Reverse Ajax, Part 4: Atmosphere and CometD

Learn to use the best open source Reverse Ajax libraries for Java technology servers

Mathieu Carbou

This series explores how to develop event-driven web applications using Reverse Ajax techniques. Part 1 introduced different ways to implement a Reverse Ajax communication. Part 2 explained how to implement Reverse Ajax using WebSockets, and discussed the limitations of web servers using Comet and WebSockets. Part 3 explored the difficulties of implementing your own Comet or WebSockets communication system if you need to support several servers or provide an independent Web application that users can deploy on their own server. It also introduced Socket.IO. In this article, learn about Atmosphere and CometD—the best-known open source Reverse Ajax libraries for Java technology servers.

View more content in this series

Introduction

This series shows you how to develop event-driven web applications using Reverse Ajax techniques. Part 1 introduced Reverse Ajax, polling, streaming, Comet, and long polling. Part 2 explained how to implement Reverse Ajax using WebSockets and discussed the limitations of web servers using Comet and WebSockets. Part 3 showed that implementing your own Comet or WebSockets communication system can be difficult if you need to support several servers or provide an independent web application for users to deploy on their own server. Even if the JavaScript code on the client side is simple, you would need some exception handling, reconnection, and acknowledgement features. On the server side, the lack of a global API and the multiple web server APIs lead to the need for frameworks, which brings an abstraction. Part 3 also discussed Socket.IO.

In this article, learn about Atmosphere and CometD. They are the most widely known open source Reverse Ajax libraries for Java servers.

You can download the source code used in this article.
Prerequisites
Ideally, to get the most out of this article, you should know JavaScript and Java. To run the sample in the article, you will also need the latest version of Maven and the JDK (see Related topics).

The Atmosphere framework
Atmosphere is a Java technology framework that provides a common API for using the Comet and WebSocket features of many of the web servers, including Tomcat, Jetty, GlassFish, Weblogic, Grizzly, JBossWeb, JBoss, and Resin. Any web server supporting the Servlet 3.0 Specification is also supported. Of the frameworks presented in this series of articles, Atmosphere supports the most servers.

Atmosphere can detect native server APIs (for Comet and WebSockets) and switch back to Servlet 3.0, if available, for Comet. Or, also for Comet, it will fall back to a "managed" asynchronous mode (but not as scalable as Jetty Continuations). Atmosphere has been around for more than two years, and it is still in active development. It is used in large web applications such as JIRA, which is one of the best known issue trackers. Figure 1 shows the Atmosphere architecture.

Figure 1. Architectural view of Atmosphere

The Atmosphere framework is composed of the Atmosphere runtime, which provides a common API for all different web server solutions and standards. On top of this, the client can access the API and Reverse Ajax features through the Google Web Toolkit (GWT) by setting up a mere servlet. Or, you can also use Jersey, a framework implementing the JSR-311 (JAX-RS specification). Thus, Atmosphere can be used in restful services with additional annotations provided. After configuring your chosen module, you can then access the Atmosphere runtime by implementing some classes (discussed later in this article). You can also optionally use some provided plugins that add support for clustering, messaging, dependency injection, and more. If you're using a web framework (Wicket, Struts, Spring MVC), you can transparently add Reverse Ajax support by using the MeteorServlet of Atmosphere. This servlet exposes a Meteor object that can be retrieved within your controllers and services to suspend or resume the requests.

Atmosphere's strength remains on the server side: it provides a standardized API that covers all of the different solutions and ways to communicate with WebSockets or Comet. Atmosphere does not use a protocol between the client and server, like Socket.IO and CometD. Both libraries provide
a client-side JavaScript and a server-side servlet that communicate using a specific protocol (for handshaking, messaging, acknowledgment, and heartbeat). Atmosphere's goal is to provide a common communication channel on the server side. If you need to use a specific protocol, such as Bayeux (a protocol used by CometD), you have to develop your own “handler” in Atmosphere. The CometD plugin does just that: it takes advantage of Atmosphere's API to suspend and resume the requests, and it delegates to CometD classes to manage the CometD communication using the Bayeux protocol.

Atmosphere comes with a jQuery client library to facilitate the connection setup, which is able to automatically detect the best transport available (WebSockets or CometD). Usage of Atmosphere's jQuery plugin is similar to the HTML5 WebSockets API. First, you connect to the server, register a callback to receive messages, and then you can push some data.

The source code included with this article contains an Atmosphere sample that directly uses a handler with the Atmosphere servlet. The client code is always the same; it's the same code used in Parts 1, 2, and 3 of this series (in the chat samples using Comet long polling). You could have used Atmosphere's jQuery plugin, but it's not required because Atmosphere does not enforce any communication protocol. It is strongly suggested that you look at the other examples in Atmosphere's project, especially those using JSR-311 annotations (Jersey). They really simplify the writing of handlers.

Listing 1 shows the Atmosphere handler interface.

