LogicBlaze

Ajax Tutorial

FUSE

The Open Source SOA Platform

09/06 Version 1.2.1

FUSE Ajax Tutorial

This is a tutorial for a real world example of a stock portfolio publisher using Ajax and Apache ActiveMQ (AMQ). This demonstration uses features of Ajax to show multiple interactive components on the same Web page. The components in this case are a live portfolio and chat room.

1. Introduction

This demo uses the example of a Stock Portfolio to demonstrate the use of Ajax and Apache ActiveMQ and to show how to integrate them.

Ajax (short for Asynchronous JavaScript and XML) is an approach to web application programming involving many technologies. Web applications require many time-consuming page reloads after calling the server. The purpose of Ajax is to minimize page reloads by sending only the necessary data to the server instead of sending the entire page. Inside Ajax one can use XHTML and CSS for standards-based presentation, dynamic display and interaction. The Document Object Model, XML and XSLT conduct the data exchange and manipulation. Then we use XMLHttpRequest for asynchronous data retrieval. Lastly we use JavaScript to bind everything together. The result is a Web application that provides almost instant feedback from the back-end with no downtime from page reloads without having to use proprietary technologies.

Apache ActiveMQ supports Ajax using the same basis as the REST connector for Apache ActiveMQ. The REST connector allows any Web capable device to send or receive messages over JMS. Please refer to the Introduction to the Apache ActiveMQ document for more information on Apache ActiveMQ.

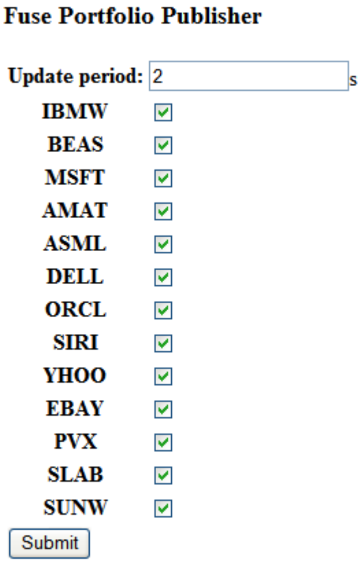
Ajax and LogicBlaze FUSE's SOA go hand in hand. In the LogicBlaze SOA model business services can be exposed through XML web services. Ajax clients are excellent consumers for these services.

2. Running the Demo

The Ajax demo is preinstalled and ready to run as part of the LogicBlaze FUSE distribution. To run the demo follow the instructions in Running the Demo below.

To run the demo:

1. Start LogicBlaze FUSE per the LogicBlaze FUSE Getting Started Guide
2. In a browser, go to `http://localhost:8080/ajax-demo`.
3. When the page opens you will notice you have two links. Click on both of them. The links will automatically open in a new window.
4. In the LogicBlaze FUSE Portfolio Publisher window you will see a list of stocks. Feel free to select or deselect as many stocks as you wish. We suggest leaving a few checked for the sake of the example.

