STOMP Over WebSocket

What is STOMP?

STOMP is a simple text-orientated messaging protocol. It defines an interoperable wire format so that any of the available STOMP clients can communicate with any STOMP message broker to provide easy and widespread messaging interoperability among languages and platforms (the STOMP web site has a list of STOMP client and server implementations.

What is the WebSocket API?

WebSockets are "TCP for the Web".

When Google announced the availability of WebSocket in Google Chrome, it explained the idea behind WebSockets:

> The WebSocket API enables web applications to handle bidirectional communications with server-side process in a straightforward way. Developers have been using XMLHttpRequest ("XHR") for such purposes, but XHR makes developing web applications that communicate back and forth to the server unnecessarily complex. XHR is basically asynchronous HTTP, and because you need to use a tricky technique like long-hanging GET for sending data from the server to the browser, simple tasks rapidly become complex. As opposed to XMLHttpRequest, WebSockets provide a real bidirectional communication channel in your browser. Once you get a WebSocket connection, you can send data from browser to server by calling a send() method, and receive data from server to browser by an onmessage event handler.
>
> In addition to the new WebSocket API, there is also a new protocol (the "WebSocket Protocol") that the browser uses to communicate with servers. The protocol is not raw TCP because it needs to provide the browser's "same-origin" security model. It's also not HTTP because WebSocket traffic differers from HTTP's request-response model. WebSocket communications using the new WebSocket protocol should use less bandwidth because, unlike a series of XHRs and hanging GETs, no headers are exchanged once the single connection has been established. To use this new API and protocol and take advantage of the simpler programming model and more efficient network traffic, you do need a new server implementation to communicate with.

The API is part of HTML5 and is supported (at various degree...) by most modern Web Browsers (including Google Chrome, Firefox and Safari on Mac OS X and iOS).

Protocol Support

This library supports multiple version of STOMP protocols:

- STOMP 1.0
- STOMP 1.1 (including heart-beating)

Server Requirements

This library is not a *pure* STOMP client. It is aimed to run on the WebSockets protocol which is not TCP. Basically, the WebSocket protocol requires a *handshake* between the browser's client and the server to ensure the browser's "same-origin" security model remains in effect.

This means that this library can not connect to regular STOMP brokers since they would not understand the handshake initiated by the WebSocket which is not part of the STOMP protocol and would likely reject the connection.

There are ongoing works to add WebSocket support to STOMP broker so that they will accept STOMP connections over the WebSocket protocol.

HornetQ

HornetQ is the Open Source messaging system developed by Red Hat and JBoss.

To start HornetQ with support for STOMP Over WebSocket, download the latest version and run the following steps:

```
$ cd hornetq-x.y.z/examples/jms/stomp-websockets
$ mvn clean install
...
INFO: HQ221020: Started Netty Acceptor version 3.6.2.Final-c0d783c localhost:61614 for STOMP_WS
Apr 15, 2013 1:15:33 PM org.hornetq.core.server.impl.HornetQServerImpl$SharedStoreLiveActivatior
INFO: HQ221007: Server is now live
Apr 15, 2013 1:15:33 PM org.hornetq.core.server.impl.HornetQServerImpl start
INFO: HQ221001: HornetQ Server version 2.3.0.CR2 (black'n'yellow2, 123) [c9e29e45-a5bd-11e2-976a
```

HornetQ is now started and listens to STOMP over WebSocket on the port `61614`.
It accepts *WebSocket connections* from the URL `ws://localhost:61614/stomp`

To configure and run HornetQ with STOMP Over WebSocket enabled, follow the instructions.

ActiveMQ

ActiveMQ is the Open Source messaging system developed by Apache. Starting with 5.4 snapshots, ActiveMQ supports STOMP Over WebSocket.

To configure and run ActiveMQ with STOMP Over WebSocket enabled, follow the instructions.

ActiveMQ Apollo

ActiveMQ Apollo is the next generation of ActiveMQ broker. From the start, Apollo supports STOMP Over WebSocket.

To configure and run Apollo with STOMP Over WebSocket enabled, follow the instructions.

RabbitMQ

RabbitMQ is Open Source messaging system sponsored by VMware.

