.println("onError::" + t.getMessage());
44.     }
45. }
```

The ***Eclipse Neon IDE*** for Java web development and ***Apache Tomcat 9*** were used as described in



This setup allows to run the WebSocket server application on Tomcat from within Eclipse. The above image shows the basic structure of the dynamic web project in the Project Explorer. Note the **`tomcat-websocket.jar`** file in the Apache Tomcat v9.0 server runtime library. It contains the Java API for WebSocket support.

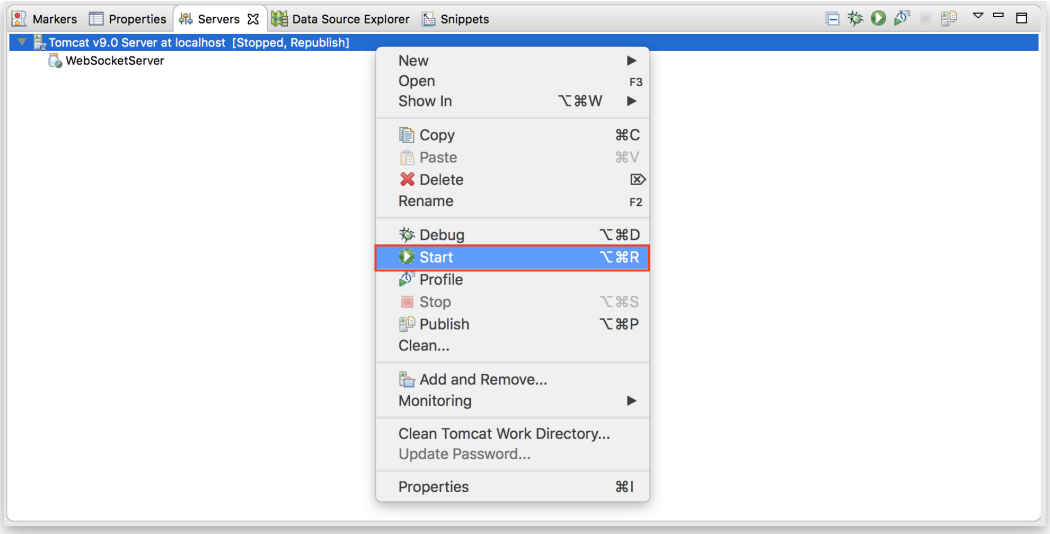
2 Running the WebSocket Server on Tomcat

To export the WebSocket server application so that it can be deployed to an external Tomcat server, see the below post on creating a WAR file:

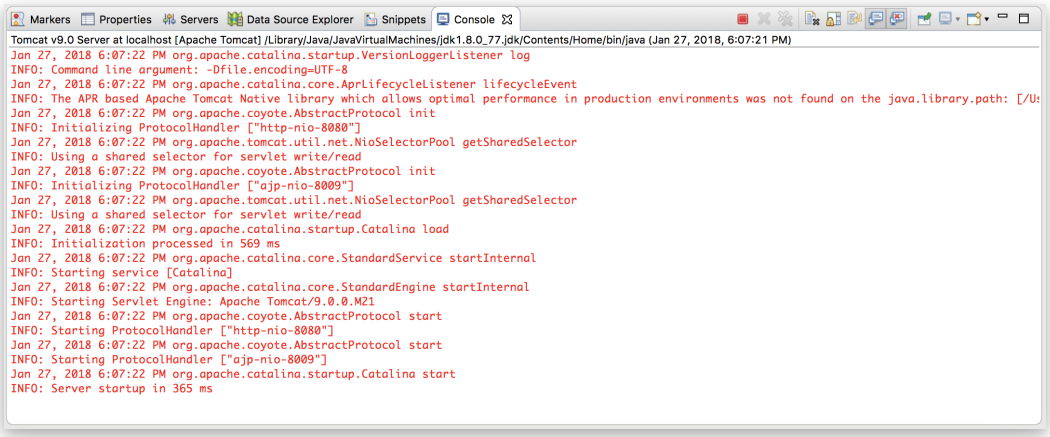
Eclipse Neon – Export Dynamic Web Project to WAR File

Here, the web application is run from within Eclipse using the dynamic web project setup mentioned earlier. No configuration changes to Tomcat are required to run a WebSocket server endpoint. The port number will be the same as the one used for connections via the **`http`** protocol, e.g. **`8080`**.

ARCHIVES
- March 2018 (1)
- February 2018 (2)
- January 2018 (6)
- December 2017 (1)
- November 2017 (2)
- October 2017 (2)
- September 2017 (2)
- June 2017 (2)
- May 2017 (3)
- April 2017 (2)
- February 2017 (1)
- January 2017 (1)
- October 2016 (1)
- September 2016 (1)
- August 2016 (1)
- July 2016 (1)
- June 2016 (1)
- May 2016 (3)
- April 2016 (1)
- February 2016 (2)
- January 2016 (5)
- December 2015 (2)



Use the ***Console*** tab in Eclipse to view the Tomcat server output and to catch potential errors on startup.



If there are no errors, the WebSocket server is up and running and ready to accept connections from clients. The WebSocket server endpoint URL for accessing it from a local machine would be:

```
ws://127.0.0.1:8080/WebSocketServer/endpoint
```

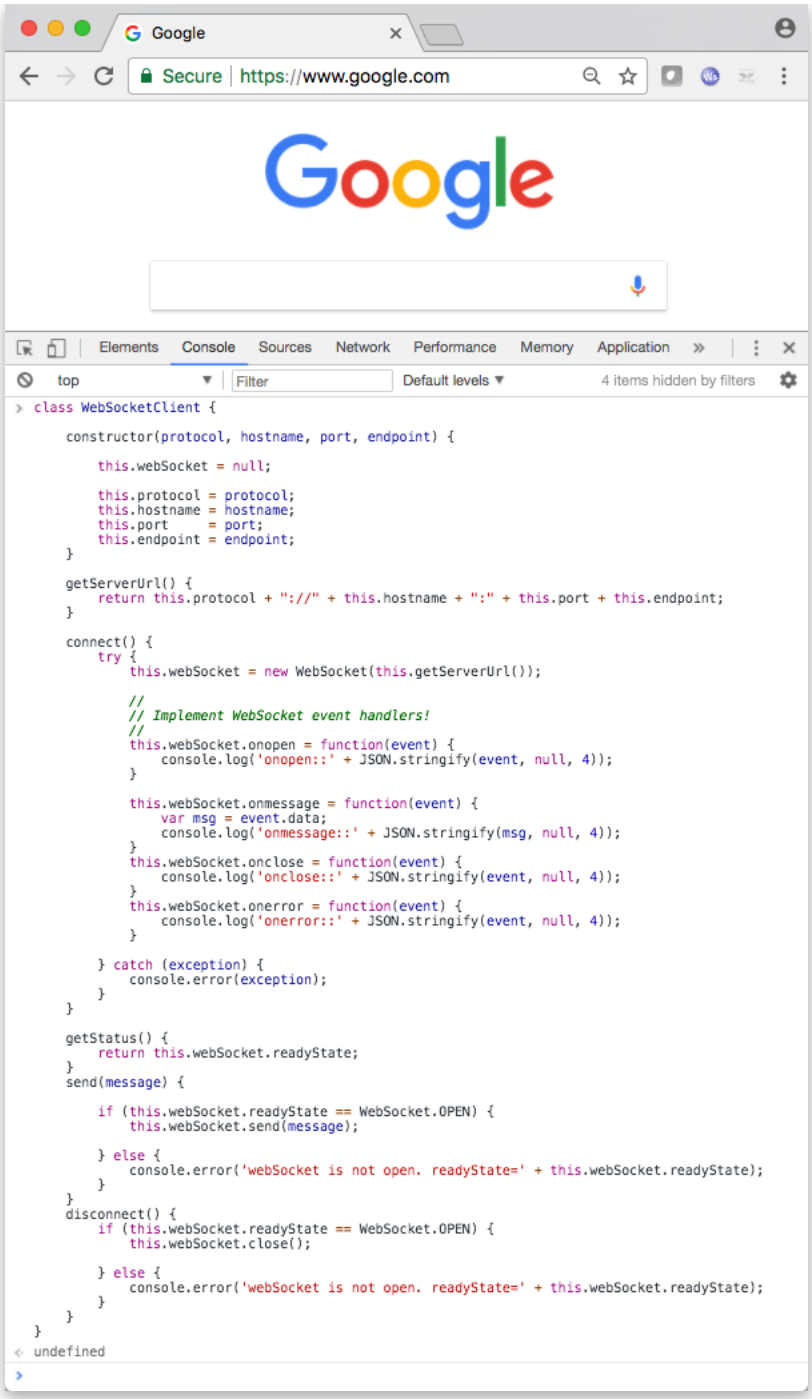
3 Testing the WebSocket Server Endpoint from Web Browser

The ***JavaScript*** source code given below creates a class called **`WebSocketClient`** which implements a basic WebSocket client.