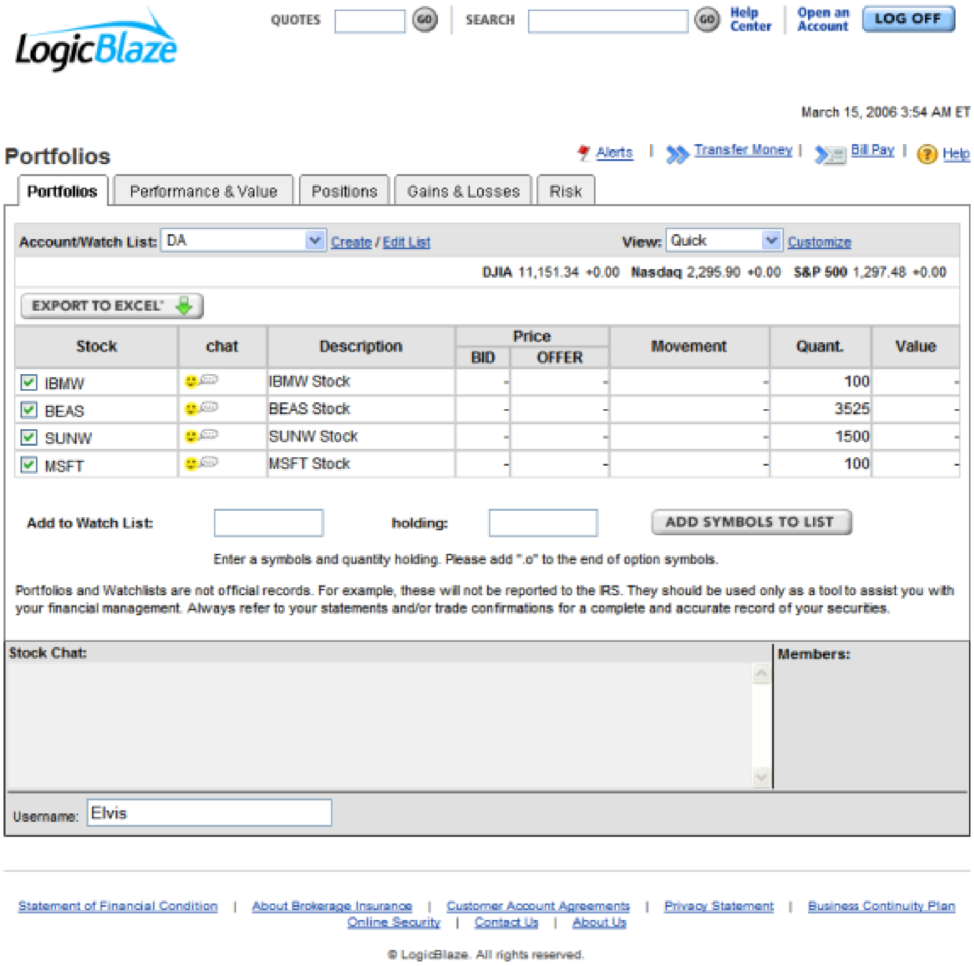


Fuse Portfolio Publisher

Update period: 2 s

IBMW	☑
BEAS	☑
MSFT	☑
AMAT	☑
ASML	☑
DELL	☑
ORCL	☑
SIRI	☑
YHOO	☑
EBAY	☑
PVX	☑
SLAB	☑
SUNW	☑

5. In the Ajax Demo window you will see your Portfolio.



March 15, 2006 3:54 AM ET

Portfolios

Account/Watch List: DA

View: Quick

DJIA 11,151.34 +0.00 Nasdaq 2,295.90 +0.00 S&P 500 1,297.48 +0.00

Stock	chat	Description	Price		Movement	Quant.	Value
			BID	OFFER			
☑ IBMW		IBMW Stock	-	-	-	100	-
☑ BEAS		BEAS Stock	-	-	-	3525	-
☑ SUNW		SUNW Stock	-	-	-	1500	-
☑ MSFT		MSFT Stock	-	-	-	100	-

Add to Watch List: holding:

Enter a symbols and quantity holding. Please add ".o" to the end of option symbols.

Portfolios and Watchlists are not official records. For example, these will not be reported to the IRS. They should be used only as a tool to assist you with your financial management. Always refer to your statements and/or trade confirmations for a complete and accurate record of your securities.

Stock Chat:

Members:

Username: Elvis

Statement of Financial Condition | About Brokerage Insurance | Customer Account Agreements | Privacy Statement | Business Continuity Plan
Online Security | Contact Us | About Us

© LogicBlaze. All rights reserved.

Portfolio

- To add a stock to your portfolio, type in any stock name (names are case sensitive) listed in your Portfolio Publisher window into the "Add to Watch List" field. Type in a number to the"holding" field. Click "Add Symbols to List" to add the information from the field to your Portfolio.
- To subtract a stock from your portfolio simply uncheck the check box from next to the stock's symbol.

Stock Chat

- Type in a user name in the `Username` field. "Elvis" has been pre-entered for your convenience.
- To join a chat room for a stock click on the face icon in the chat column that is associated to the stock that you wish to chat about. Your name will appear in the “Members” column. You will receive an error message if you fail to add a username.
- Type in a message in the “Chat” field and click "Send" to send message. You will receive an error message if you fail to type a message.
- Click "Leave" to leave room.

6. Now that you have your Portfolio set up click the "Submit" button in your "Portfolio Publisher" window.
7. Watch your portfolio change based the rate established by your "Update Period" field.
8. Navigate into the LogicBlaze FUSE Console and scroll down to the "Topics" Portal to see details. For instructions on the console please see the LogicBlaze FUSE Console Guide.