To configure and run RabbitMQ with STOMP Over WebSocket enabled, follow the instructions to install the Web-Stomp plugin.

Stilts & Torquebox

Stilts is a STOMP-native messaging framework which aims to address treating STOMP as primary contract for messaging, and integrating around it, instead of simply applying STOMP shims to existing services.

TorqueBox uses the Stilts project to provide its WebSockets and STOMP stack.

Download stomp.js JavaScript file

You can download stomp.js to use it in your Web applications

A minified version is also provided to be used in production.

This JavaScript file is generated from CoffeeScript files. See the Contribute section to download the source code or browse the annotated source code.

STOMP API

STOMP Frame

STOMP Over WebSocket provides a straightforward mapping from a STOMP frame to a JavaScript object.

Frame Object

Property	Type	Notes
`command`	String	name of the frame (`"CONNECT"`, `"SEND"`, etc.)
`headers`	JavaScript object	
`body`	String	

The `command` and `headers` properties will always be defined but the `headers` can be empty if the frame has no headers. The `body` can be `null` if the frame does not have a body.

Create a STOMP client

In a Web browser with regular Web Socket

STOMP JavaScript clients will communicate to a STOMP server using a `ws://` URL.

To create a STOMP client JavaScript object, you need to call `Stomp.client(url)` with the URL corresponding to the server's WebSocket endpoint:

```
var url = "ws://localhost:61614/stomp";
var client = Stomp.client(url);
```

The `Stomp.client(url, protocols)` can also be used to override the default subprotocols provided by the library: `['v10.stomp', 'v11.stomp]'` (for STOMP 1.0 & 1.1 specifications). This second argument can either be a single string or an array of strings to specify multiple subprotocols.

In the Web browser with a custom WebSocket

Web browsers supports different versions of the WebSocket protocol. Some older browsers does not provide the WebSocket JavaScript or expose it under another name. By default, `stomp.js` will use the Web browser native `WebSocket` class to create the WebSocket.

However it is possible to use other type of WebSockets by using the `Stomp.over(ws)` method. This method expects an object that conforms to the WebSocket definition.

For example, it is possible to use the implementation provided by the SockJS project which falls back to a variety of browser-specific transport protocols instead:

```
<script src="http://cdn.sockjs.org/sockjs-0.3.min.js"></script>
<script>
  // use SockJS implementation instead of the browser's native implementation
  var ws = new SockJS(url);
  var client = Stomp.over(ws);
  [...]
</script>
```

Use `Stomp.client(url)` to use regular WebSockets or use `Stomp.over(ws)` if you required another type of WebSocket.

Apart from this initialization, the STOMP API remains the same in both cases.

In a node.js application

The library can also be used in node.js application by using the stompjs npm package.

```
$ npm install stompjs
```

In the node.js app, require the module with:

```
var Stomp = require('stompjs');
```

To connect to a STOMP broker over a *TCP socket*, use the `Stomp.overTCP(host, port)` method:

```
var client = Stomp.overTCP('localhost', 61613);
```

To connect to a STOMP broker over a *Web Socket*, use instead the `Stomp.overWS(url)` method:

```
var client = Stomp.overWS('ws://localhost:61614/stomp');
```

Apart from this initialization, the STOMP API remains the same whether it is running in a Web browser or in node.js application.

Connection to the server

Once a STOMP client is created, it must call its `connect()` method to effectively connect and authenticate to the STOMP server. The method takes two mandatory arguments, `login` and `passcode` corresponding to the user credentials.

Behind the scene, the client will open a connection using a WebSocket and send a CONNECT frame.

The connection is done asynchronously: you have no guarantee to be effectively connected when the call to `connect` returns. To be notified of the connection, you need to pass a `connect_callback` function to the `connect()` method:

```
var connect_callback = function() {
  // called back after the client is connected and authenticated to the STOMP server
};
```

But what happens if the connection fails? the `connect()` method accepts an optional `error_callback` argument which will be called if the client is not able to connect to the server. The callback will be called with a single argument, an error object corresponding to STOMP ERROR frame:

```
var error_callback = function(error) {
  // display the error's message header:
  alert(error.headers.message);
};
```

The `connect()` method accepts different number of arguments to provide a simple API to use in most cases:

```
client.connect(login, passcode, connectCallback);
client.connect(login, passcode, connectCallback, errorCallback);
client.connect(login, passcode, connectCallback, errorCallback, host);
```

where `login`, `passcode` are strings and `connectCallback` and `errorCallback` are functions (some brokers also require to pass a host String).