**Listing 1. AtmosphereHandler interface**

```java
public interface AtmosphereHandler<F, G> {
    void onRequest(AtmosphereResource<F, G> resource)
        throws IOException;
    void onStateChange(AtmosphereResourceEvent<F, G> event)
        throws IOException;
    void destroy();
}
```

The onRequest method receives all requests from the client and decides whether to suspend or resume them (or do nothing). Each time a request is suspended or resumed, a broadcast is sent, or a timeout occurs, an event is sent and received by the onStateChange method.

The implementation of the onRequest method for the Comet chat is shown in Listing 2.
Reverse Ajax, Part 4: Atmosphere and CometD

Listing 2. AtmosphereHandler interface - onRequest

```java
Broadcaster broadcaster = 
    BroadcasterFactory.getDefault().lookup(
        DefaultBroadcaster.class, ChatHandler.class.getName(), true);
broadcaster.setScope(Broadcaster.SCOPE.APPLICATION);
resource.setBroadcaster(broadcaster);
HttpServletRequest req = resource.getRequest();
String user = (String) req.getSession().getAttribute("user");
if (user != null) {
    if ("GET".equals(req.getMethod())) {
        resource.suspend(-1, false);
    } else if ("POST".equals(req.getMethod())) {
        String cmd = req.getParameter("cmd");
        String message = req.getParameter("message");
        if ("disconnect".equals(cmd)) {
            close(resource);
        } else if (message != null && message.trim().length() > 0) {
            broadcaster.broadcast("[" + user + "] " + message);
        }
    }
}
```

A typical convention is to suspend GET requests and send messages using POST requests. When a message is received, it is broadcast to all resources registered within the broadcaster. Notice that the example never writes anything to the HttpServlet output stream. The broadcast or suspend action just sends events received by the other method implemented, as shown in Listing 3:

Listing 3. AtmosphereHandler interface - onStateChange

```java
Broadcaster broadcaster = 
    BroadcasterFactory.getDefault().lookup(
        DefaultBroadcaster.class, ChatHandler.class.getName(), true);
// Client closed the connection.
if (event.isCancelled()) {
    close(event.getResource());
    return;
}
try {
    String message = (String) event.getMessage();
    if (message != null) {
        PrintWriter writer = 
            event.getResource().getResponse().getWriter();
        writer.write(message);
        writer.flush();
    }
} finally {
    if (!event.isResumedOnTimeout()) {
        event.getResource().resume();
    }
}
```

And now you have all you need for a working Comet chat. To summarize, the important Atmosphere concepts are: the resource object representing the connection, and a broadcaster responsible for firing events to resources and deciding when to suspend or resume a request. Note that the example is Comet only. To be able to use WebSockets and Comet, a client-side library should be used and a more complex handler would be needed.

Table 1 outlines the pros and cons of using the Atmosphere framework.
### Table 1. Pros and cons of Atmosphere

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>If you have to deploy a web application in several web servers that you can't control. You'll have a better chance that the Reverse Ajax features of your application work correctly due to the number of web servers supported by Atmosphere.</td>
<td>Lack of documentation about Atmosphere's architecture, project, concepts, and APIs, which is helpful if you need to go into the source code or analyze several provided samples. The API is highly technical, and sometimes obscure, compared to the simple APIs of other frameworks, such as Socket.IO and CometD. Even when using Atmosphere annotations, some names and attributes were too technical.</td>
</tr>
<tr>
<td>When you need a common API over a raw Reverse Ajax communication, without any protocol defined, because you want to develop or extend it.</td>
<td>Though a good abstraction on the server side, there is not a good client library. There is no protocol, so all additional features are left to the developer. The current library is too simple for the needs of a big, scalable web application if you require advanced timeout detection, acknowledgement, back-off, cross-domain and so on, especially when working with mobile devices. In this case, CometD is much more reliable; it leverages a communication protocol that can be used to activate some control flow and error detection, all provided within CometD. Using the CometD JavaScript client with the Atmosphere CometD plugin could be a good alternative if you need extra features.</td>
</tr>
</tbody>
</table>

### The CometD framework

The CometD framework, an event-driven communication solution based on HTTP, has been around for several years. Version 2 added support for annotation configuration and WebSockets. The CometD framework provides a Java server part and a Java client part, plus JavaScript client libraries based upon jQuery and Dojo. CometD uses a standardized communication protocol called Bayeux, allowing you to activate some extensions for message acknowledgement, flow control, synchronization, clustering, and so on.

CometD's event-driven approach fits very well with the new concept of event-driven web development. As with traditional desktop user interfaces, all of the components communicate through a bus to send notifications and receive events. All communication is therefore asynchronous.

The CometD framework:

- Is well-documented.
- Provides samples and Maven archetypes to facilitate the startup of a project.
- Has a well-designed API that enables extension development.
- Provides a clustering module, called Oort, that gives you the ability to run multiple CometD web servers as nodes in a cluster behind a load balancer to scale to a larger number of HTTP connections.
- Supports security policies to allow a fine-grain configuration of who can send messages over which channel.
- Integrates quite well with Spring and Google Guice (dependency injection frameworks).