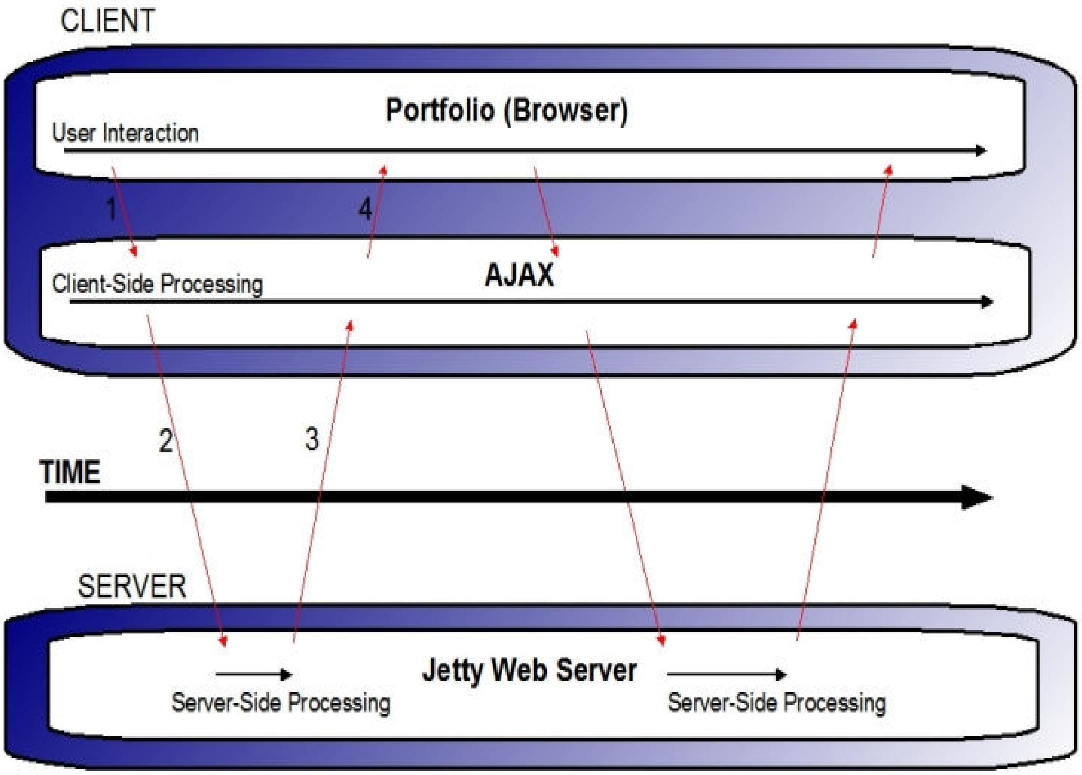
The other links located on your Portfolio page are there to show you what you can do. They are not functioning links.

2.1. How the Ajax Demo Works

This section traces the Ajax demonstration emphasizing the value of the architecture used.

The portfolio - time emphasis

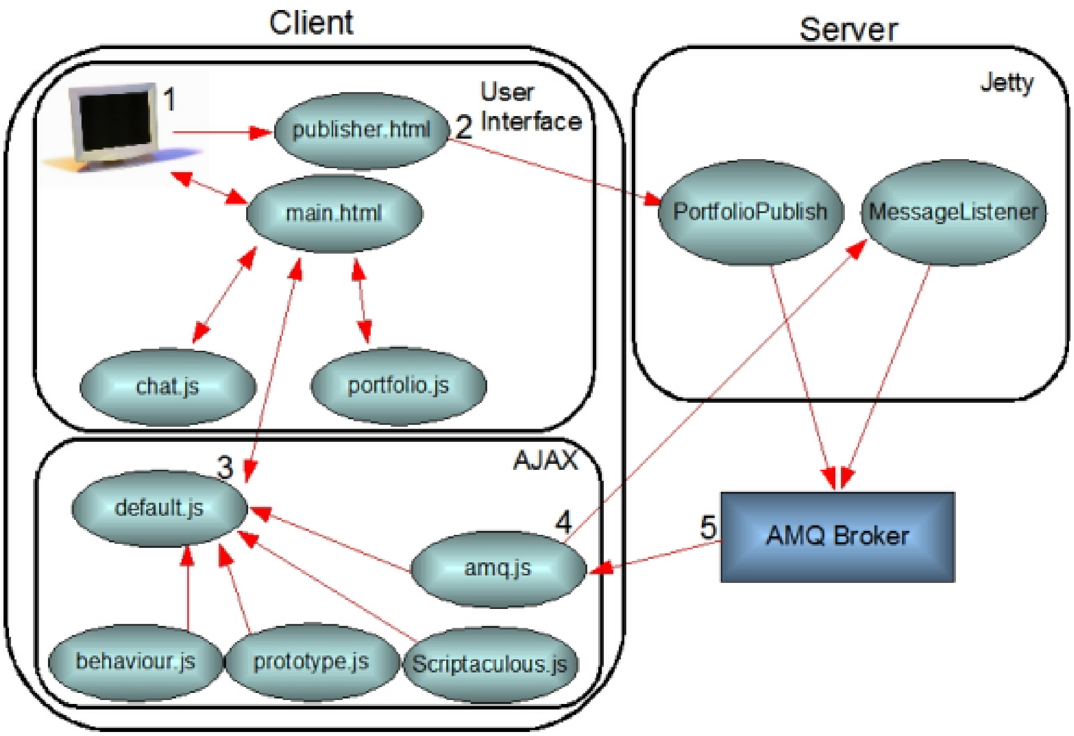
This demo will trace the path of a message in accordance to the figure. After reading this section you will better understand the principals of Ajax and messaging in Apache ActiveMQ.