```
1.  class WebSocketClient {
2.
3.      constructor(protocol, hostname, port, endpoint) {
4.
5.          this.webSocket = null;
6.
7.          this.protocol = protocol;
8.          this.hostname = hostname;
9.          this.port     = port;
10.         this.endpoint = endpoint;
11.     }
```

```
18.
19.         try {
20.             this.webSocket = new WebSocket(this.getServerUrl());
21.
22.             //
23.             // Implement WebSocket event handlers!
24.             //
25.             this.webSocket.onopen = function(event) {
26.                 console.log('onopen::' + JSON.stringify(event, null, 4));
27.             }
28.
29.             this.webSocket.onmessage = function(event) {
30.                 var msg = event.data;
31.                 console.log('onmessage::' + JSON.stringify(msg, null, 4));
32.             }
33.
34.             this.webSocket.onclose = function(event) {
35.                 console.log('onclose::' + JSON.stringify(event, null, 4));
36.             }
37.
38.             this.webSocket.onerror = function(event) {
39.                 console.log('onerror::' + JSON.stringify(event, null, 4));
40.             }
41.
42.         } catch (exception) {
43.             console.error(exception);
44.         }
45.     }
46.
47.     getStatus() {
48.         return this.webSocket.readyState;
49.     }
50.
51.     send(message) {
52.
53.         if (this.webSocket.readyState == WebSocket.OPEN) {
54.             this.webSocket.send(message);
55.
56.         } else {
57.             console.error('webSocket is not open. readyState=' + this.webSocket.readyState);
58.         }
59.     }
60.
61.     disconnect() {
62.
63.         if (this.webSocket.readyState == WebSocket.OPEN) {
64.             this.webSocket.close();
65.
66.         } else {
67.             console.error('webSocket is not open. readyState=' + this.webSocket.readyState);
68.         }
69.     }
70. }
```

The JavaScript code can be executed with Chrome (*or any other browser*), using its built-in ***Developer Tools***. Open a new window in Chrome, copy the above JavaScript WebSocket client code and paste it into the console provided by the Chrome Developer tools.

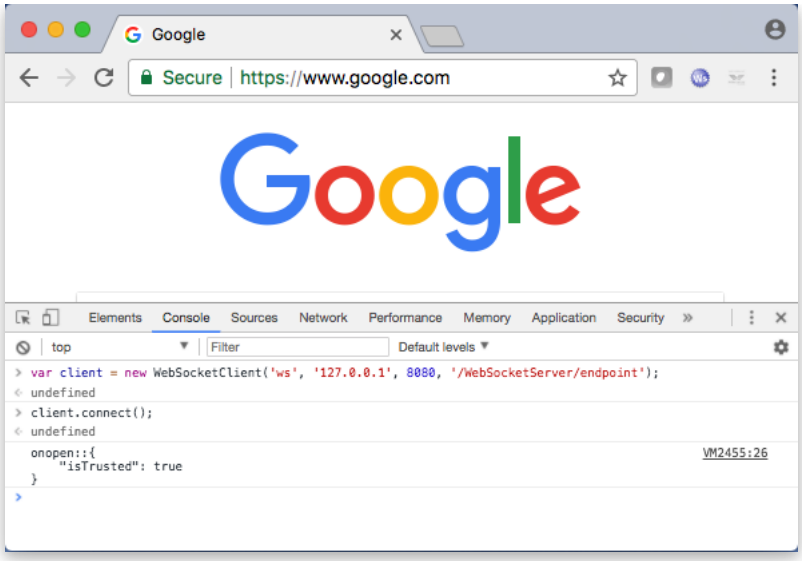


The **`WebSocketClient`** class has now been defined on the current page. In the console, enter the JavaScript code below to create a new object of that class.

