EXPOSING JMS-BASED TRADING SOA MESSAGING API TO JS WEB CLIENTS

Claudiu VINȚE

Bucharest University of Economic Studies, Romania

claudiu.vinte@ie.ase.ro

Andrei TOMA

Bucharest University of Economic Studies, Romania

andrei.toma@ie.ase.ro

Alexandru BĂRBULESCU

Bucharest University of Economic Studies, Romania

alexbarbulescu@ie.ase.ro

Abstract. *One of our ongoing research paths upon the academic trading system ASETS has been focused on how to expose, in a coherent yet flexible manner, the service oriented architecture functionality of the server side to a broader range of front-end client applications. This paper introduces our development regarding the communication mechanism between JavaScript web client, specifically the map of the Bucharest Stock Exchange (BSE), and the JMS-linked services responsible for creating the market map allocation and feeding the web application with updates, whenever they occur on the server side. ASETS trading architecture consists in a comprehensive collection of services interconnected through a custom-designed message oriented API built on top of JMS. Online financial trading systems require data push for near real-time stock price delivery. The data flow generated asynchronously on the server side has to be pushed to the front end without relying on explicit request from the client. Our solution proposes a gateway component that communicates via websockets with the JavaScript web application, along with an API extension that facilitates in particular the exposure of JMS's pub-sub communication model to JavaScript web clients.*

Keywords: JMS, Message Pushing, Publisher-Subscriber, Trading Architectures, WebSockets

JEL classification: G23, G24

DOI: 10.12948/ie2019.02.06

1. Introduction

Trading activity in general and high frequency trading in particular require that information is delivered on the fly as soon as it is created. Web pages and mobile applications have to be updated in real time or near real time. Push technology can offers benefits for a multitude of financial services: online trading platforms for capital markets, live price dissemination, order submission, spread betting etc.

According to Alessandro Alinone from Lightstreamer Technology, there have been manifested four waves of data push [A].

Introduced in the middle of the 90's, the polling technique was the first attempt from the browser side for obtaining near real-time information from the server-side: the browser sends HTTP requests at regular intervals, expecting an immediate response from the server. This is an acceptable solution if the exact interval of message delivery is known, and consequently the client request can be more or less synchronized, the therefore have the client requests sent only

when information is available on the server. However, real-time data is seldom generated at regular time intervals and, inevitably, there are made unnecessary requests. The result is that, in low-message-rate situations, many connections are opened and closed without being necessary [B].

In order to simulate server side push technologies, around the year 2000 Comet long polling implemented a mechanism based on delaying the completion of a HTTP response to deliver messages to the client. Comet-based push is generally implemented in JavaScript and uses connection strategies such as long-polling or streaming [A].

In the case of long-polling, the browser sends a request to the server and the server keeps the connection open for a previously set period, or timeout period. If new data is generated on the server-side, then a notification is sent back to the client within that period of time, as response. If notification is not available for the client within the set timeframe, then the server sends a response as a NACK to terminate the open request. It has to be emphasised that, in cases that involve flows of generated with high frequency, the long-polling technique does not provide any substantial performance improvements over regular polling, and can even generate counterproductive outcomes.

Over the past decade the software solutions that have been implemented through service oriented architectures employed various messaging models and relied preponderantly on message oriented middlewares (MOM). Message brokers based on Java Message Service (JMS) API in particular, have proved to offer the most flexible way of integrating collections of loosely coupled services, particularly suitable for server-side distributed architecture of trading systems [1].

2. The need to expose server-side JMS messages via a JavaScript web application

ASETS (*ASE Trading System*) is a service oriented architecture implemented integrally in Java and based *Java Message Service* (JMS) for delivering asynchronous durable message- based communication mechanisms. Our initial solution regarding the Bucharest Stock Exchange (BSE) market map had been implemented as a Java applet that can launch in a web browser [2]. Since many main stream browsers had either removed or planned to remove security standards based on plugin support, Oracle has advised Java developers to migrate from Java applets to plugin-free Java Web-start technologies [3]. This is an option that might itself become unsuitable and deprecated sometime in the future and, on the other hand, is still a solution that cannot be considered a viable alternative to web applications which are run on any platform that support a modern full-fledged web browser, particularly on mobile devices. In this context, replacing the original ASETS Java applet for the BSE market map, built with the *Capital Allocation through Rectangular Distribution* (CARD) algorithm, with a JavaScript web application appears to be the natural solution at this point in time.