1. When a user joins a room inside the portfolio, a message is sent from the user's interface to the Ajax engine. Please visit the Sending Messages section for more information.
2. The Ajax engine takes the JavaScript call and makes the appropriate HTTP requests. The HTTP transport takes the request to the Jetty web server. Please refer to the *MessageListenerServlet* portion of this document for more information.
3. The Jetty Web server sends the information requested back to the Ajax engine in the form of XML. Please visit the Receiving Messages and Polls section for more information.
4. The XML is sent back to the user interface in the form of HTML data for the user's browser to display the results of the call made in step one.

2.2. The Portfolio – Messaging Emphasis

This section is a detailed look at the stock portfolio and how it messages.



1. Open `http://localhost:8080/Ajax-demo` and open the two links `main.html` (the portfolio) and `publisher.html` (the publisher). In the `main.html` file notice there is a chat section and a stock table. For data to appear in the stock table or messages to appear in the chat window messages must be sent and received.
2. Inside `publisher.html` select the stocks that match the ones listed in our portfolio's stock table by checking the appropriate check boxes. By clicking "Submit" the `PortfolioPublisherServlet` is called. This file is a Java class that creates numbers for the checked stocks the data using random number generation functionality.

   The location of the `PortfolioPublisherServlet` is mapped by the `web.xml` located in the WEB-INF directory. The `web.xml` file contains the mapping of servlets.
3. The Ajax features are provided on the client side by the `amq.js` script. This script depends on the `behaviour.js` and `prototype.js` script. All of these scripts are loaded to the `main.html` by the `default.js` as follows.

```
<script type="text/javascript" src="/context/js/default.js"></script>
```

Including these scripts results in the creation of a JavaScript object called AMQ, which provides the API the ability to send messages and to subscribe to channels and topics.

4. As the user interacts with the `main.html` requests are created and take the form of JavaScript that calls on the Ajax layer. An example of this is sending messages when joining or leaving a room. The following from the `chat.js` file shows this:

    ```
    amq.sendMessage(room._chatMembership, "<message type='join' from='" +
    room._username + "'/>");
    ```

    and

    ```
    amq.sendMessage(room._chatMembership, "<message type='leave' from='" +
    room._username + "'/>");
    ```

    The destination for these messages is room: `_chatMembership` and the message is to join or leave the room.

    These two functions are available to us through the functionality of the Ajax layer. The calls go out to the `messageListenerServlet`. From the `messageListenerServlet` they go to the AMQ Message Broker.

    See *Sending a Message* and *Message Listener Servlet* sections for more information.

5. Messages come back from the AMQ Message Broker and the JavaScript in `chat.js` and `portfolio.js` are used to display back to the user the contents of the messages.

    For example, to add a row to the stock table or join a chat room you have to register a listener to be able to receive the information that is being sent out by others.

    This following line of code in `portfolio.js` adds a listener for each new stock row a user creates:

    ```
    amq.addListener('stocks','topic://STOCKS.' + stock, portfolio._quote);
    ```

    This following line of code in `chat.js` checks for new messages in the chat:

    ```
    amq.addListener('chat',room._chatMembership,room._chat);
    ```

    When you register listeners you must send a message subscription request to the server as a POST like any other message but as type listen. To learn more about receiving messages please refer to the *Receiving Messages and Polls* section of this document.

2.3. How to Message in LogicBlaze FUSE

This section will cover the information you need to know to successfully message in LogicBlaze FUSE using the Apache AcitveMQ messaging capabilities.