```
var client = new WebSocketClient('ws', '127.0.0.1', 8080, '/WebSocketServer/endpoint');
```

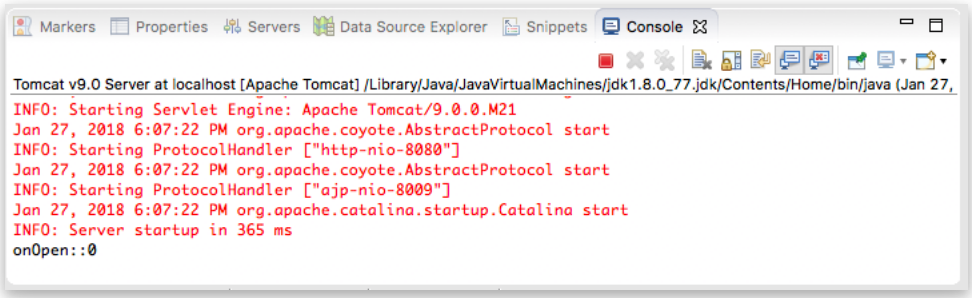
Then call the **`connect`** method to open a new connection to the WebSocket server endpoint. The browser will call the event handler **`onopen`** once the connection has been established.

```
client.connect();
```



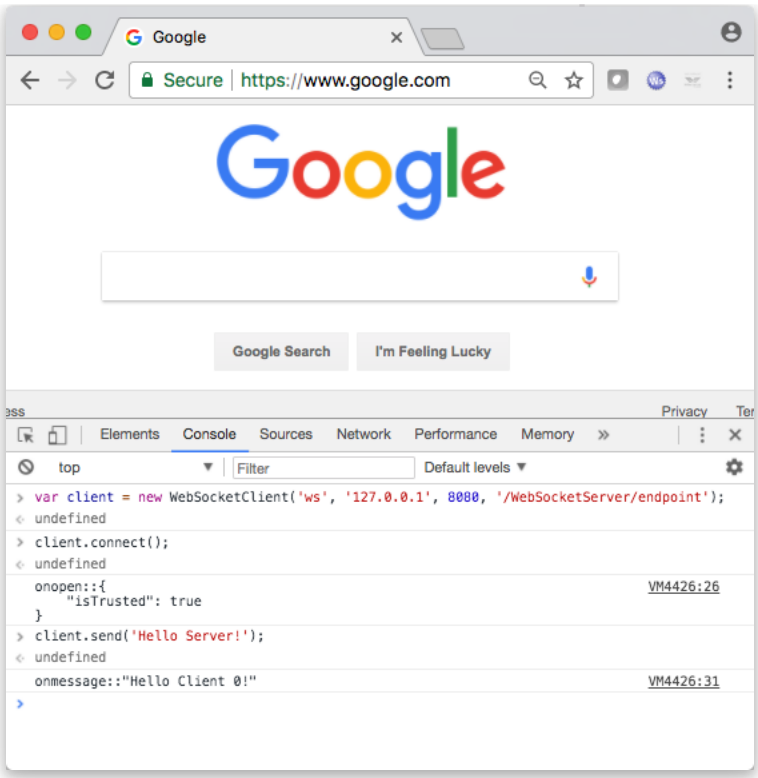
At this point, the container that is hosting the WebSocket server endpoint will call the server implementation's **`onOpen`** method. In this example, the server simply prints the session Id, which is **`0`** here.

- See the **`onOpen`** method of the **`MyWebSocket.java`** class from section 1.
- See the Oracle documentation for details on javax.websocket.Session.

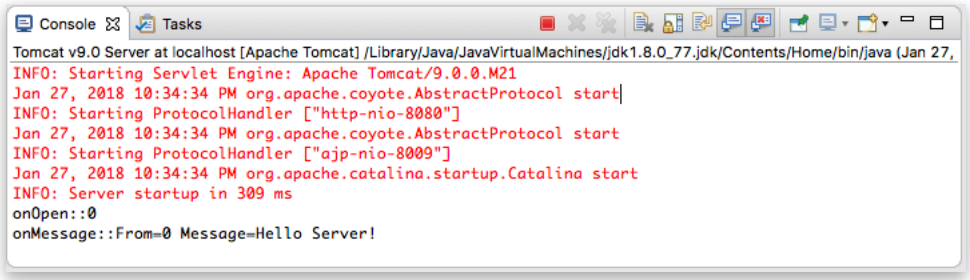


On the client side, execute the **`send(String message)`** method using the JavaScript below to send a new message to the server.