The `connect()` method also accepts two other variants if you need to pass additional headers:

```
client.connect(headers, connectCallback);
client.connect(headers, connectCallback, errorCallback);
```

where `header` is a map and `connectCallback` and `errorCallback` are functions.

Please note that if you use these forms, you **must** add the `login`, `passcode` (and eventually `host`) headers yourself:

```
    var headers = {
      login: 'mylogin',
      passcode: 'mypasscode',
      // additional header
      'client-id': 'my-client-id'
    };
    client.connect(headers, connectCallback);
```

To disconnect a client from the server, you can call its `disconnect()` method. The disconnection is asynchronous: to be notified when the disconnection is effective, the `disconnect` method takes an optional `callback` argument.

```
client.disconnect(function() {
  alert("See you next time!");
};
```

When a client is disconnected, it can no longer send or receive messages.

Heart-beating

If the STOMP broker accepts STOMP 1.1 frames, heart-beating is enabled by default.

The `client` object has a `heartbeat` field which can be used to configure heart-beating by changing its `incoming` and `outgoing` integer fields (default value for both is `10000`ms):

```
    client.heartbeat.outgoing = 20000; // client will send heartbeats every 20000ms
    client.heartbeat.incoming = 0;     // client does not want to receive heartbeats
                                       // from the server
```

The heart-beating is using `window.setInterval()` to regularly send heart-beats and/or check server heart-beats.

Send messages

When the client is connected to the server, it can send STOMP messages using the `send()` method. The method takes a mandatory `destination` argument corresponding to the STOMP destination. It also takes two optional arguments: `headers`, a JavaScript object containing additional message headers and `body`, a String object.

```
client.send("/queue/test", {priority: 9}, "Hello, STOMP");
```

The client will send a STOMP SEND frame to `/queue/test` destination with a header `priority` set to `9` and a body `Hello, STOMP`.

If you want to send a message with a body, you must also pass the `headers` *argument. If you have no headers to pass, use an empty JavaScript literal* `{}`*:*

```
client.send(destination, {}, body);
```

Subscribe and receive messages

To receive messages in the browser, the STOMP client must first subscribe to a destination.

You can use the `subscribe()` method to subscribe to a destination. The method takes 2 mandatory arguments: `destination`, a String corresponding to the destination and `callback`, a function with one `message` argument and an *optional* argument `headers`, a JavaScript object for additional headers.

```
var subscription = client.subscribe("/queue/test", callback);
```

The `subscribe()` methods returns a JavaScript obect with 1 attribute, `id`, that correspond to the client subscription ID and one method `unsubscribe()` that can be used later on to unsubscribe the client from this destination.

By default, the library will generate an unique ID if there is none provided in the headers. To use your own ID, pass it using the `headers` argument:

```
var mysubid = '...';
var subscription = client.subscribe(destination, callback, { id: mysubid });
```

The client will send a STOMP SUBSCRIBE frame to the server and register the callback. Every time the server send a message to the client, the client will in turn call the callback with a STOMP Frame object corresponding to the message:

```
callback = function(message) {
  // called when the client receives a STOMP message from the server
  if (message.body) {
    alert("got message with body " + message.body)
  } else {
    alert("got empty message");
  }
});
```

The `subscribe()` method takes an optional `headers` argument to specify additional headers when subscribing to a destination:

```
var headers = {ack: 'client', 'selector': "location = 'Europe'"};
client.subscribe("/queue/test", message_callback, headers);
```

The client specifies that it will handle the message acknowledgement and is interested to receive only messages matching the selector `location = 'Europe'`.