As ASETS is built on a SOA architecture, it offers a pool of components on the server-side, each of them exposing its functionality as a service provider to the other components [4]. If using a *point-to-point* (p2p) asynchronous communication model is necessary, then requests for services and replies are passed through a message-oriented middleware (MOM) that employs messages queues. Producer services load messages on the queue and consumer services consume these messages as soon as they are produced. In this paradigm, the components of a MOM-based architecture are client applications, messages, and message providers or brokers. If the functional scenario requires publisher-subscriber model, then a *topic* instance is created on the message broker on which the publisher service posts messages from which the subscriber services receive the messages through a multicast type of communication mechanism [5]. The Java CARD applet communicates with the services from server-side using the following ASETS API Java classes built on JMS [6]:

- **RectangleListSenderWithReply** – p2p request to CARD server for the BSE map allocation;
- **RectangleListReceiverWithReply** – p2p reply from CARD server, containing the actual map allocation data;
- **InstrumentListSenderWithReply** – p2p request to ASETS DB server for the instruments' related data that has to be posted on the map;
- **InstrumentListReceiverWithReply** – p2p reply from ASETS DB server, containing data related to stocks listed on the regular section of BSE;
- **PriceListSenderWithReply** – p2p request to ASETS DB server for bulk prices of listed stock on BSE, the request may concern historical data as well;
- **PriceListReceiverWithReply** – p2p reply from ASETS DB server, containing the price list according to the request;
- **PricePublisher** – the publisher of individual price data received from BSE through ASETS *Delayed Data Feed* (DDF);
- **PriceSubscriber** – the subscriber to PRICE_TOPIC for receiving price data whenever is available from the market.

All these API classes are available across the system for all Java applications. If this functionality is to be exposed on the web, a mechanism that would have to cover the data exchange between the web application and the ASETS server-side is required. The aforementioned mechanism should also be able to receive messages generated server-side, whenever they are available/generated.

3. The proposed solution for linking the JavaScript web application with the JMS-based ASETS server-side

In order to make the pub-sub model provided by JMS available on the web, we employed websocket-based communication between the JavaScript web application and the ASETS services [7]. While JS websockets do not require JSON content for messaging, it is advantageous to use a serialization scheme which is language agnostic, hence the need for designing and implementing a component that plays the role of a gateway for converting the JSON requests to Java serialized JMS *ObjectMessage* type, and vice versa. This gateway component would be responsible for pushing the marshaled Java JMS messages to JSON-based ones toward the web application, whenever these messages are produced by ASETS services and consumed by the gateway.

The overall architecture of our proposed solution is presented in *Figure 1*. The component ASETS JavaScript to Java Gateway (JSJG) is a Java service which resides on the server-side, while the JavaScript BVB Market Map is a web application running in web browser. These two new components communicate through websockets, and exchange JSON-based messages [11-13].

The JS web application is Single Page Application (SPA), and employs Data-Driven Documents (D3 version 5 library) to draw the market map, and reflect the price updates received as pushed messages from the server-side [8].

The JavaScript web application fetches from the gateway the data related to the map allocation and prices concerning the listed stocks that are part of the map. The below JS code snippets show how the processed is conducted.

```
...
//get initial data
let response = await fetch(`${SERVER}/rectangles?batch=1`)
let data = await response.json()
```

```
//connect to web socket server
const socket = new WebSocket('ws://ip_address:port')
...
```

Figure 1. The overall proposed architecture, including a JS to Java (and back) Gateway (JSJG)

Once the initial map allocation is received, along with the bulk price data, which may contain the closing prices of the listed stocks from the previous trading day, then the map updates the colour of the rectangles, representing the listed companies, based on the price updates received from the JSJG gateway.

```
...
//on receiving a message from the server, data is updated through d3
//new - item is added, exit - item is deleted, update - item is updated
socket.addEventListener('message', (event) => {
    let message = JSON.parse(event.data)
    let dataUpdate = message.content
    console.warn(dataUpdate)
})
...
```

The below *Figure 2* depicts the BSE market map created by the JS web application. The D3 library offers support for showing additional information in a hovering balloon [14-15].

On the server side, the JSJG gateway receives from the JS web application JSON formatted messages as requests, convert them to Java JMS messages and place them on the corresponding message queue of the message broker. The p2p JMS replies are then marshaled from Java objects to JSON messages and sent back to JS market map application.

The JSJG gateway maintains websocket connections with the web application and, on the other hand, has multiple JMS connections to various JMS queues for consuming instrument data, price data, and map allocation data, along with the subscription to the topic at which a published price updates fetched from BSE. Below are some code samples.

The JSON messages that the gateway receives from the JS web client are decoded and new Java objects are created to map the <key, value> pairs received.