```
client.send('Hello Server!');
```



The server returns a greeting message to the client, here *"Hello Client 0!"* See ***line 34*** of the **`MyWebSocket.java`** class from section 1. At the same time, the container that is hosting the WebSocket server endpoint will call the server implementation's **`onMessage`** method. The server simply prints the client's message to the console.



4 Automatically Pushing Notifications to WebSocket Clients

In this section, the **`MyWebSocket.java`** class is changed a bit so that clients can subscribe to a ***notification service*** that will periodically send the server's current system time. The clients that want to receive those notifications need to provide a URL query string parameter as shown below:

```
ws://127.0.0.1:8080/WebSocketServer/endpoint?push=TIME
```

New code has been added to the **`onOpen`** method between ***line 21*** and ***line 32*** as shown below. The **`PushTimeService`** class maintains a collection of active Session objects and periodically sends the current server time to all clients that have subscribed.

```
1. package com.pgx.java.web;
```

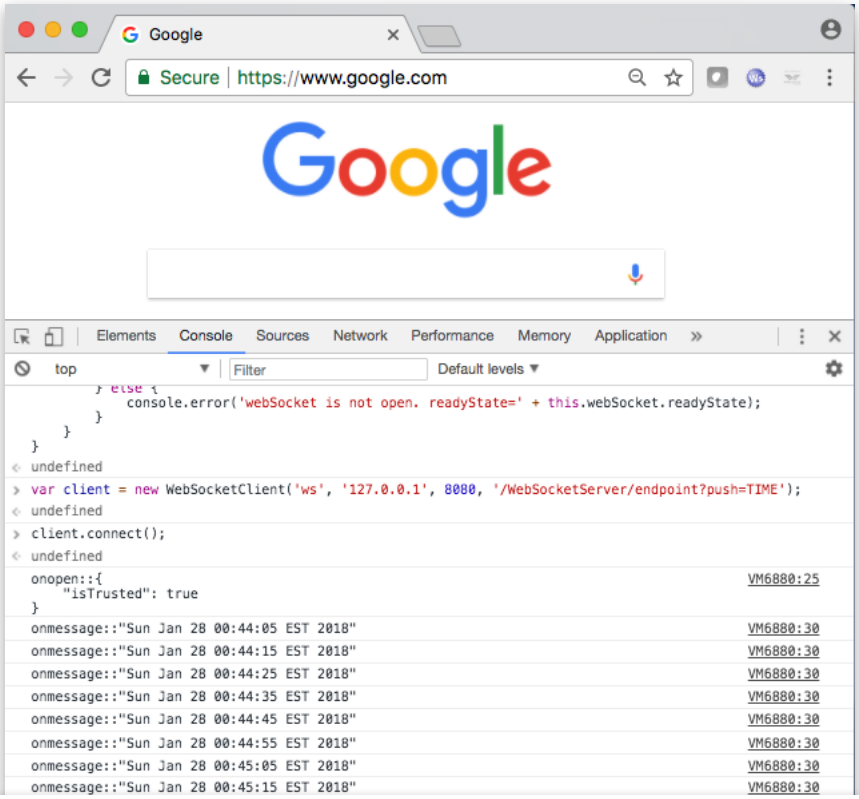
```
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;

@ServerEndpoint("/endpoint")
public class MyWebSocket {

    @OnOpen
    public void onOpen(Session session) {
        System.out.println("onOpen::" + session.getId());

        ////////////////////////////////////////////////////////////////////////////
        // Access request parameters from URL query String.
        // If a client subscribes, add Session to PushTimeService.
        //
        Map<String, List<String>> params = session.getRequestParameterMap();

        if (params.get("push") != null && (params.get("push").get(0).equals("TIME"))) {

          PushTimeService.initialize();
          PushTimeService.add(session);
        }
        ////////////////////////////////////////////////////////////////////////////
    }