Message Listener Servlet

The Ajax features of AMQ are handled on the server side by the `MessageListenerServlet`. This servlet is responsible for tracking the existing clients (using a `HttpSession`) and for creating the AMQ and `javax.jms` objects required by the client to send and receive messages (for example, `Destination`, `MessageConsumer`, `MessageAvailableListener`). This servlet should be mapped to `/AMQ/*` in the Web application context serving the Ajax client (this can be changed, but the client JavaScript `amq.uri` field needs to be updated to match.)

```
<servlet>
  <servlet-name>MessageListenerServlet</servlet-name>
  <servlet-class>org.apache.activemq.web.MessageListenerServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

Sending Messages

To send a JMS message from the JavaScript client is done by calling the method:

```
amq.sendMessage(myDestination,myMessage);
```

where `myDestination` is the URL string address of the destination (for example, `topic://MY.NAME` or `channel://MY.NAME`) and `myMessage` is any well formed XML or plain text encoded as XML content.

When a message is sent from the client it is encoded as the content of a `POST` request, using the prototype API for XmlHttpRequest. The `AMQ` object may combine several `sendMessage` calls into a single POST if it can do so without adding additional delays (see “polls” below).

When the `MessageListenerServlet` receives a `POST`, the messages are decoded as `application/x-www-form-urlencoded` parameters with their type (in this case *send as* opposed to *listen* or *unlisten* - see below) and destination. If a destination channel or topic do not exist, it is created. The message is sent to the destination as a `TextMessage`.

Receiving Messages

To receive messages, the client must define a message handling function and register it with the AMQ object. For example:

```
var myHandler =
{
 rcvMessage: function(message)
 {
  alert("received "+message);
 }
};
amq.addListener(myId,myDestination,myHandler.rcvMessage);
```

where `myId` is a string identifier that can be used for a later call to `amq.removeHandler(myId)` and `myDestination` is a URL string address of the destination (e.g. `topic://MY.NAME` or `channel://MY.NAME`). When a message is received, a call back to the `myHandler.rcvMessage` function passes the message to your handling code.

When a client registers a listener, a message subscription request is sent from the client to the server in a POST in the same way as a message, but with a type of listen. When the `MessageListenerServlet` receives a listen message, it creates a `MessageAvailableConsumer` and registers a Listener on it.

Polls

When a Listener created by the `MessageListenerServlet` is called to indicate that a message is available, due to the limitations of the HTTP client-server model, it is not possible to send that message directly to the Ajax client. Instead the client must perform a special type of Poll for messages. Polling normally means periodically making a request to see if there are messages available and there is a trade off: either the poll frequency is high and excessive load is generated when the system is idle; or the frequency is low and the latency for detecting new messages is high.

To avoid the load vs. latency trade-off, AMQ uses a waiting poll mechanism. As soon as the amq.js script is loaded, the client begins polling the server for available messages. A poll request can be sent as a GET request or as a POST if there are other messages ready to be delivered from the client to the server. When the `MessageListenerServlet` receives a poll it:

1. If the poll request is a POST, all send, listen and unlisten messages are processed
2. If there are no messages available for the client on any of the subscribed channels or topic, the servlet suspends the request handling until:
   - A `MessageAvailableConsumer` Listener is called to indicate that a message is now available; or
   - A timeout expires (normally around 30 seconds, which is less than all common TCP/IP, proxy and browser timeouts).
3. A HTTP response is returned to the client containing all available messages encapsulated as `text/xml`.

When the `amq.js` JavaScript receives the response to the poll, it processes all the messages by passing them to the registered handler functions. Once it has processed all the messages, it immediately sends another poll to the server.

Thus the idle state of the AMQ Ajax feature is a poll request "parked" in the server, waiting for messages to be sent to the client. Periodically this "parked" request is refreshed by a timeout that prevents any TCP/IP, proxy or browser timeout closing the connection. The server is thus able to asynchronously send a message to the client by waking up the "parked" request and allowing the response to be sent.

The client is able to asynchronously send a message to the server by creating (or using an existing) second connection to the server. However, during the processing of the poll response, normal client message sending is suspended, so that all messages to be sent are queued and sent as a single POST with the poll that will be sent (with no delay) at the end of the processing. This ensures that only two connections are required between client and server (the normal for most browsers).

Threadless Waiting

The waiting poll described above is implemented using the Jetty 6 Continuation mechanism. This allows the thread associated with the request to be released during the wait, so that the container does not need to have a thread per client (which may be a large number). If another servlet container is used, the Continuation mechanism falls back to use a wait and the thread is not released.

3. Additional Resources

ActiveMQ Web site: http://activemq.org

Article explaining Ajax: http://www.adaptivepath.com/publications/essays/archives/000385.php

Article about Ajax and SOA: http://www.xml.com/lpt/a/2005/10/05/ajax-web-20-soa.html

Web sites on Ajax: http://www.ajaxinfo.com, http://www.ajaxian.com