Creating a SmartClient application using Atmosphere and Jersey

Description

SmartClient already has a real-time streaming technology that we call SmartClient Real-Time Messaging, but there are other frameworks/technologies that you can use to achieve the same goal. One of such frameworks is the Atmosphere real-time streaming framework

https://github.com/Atmosphere/atmosphere

We have taken the Atmosphere real-time streaming framework to demonstrate integrating it with a SmartClient grid. For that, we have modified the sample code found here, and have replicated using Atmosphere:

http://www.smartclient.com/#FSportfolioGrid

As part of the example we have developed also a small javascript class, AtmosSocket, that is a wrapper around the Atmosphere API that simplifies the access.

The application has been deployed and tested on glassfish 3.1.2, Tomcat 6 and Google Plugin for Eclipse development server.

Setting up

Setting up libraries

First step is to get the required java libraries for the project.

We will need:

1. atmosphere libraries (runtime, jersey, annotations version 2.1.0)
2. jersey libraries (core, server, servlet version 1.17.1)
3. jackson (core, jaxrs, mapper version 1.9.13)
4. slf4j library (version 1.6.1)
5. asm library (version 3.1)

We have decided to deploy all the required libraries together with our application, so we do not need to do any special setup of the application servers. Take into account that in a production environment you will probably prefer to deploy some of the required libraries into the application server's common library directory.

We also will need the atmosphere javascript libraries, atmosphere.js (version 2.1.4).

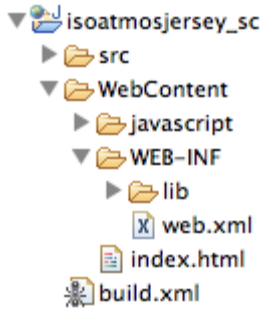
These are the sources for all libraries, in case you are not using Maven:

Java libraries	All the required libraries for release 2.1.0 can be found in Maven repository for atmosphere
jquery-atmosphere js library	The version 2.1.4 of this library can be downloaded here.

Also you will find scatmos.js, that contains AtmosSocket, a wrapper class around the Atmosphere API that provides easier access to this API.

Project structure

If you uncompress the contents of the provided zip, you will find the following folder structure:



The `src` folder contains the required source code for the server part (the jersey resource).
The `WebContent` folder contains at the top level the index.html file, with the sample application embedded.
The `WebContent/javascript` directory contains both the atmosphere library, our new wrapper javascript class `AtmosSocket` and the sample data for the demo.
The `WebContent/WEB-INF/lib` directory contains all the required java libraries for the server part to work.
Before you can run the sample the `WebContent` folder must be populated with a copy of the isomorphic libraries. Just copy the `isomorphic` folder from your smartclientRuntime into `WebContent/isomorphic`.
After you have copied the isomorphic libraries into the require folder, you can compile the project using eclipse, or the ant `build.xml` file provided. You will obtain a war file that you can deploy on your servlet container or application server.

The Atmosphere Smartclient API (wrapper)

As explained before, we have encapsulated the access to Atmosphere into a wrapper class, AtmosSocket, that eases the access to the Atmosphere API from SmartClient.

This API provides the new AtmosSocket class with this class method to create a new socket and subscribe to it:

`subscribe(service, channelId, messageHandler, subscribedHandler)`	This will create a new AtmosSocket object and will subscribe to it (see properties below to see details on each parameter). Once subscribed, each time the client receives a message sent from the server to our client, the atmosphere framework will call the “onMessage” callback method, that we registered when we configured the connection before subscribing.

This API also provides these three instance methods:

`subscribe()`	You can use this method to subscribe to an already created AtmosSocket object.
`pushData(data)`	The data field is a JSON Object, it may contain whatever we want to send to the server encapsulated as a json object.
`unsubscribe()`	To stop receiving messages from Atmosphere.

The class also has these properties:

`context`	The context set when we configured the AtmosphereServlet in web.xml deployment descriptor, as "url-pattern"
`service`	The identifier of the Jersey service as set with the @Path annotation
`channelId`	The identifier of the channel to use. Can be a generic identifier, of we want to be notified of all messages sent by the server, or a specific one, so we will only receive messages addressed to us.
`transport`	Transport protocol. We use streaming (WebSocket) by default.

`fallbackTransport`	In case the selected transport protocol is not available in both sides (client and server), the framework will negotiate to find the next common protocol. This is our preferred protocol in case the first one is not available. By default is long-polling.
`messageHandler`	function that will be called when a new message arrives.
`subscribedHandler`	function that will be called when the subscription has been done (before that, we should not use the pushData() function, or it will fail).