    @OnClose
    public void onClose(Session session) {
        System.out.println("onClose::" +  session.getId());
    }

    @OnMessage
    public void onMessage(String message, Session session) {
        System.out.println("onMessage::From=" + session.getId() + " Message=" + message);

        try {
            session.getBasicRemote().sendText("Hello Client " + session.getId() + "!");

        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @OnError
    public void onError(Throwable t) {
        System.out.println("onError::" + t.getMessage());
    }
}
```

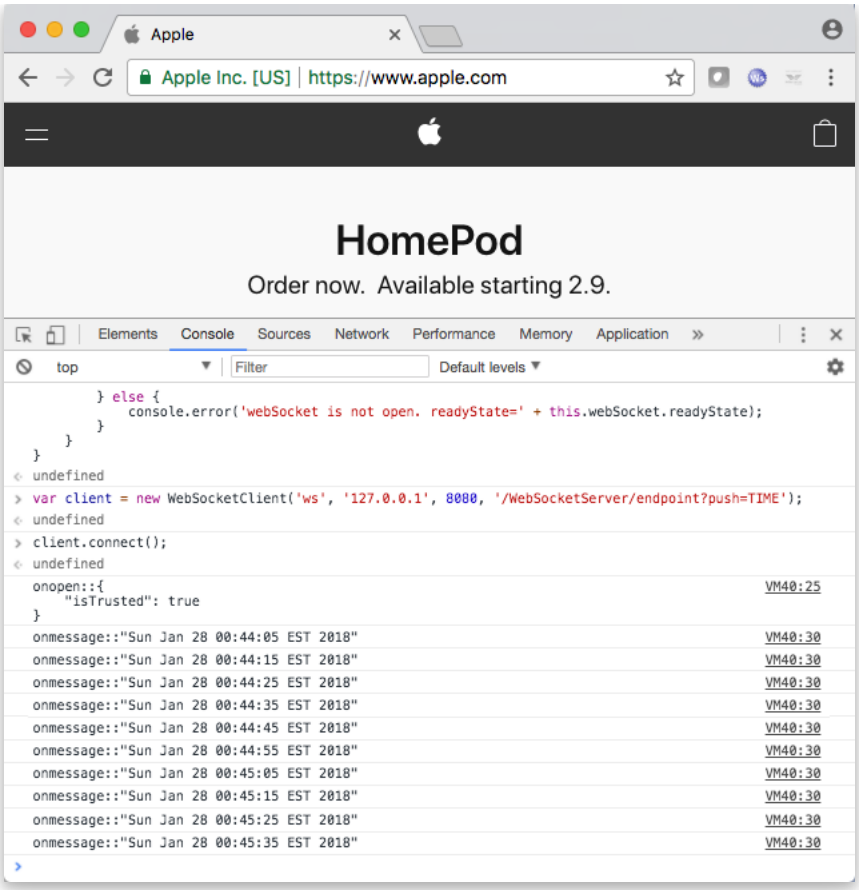
The Java source code of the **PushTimeService.java** class is given below. This class implements a singleton and runs in a separate thread. ***Every 10 seconds***, it iterates over its collection of subscribed clients and sends the server time to each one as a WebSocket message. If the session has been closed, it is removed from the collection.

```
package com.pgx.java.web;

import java.util.*;
import javax.websocket.Session;

public class PushTimeService implements Runnable {

    private static PushTimeService instance;
    private static Map<String, Session> sMap = new HashMap<String, Session>();

    private PushTimeService() {}

```

```
            instance = new PushTimeService();
            new Thread(instance).start();
        }
    }

    @Override
    public void run() {

        while (true) {
            try {
                Thread.sleep(10*1000);

                for (String key : sMap.keySet()) {

                    Session s = sMap.get(key);

                    if (s.isOpen()) {
                        Date d = new Date(System.currentTimeMillis());
                        s.getBasicRemote().sendText(d.toString());

                    } else {
                        sMap.remove(key);
                    }
                }

            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

In the below scenario, 2 WebSocket clients have been started in different browsers using the **`WebSocketClient`** JavaScript class from section 3. Both have subscribed to receive the server system time.