If you want to subscribe the client to multiple destinations, you can use the same callback to receive all the messages:

```
onmessage = function(message) {
  // called every time the client receives a message
}
var sub1 = client.subscribe("queue/test", onmessage);
var sub2 = client.subscribe("queue/another", onmessage);
```

To stop receiving messages, the client can use the `unsubscribe()` method on the object returned by the `subscribe()` method.

```
var subscription = client.subscribe(...);

...
```

```
subscription.unsubscribe();
```

JSON support

The body of a STOMP message must be a `String`. If you want to send and receive JSON objects, you can use `JSON.stringify()` and `JSON.parse()` to transform the JSON object to a String and vice versa.

```
var quote = {symbol: 'APPL', value: 195.46};
client.send("/topic/stocks", {}, JSON.stringify(quote));

client.subcribe("/topic/stocks", function(message) {
  var quote = JSON.parse(message.body);
  alert(quote.symbol + " is at " + quote.value);
};
```

Acknowledgment

By default, STOMP messages will be automatically acknowledged by the server before the message is delivered to the client.

The client can chose instead to handle message acknowledgement by subscribing to a destination and specify a `ack` header set to `client` or `client-individual`.

In that case, the client must use the `message.ack()` method to inform the server that it has acknowledge the message.

```
var subscription = client.subscribe("/queue/test",
  function(message) {
    // do something with the message
    ...
    // and acknowledge it
    message.ack();
  },
  {ack: 'client'}
);
```

The `ack()` method accepts a `headers` argument for additional headers to acknowledge the message. For example, it is possible to acknowledge a message as part of a transaction and ask for a receipt when the `ACK` STOMP frame has effectively be processed by the broker:

```
var tx = client.begin();
message.ack({ transaction: tx.id, receipt: 'my-receipt' });
tx.commit();
```

The `nack()` method can also be used to inform STOMP 1.1 brokers that the client did *not* consume the message. It takes the same arguments than the `ack()` method.

Transactions

Messages can be sent and acknowledged *in a transaction*.

A transaction is started by the client using its `begin()` method which takes an optional `transaction`, a String which uniquely identifies the transaction. If no `transaction` is passed, the library will generate one automatically.

This methods returns a JavaScript object with a `id` attribute corresponding to the transaction ID and two methods:

- `commit()` to commit the transaction
- `abort()` to abort the transaction

The client can then send and/or acknowledge messages in the transaction by specifying a `transaction` set with the transaction `id`.

```
// start the transaction
var tx = client.begin();
// send the message in a transaction
client.send("/queue/test", {transaction: tx.id}, "message in a transaction");
// commit the transaction to effectively send the message
tx.commit();
```

If you forget to add the `transaction` *header when calling* `send()` *the message will not be part of the transaction and will be sent directly without waiting for the completion of the transaction.*

```
var txid = "unique_transaction_identifier";
// start the transaction
var tx = client.begin();
// oops! send the message outside the transaction
client.send("/queue/test", {}, "I thought I was in a transaction!");
tx.abort(); // Too late! the message has been sent
```

Debug

There are few tests in the code and it is helpful to see what is sent and received from the library to debug application.

The client can set its `debug` property to a function with takes a `String` argument to see all the debug statements of the library:

```
client.debug = function(str) {
  // append the debug log to a #debug div somewhere in the page using JQuery:
  $("#debug").append(str + "\n");
};
```

By default, the debug messages are logged in the browser window's console.

Example

The source code contains a chat example in `examples/chat/index.html`

You need to start a STOMP server with support for WebSocket (using for example HornetQ).

Click on the `Connect` button to connect to the server and subscribe to the `/queue/test/` queue.

You can then type messages in the form at the bottom of the page to send STOMP messages to the queue. Messages received by the client will be displayed at the top of the page.

You can also send regular STOMP messages and see them displayed in the browser. For example using directly `telnet` on STOMP default port:

```
$ telnet localhost 61613
CONNECT
login:guest
passcode:guest

^@

CONNECTED
session:1092296064
```

`^@` is a null (`control-@` in ASCII) byte.

```
SEND
destination:/queue/test
```

```
Hello from TCP!
^@
```

You should now have received this message in your browser.

Contribute

The source code is hosted on GitHub:

```
git clone git://github.com/jmesnil/stomp-websocket.git
```