Figure 2. JavaScript SPA for presenting BSE market map

The code snippet below is an example concerning the construction of a Java Control object, which models a request to be placed on JMS queue, from a JSON message received from the web application.

```
...
@Override
public boolean willDecode(String string) {
    boolean decodes = false;
    messageMap = new HashMap<>();
    JsonParser parser = Json.createParser(new StringReader(string));
    while (parser.hasNext()) {
        if (parser.next() == JsonParser.Event.KEY_NAME) {
            String key = parser.getString();
            parser.next();
            String value = parser.getString();
            messageMap.put(key, value);
        }
    }
    Set keys = messageMap.keySet();
    ...
    return decodes;
}
...
```

4. Conclusions and further research

This paper presents our development regarding the communication mechanism between JavaScript web client, specifically the map of the Bucharest Stock Exchange (BSE), and the JMS-linked services responsible for creating the market map allocation and feeding the web application with updates, whenever they occur on the server side. Online financial trading systems require data push for near real-time stock price delivery. The data flow generated asynchronously on the server side has to be pushed to the front end without relying on explicit request from the client. Our solution proposes a gateway component that communicates via websockets with the JavaScript web application. We have created an ASETS API extension in order to cope with the JMS's pub-sub communication model in particular, and facilitate the

server-side message pushing to JavaScript web client. As further research and development, we intend to extend the ASETS API and the gateway functionality in order to cover the entire range of JMS messages required for designing and developing a rich JavaScript web-based application as a trading GUI for the academic trading platform ASETS.

Acknowledgment

This work was supported by a grant by UEFISCDI (“Unitatea Executivă pentru Finanțarea Învățământului Superior, a Cercetării, Dezvoltării și Inovării”), project FutureWeb (“Modelarea empirică și dezvoltarea experimentală a instrumentelor asociate tehnologiilor emergente din domeniul rețelelor sociale online”), project number: PN-III-P1-1.2-PCCDI-2017-0800, 86PCCDI/2018.

References

- [1] A. S. Tanenbaum, Maarten van Steen, *Distributed Systems - Principles and Paradigm*, Vrije Universiteit Amsterdam, The Netherlands, Prentice Hall, New Jersey, 2002, pp. 99-119, 414-488, 648-677
- [2] C. Vințe, Alexandru Jurubiță, “SOA and Web Technology for Building BSE Market Map”, *Informatica Economica Journal* vol. 16, no. 1/2012, pp. 60-69, ISSN 1453-1305, Available: <http://revistaie.ase.ro/content/61/06%20-%20Vinte.pdf>
- [3] C. Boja, M. Doinea, “Security of Mobile Clients in Event-Driven Distributed Architectures”, *Proceedings of the 6th International Conference on Security for Information Technology and Communications (SECITC'13)*, 2013
- [4] C. Vințe, “ASETS – An Academic Trading Simulation Platform”, *Informatica Economica Journal* Vol. 14, Issue 2/2010, pp 97-107, ISSN 1453-1305, Available: <http://revistaie.ase.ro/content/54/10%20Vinte.pdf>
- [5] M. Richards, Richard Monson-Haefel, David A. Chappell, *Java Message Service (Second Edition)*, O'Reilly Media Inc., Sebastopol, California, 2009
- [6] B. Snyder, Dejan Bosanac, Rob Davies, *ActiveMQ In Action*, Manning Publications Co., Stamford, 2011, pp. 247-254
- [7] C. Vințe, Alexandru Lixandru, “Leveraging Server Side Messaging Through Websocket-based Front End”, *Proceeding of the 13th International Conference on Informatics in Economy* (ISSN: 2247 – 1480)
- [8] C. Vințe, “A GCM Solution for Leveraging Server-side JMS Functionality to Android-based Trading Application”, *Informatica Economica Journal* Vol. 17, Issue 3/2013, pp 49-59, ISSN 1453-1305, Available: <http://revistaie.ase.ro/content/67/05%20-%20Vinte.pdf>
- [9] A. Alinone, “From Data Push to WebSockets - The History of Data Push and the Lightstreamer Technology”, *Lightstreamer Technology white paper*, last updated: 17 October 2018, Internet: https://lightstreamer.com/share/docs/Lightstreamer_presentation.pdf
- [10] B. Peter Lubbers & Frank Greco, “HTML5 Web Sockets: A Quantum Leap in Scalability for the Web”, last updated 17 April 2019, Internet: <https://websocket.org/quantum.html>
- [11] The latest version of Java Web Socket API, last updated 20 March 2019, Internet: <https://html.spec.whatwg.org/multipage/web-sockets.html#network>
- [12] STOMP Protocol Specification, Version 1.2, Internet: <http://stomp.github.io/stomp-specification-1.2.html>, October 22, 2012
- [13] JavaScript client websocket API, Internet: <https://developer.mozilla.org/en-US/docs/Web/API/WebSocket>
- [14] Node.js websocket API, Internet: <https://github.com/websockets/ws>
- [15] Data Driven Documents library - d3.js v5, Internet: <https://d3js.org/>