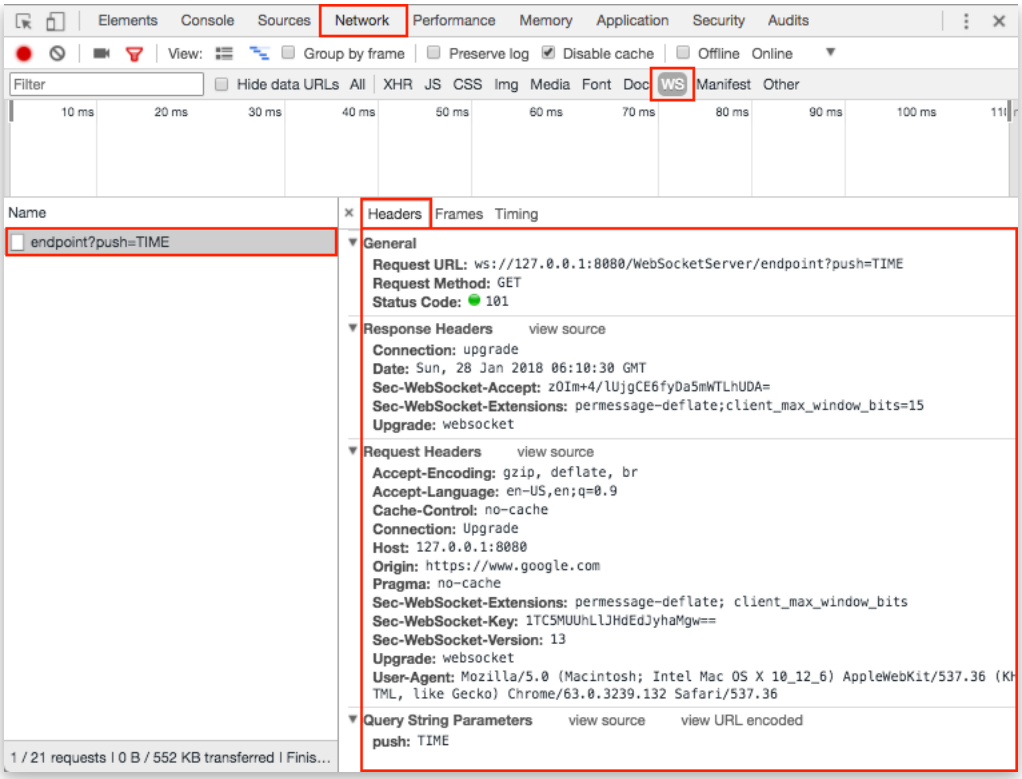




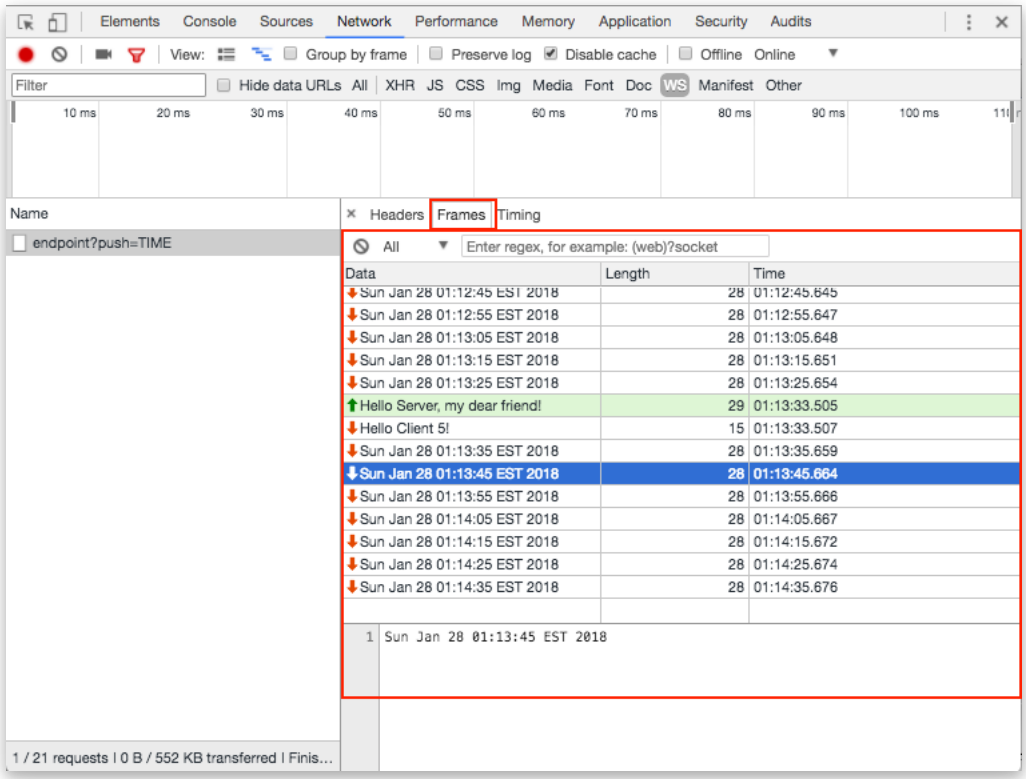
Similar approaches can be used to push a variety of event driven data from the server to clients in real time. Since the client's WebSocket connection resides in the UI layer, WebSockets are a good solution (*alternative to polling*) for driving real-time UI components such as stock tickers or notifications.