Using the client API

When a Smartclient application needs to connect to the server using an Atmosphere service, the first step is to subscribe the application to the service. For that, we must create a new AtmosSocket object, providing the service name, a channelId for this channel, a handler function for the received messages and a handler function for the “subscribed” event.
The `service` property is a String that identifies the specific Jersey service among all the services that a Jersey server may offer. In our case we have only implemented one service, the “quotes” service.
The `channelId` is a String. All the clients with the same channelId will receive a copy of any single message broadcasted by the server to that channel, so if we want to receive server events that can only be seen by our client, we must provide a unique `channelId` for each client.
Together with the `channelId`, we could also setup the transport and fallback-transport methods, by default set to “websocket” and “long-polling” (the fallback transport method is the preferred method to use in case the server does not support websocket. If you are deploying on glassfish, that supports natively websockets, the demo application will use websockets as transport).
The `messageHandler(receivedData)` function is going to be called by the framework each time a server message is received through the channel in the client. Take into account that the received message will be a json representation of the java object that you defined in the server side, we will discuss further on this topic later when describing the *Jersey rersource*.
The `subscribedHandler()` function is going to be called by the framework once the connection of the socket has been established. We should not send any data to the server over the socket until this method has been called back, or we will get errors.

```
	// Prepare to start reveiving messages from the stockQuotes channel to update data grid
	// As channelId we will use a random number, to assure that a unique channel
	// is stablished between this specific client and the server
	var channelId = Math.random() * isc.timestamp();
	var atmSocket = isc.AtmosSocket.create({
		service: "quotes",
		channelId:channelId,
		messageHandler: updateStockQuotes,
		subscribedHandler: hasSubscribed
	});
	// Request registration. When completed, the subscribedHandler function will be invoked.
	atmSocket.subscribe();
```

This is an example of a `messageHandler` function:

```
	function updateStockQuotes(recData) {
		// The response is a json object
		var response = JSON.parse(recData.responseBody);
		// Now we can access the properties of the response
		var data = response.stockData;

		...
		// use the incoming data to refresh a clientOnly datasource’s data
		for (i = 0; i < data.size(); i++) {
			...
		}
	}
```

This is an example of a `subscribedHandler` function:

```
	function hasSubscribed() {
		generateUpdatesButton.click(); // Once subscribed, ask the server to start sending back data
	}
```

Server side: the Jersey resource

Let's start our sample application by designing the server side. We are going to do a new version of the existing example FSPortfolioGrid. In that example, the client side communicates via "Real-Time Messaging" with a server application that generates random changes for a list of stock quotes. What we are going to do for this example is to create a new application that will do the same, but using Atmosphere to communicate with the server. Our application will subscribe to certain service (identified as "`jersey/rpc`"), will get ready to receive messages from the service and then will send a message to the server requesting the service to start sending updates back to the client.

When using Atmosphere, the server side can be implemented in a number of different ways, in this case we will use Jersey. To implement a new Jersey service we only need to implement a class, with a couple of methods: one to process the subscription request to the service and another one to process all the messages that come from the client.

The `JerseyRpc` class represents our Jersey service. In order to respond ONLY to the clients that subscribed to this channel, when a client subscribes to the service the client sends a channelId identifying itself. This identifier arrives as part of the url used to access the jersey resource during the registration. So we must configure our Jersey resource this way:

```
@Path("/rpc/{channelid}")
public class JerseyRpc {
        private
        @PathParam("channelid")
        Broadcaster channel;
        …
}
```

Now we must implement two methods in this class. The first method that we need to implement is the one that accepts subscriptions (with the @GET annotation, as the atmosphere "`socket.subscribe()`" method sends a GET request).

```
        @GET
        @Suspend
        @Produces("application/json")
        public String suspendGet() {
                return null;
        }
```

As you can see, there is not much to do, as the subscription is solved in fact by the framework, but our service must implement this method anyway.

The second method is more interesting. We use it to respond to the messages sent from the client. In our case, the client is going to send a message with the text "start" to indicate that it is ready to accept a new burst of data. So, when the service receives a message from client, it will check if it is the "start" command, and if it is correct, the service will answer starting a new thread that will send a burst of messages, 2 per second during 90 seconds, to the client.

```
        @POST
        @Suspend
        @Broadcast(writeEntity = false)
        @Produces("application/json")
        public void broadcast(String message) {
                if(message.equals("start")) {
                        System.out.println("Start sending data to channel " + channel.getID());
                        sendResponses(channel.getID());
                } else {
                        System.out.println("Invalid command " + message +
                                        " received in channel " + channel.getID());
                }
        }
```

The `sendResponses(channelId)` method just starts a new Thread that will loop during 90 seconds, generating lists of random values and sending them to the channel identified by `channelId`.
To send the data, first we obtain an instance of a Broadcaster that will send data to any application subscribed with the

“`channelId`” with this code:

```
        Broadcaster channel = BroadcasterFactory.getDefault().lookup(channelId);
```

Then, we generate the data to encapsulate in the message. The data consists of just a List of pairs of numbers, the first being an Integer and the second a Float, so we encapsulate them in a List<Object[]>:

```
        List<Object[]> stockData = new ArrayList<Object[]>();
```

Next we instantiate a new Json response with the generated data.

```
        JsonResponse response = new JsonResponse(stockData);
```