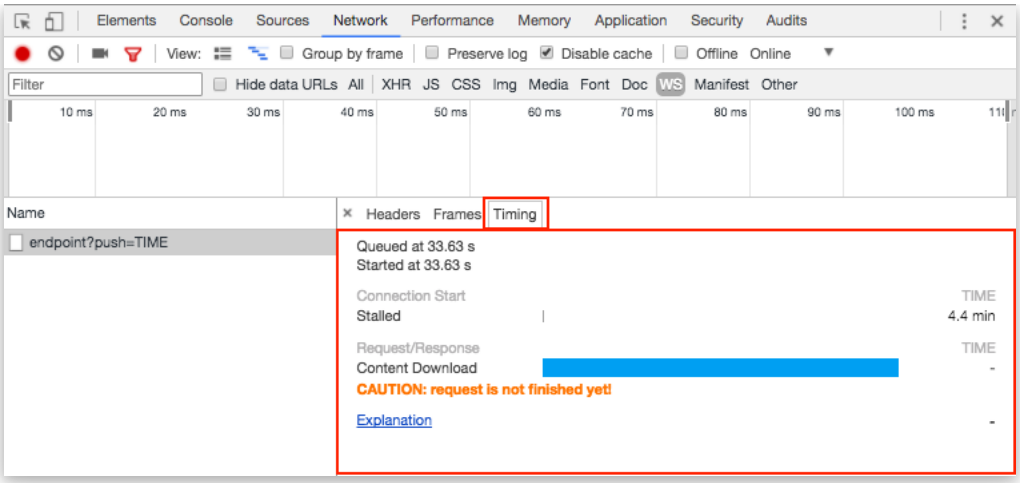
5 Monitoring WebSocket Traffic with Chrome Developer Tools

The Chrome Developer Tools provide means for basic monitoring of WebSocket traffic. Use the ***Network tab*** and then filter the traffic with the ***WS*** button to only show WebSockets. The ***Headers tab*** shows the client http request and the server response. The ***Query String Parameters*** are listed at the bottom.



Note that the server responds with a ***connection upgrade*** in the response header to upgrade the http connection to a persistent WebSocket connection. See the [protocol upgrade mechanism] for details on this process. The ***Frames tab*** shows all outgoing and incoming WebSocket messages. Click on a frame to see its content.





Related Posts

- Creating a Simple Java TCP/IP Server and Client Socket
- Creating a Simple Java UDP Server and Client Socket

2 THOUGHTS ON “EXAMPLE OF CREATING A WEBSOCKET SERVER IN JAVA”

Pingback: Java websocket client with pure javax.websocket – Ask Javascript Questions

Pingback: How to embed a websocket pushed timer into a web page? - Tutorial Guruji

Leave a Reply

You must be logged in to post a comment.

This site uses Akismet to reduce spam. Learn how your comment data is processed.