Here is where we use Jackson to convert our Java List into an object that can be sent over the Atmosphere channel. For that we only needed to create a new POJO class to hold the data in one field, in this case the field “stockData” (it is mandatory that the class has a default constructor):

```
public class JsonResponse {
        public List<Object[]> stockData;

        public JsonResponse() {
        }

        public JsonResponse(List<Object[]> stockData) {
                this.stockData = stockData;
        }
}
```

To have the instances of this class automatically translated to json, we only need to declare the package that may contain classes that need json conversion, and Jersey will do the conversion using jackson. This is done in the deployment descriptor web.xml as we will see later.
Finally, to send the json data over the channel to the client, we only need to do this:

```
        channel.broadcast(response);
```

Changes in the client application

The original application can be found in the Feature Explorer: http://www.smartclient.com/#FSportfolioGrid
The main changes that we had to apply to the origianl application were these:
First, beside the isomorphic libraries, we added to the index.html file both the atmosphere library and our wrapper:

```
</head>
        ...
        <script src="javascript/atmosphere.js"></script>
        <script src="javascript/scatmos.js"></script>
</head>
```

Next we add the script that loads the datasource and initial data for the grid:

```
<body>
        <script src="javascript/data/stockQuotes.js"></script>
        ...
```

Then we remove the initial click on the `generateUpdatesButton`, as we will wait until we have received the callback that notifies us that the channel is ready for use:

```
        isc.Button.create({
                ...
        })~~.click()~~;
        ...
        function hasSubscribed() {
                generateUpdatesButton.click();
        }
```

When the button is clicked now we push a message to the channel to request the server to start sending data:

```
~~isc.RPCManager.sendRequest({ actionURL: "/examples/StockQuotes/generate?sp="+startParameter});~~
```

```
atmSocket.pushData("start");
```

In `updateStockQuotes` we do two changes. The first one is the way we unsubscribe from the channel:

```
function updateStockQuotes(recData) {
        ...
        isc.Messaging.unsubscribe("stockQuotes"+startParameter);

        atmSocket.unsubscribe();
        ...
```

The second change is an extra step needed to transform the received data from json to javascript:

```
        var response = JSON.parse(recData.responseBody);
        var data = response.stockData;
```

Finally, we change the way we subscribe to the messaging service:

```
        isc.Messaging.subscribe("stockQuotes"+startParameter, updateStockQuotes);

        var atmSocket = isc.AtmosSocket.subscribe("quotes", startParameter, updateStockQuotes, hasSubscribed);
```

Other configuration issues

We only need to take into account one configuration file: web.xml. In that file we only need to configure the AtmosphereServlet, that is the module that connects the Atmosphere framework with our jersey service.
As we discussed before, Jersey must know what packages may contain objects that need JSON serialization, so we declare our "`com.isomorphic.examples.atmosphere.server`" package in the init-param "`com.sun.jersey.config.property.packages`", as you can see below.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" version="2.5">
  <servlet>
    <description>AtmosphereServlet</description>
    <servlet-name>AtmosphereServlet</servlet-name>
    <servlet-class>org.atmosphere.cpr.AtmosphereServlet</servlet-class>
    <init-param>
      <param-name>com.sun.jersey.config.property.packages</param-name>
      <param-value>com.isomorphic.examples.atmosphere.server, org.codehaus.jackson.jaxrs</param-value>
    </init-param>
    <init-param>
      <param-name>org.atmosphere.websocket.messageContentType</param-name>
      <param-value>application/json</param-value>
    </init-param>
    <load-on-startup>0</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>AtmosphereServlet</servlet-name>
    <url-pattern>/jersey/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Testing

To test the application, package and deploy it to your preferred application server. In case you want to test it with glassfish, you only need to execute the ant build.xml file provided, then zip the "war" folder into a war file, for instance sampleatmos_sc.war. Once you have the .war file, deploy it into glassfish as usual, then visit the application's url, for instance http://localhost:8080/sampleatmos_sc/
You should see a screen like this, with stock quotes changing dynamically:

Name	Symbol	Last	Change	Open	DayHigh	DayLow
Electronic Arts Inc.	ERTS	17.81	-0.04	17.52	17.92	17.78
Intel Corporation	INTC	21.73	-0.09	21.50	21.82	21.42
Broadcom Corporation	BRCM	42.22	-0.08	45.35	45.35	42.12
Microsoft Corporation	MSFT	27.91	0.01	27.98	28.08	27.86
Cisco Systems, Inc.	CSCO	21.64	0.06	21.17	21.67	21.02
NVIDIA Corporation	NVDA	25.44	-0.06	25.99	25.99	25.12
Micron Technology, Inc.	MU	11.03	-0.00	10.80	11.05	10.69
Applied Materials, Inc.	AMAT	16.36	-0.05	16.22	16.48	16.32
Oracle Corporation	ORCL	33.20	-0.07	33.40	33.38	33.04
Dell Inc.	DELL	13.73	0.02	13.45	13.77	13.41

Generate more updates