### Bayeux protocol

The Bayeux communication protocol is primarily over HTTP. It asynchronously provides a responsive bi-directional communication between the client and server. The Bayeux protocol is based on channels where messages are routed and delivered client to server, server to client, or
client to client (but through the server). Bayeux is a type of publish-subscribe protocol. CometD implements the Bayeux protocol, and thus provides an abstraction layer on top of Comet and WebSocket transports to route requests through Bayeux.

**Servers and internals**

CometD is bundled with three transports: JSON, JSONP, and WebSocket. They depend upon Jetty Continuations and the Jetty WebSocket API. By default, CometD can be used in Jetty 6, 7, and 8, as well as any other servers supporting the Servlet 3.0 Specification. Transports can be added and developed in the same way as extensions. You should be able to write transports supporting the Grizzly WebSocket API and others, and then add them at the step where the CometD server is configured. Figure 2 shows an overview of the main CometD blocks.

**Figure 2. Architectural view of CometD**

The security layer to access message channels is not shown in Figure 2.

The **source code** provided with this article includes a web application using CometD. The web application's descriptor contains the definition in Listing 4 for the chat example.

**Listing 4. web.xml**

```xml
<servlet>
    <servlet-name>cometd</servlet-name>
    <servlet-class>
        org.cometd.java.annotation.AnnotationCometdServlet
    </servlet-class>
    <async-supported>true</async-supported>
    <init-param>
        <param-name>services</param-name>
        <param-value>ChatService</param-value>
    </init-param>
    <init-param>
        <param-name>transports</param-name>
        <param-value>
            com.ovea.cometd.websocket.jetty8.Jetty8WebSocketTransport
        </param-value>
    </init-param>
</servlet>
```
The CometD servlet supports several options controlling global settings, such as the ability to set transports and services. In the example, assume you want to add WebSocket support for Jetty 8. The CometD service class `ChatService` on the server side will control the chatroom where everyone is speaking, as shown in Listing 5:

**Listing 5. CometD ChatService**

```java
@Service
public final class ChatService {
  @Inject
  BayeuxServer server;

  @PostConstruct
  void init() {
    server.addListener(new BayeuxServer.SessionListener() {
      @Override
      public void sessionAdded(ServerSession session) {
        // ...
      }
      @Override
      public void sessionRemoved(ServerSession session, boolean timedout) {
        // ...
      }
    });
  }

  @Configure("/**")
  void any(ConfigurableServerChannel channel) {
    channel.addAuthorizer(GrantAuthorizer.GRANT_NONE);
  }

  @Configure("/chatroom")
  void configure(ConfigurableServerChannel channel) {
    channel.addAuthorizer(new Authorizer() {
      @Override
      public Result authorize( // check that the user is in session
        // ...
      });
  }

  @Listener("/chatroom")
  void appendUser(ServerSession remote, ServerMessage.Mutable message) {
    // ...
  }
}
```

Listing 5 demonstrates some of the important features of CometD, including:

- Dependency injection
- Life cycle management
- Global channel configuration
- Security management
- Message transformation (to add the user name before all messages)
- Session management
On the client side, the example did not activate any extension -- just the raw CometD code, as shown in Listing 6:

**Listing 6. CometD client code**

```javascript
// First create the cometd object and configure it
var cometd = new $.Cometd('CometD chat client');
cometd.configure({
    url: document.location + 'cometd',
    logLevel: 'debug'
});
cometd.websocketEnabled = 'WebSocket' in window;

// Then we register some listeners. Meta channels (those with // the form /meta/<name> are specific reserved channels)
cometd.addListener('/meta/disconnect', function(message) {
    [...]
});
cometd.addListener('/meta/connect', function(message) {
    [...]
});

// Then, starting a connexion can be done using:
cometd.handshake();

// And subscriptions with:
cometd.subscribe('/chatroom', function(event) {
    [...] // event.data holds the message
});

// We finally send data to the chatroom like this:
cometd.publish('/chatroom', msg);
```

CometD's client-side API is simple to use and understand, while remaining powerful and extensible. This article only covers the major parts of the web application, so check out the [example application](#) to get a better idea of the power of CometD.

Table 2 outlines the pros and cons of using the CometD framework.
Table 2. Pros and cons of CometD

| Pros                                                                 | Cons                                                                                           |
|----------------------------------------------------------------------|                                                                                                |
| CometD offers a complete solution, from client and server side, and   | CometD currently does not support any Servlet 2.5 containers other than Jetty for Comet (Tomcat), |
| also from a standalone Java client to a server. The framework is well- | and it does not support Glassfish/Grizzly WebSocket.                                              |
| documented, has a good API, and is easy to use. Best of all, it has an |                                                                                                |
| event-driven approach. CometD and Bayeux are part of many event-     |                                                                                                |
| driven web applications. Other Reverse Ajax frameworks don’t provide  |                                                                                                |
| any event-driven mechanisms, forcing end users to develop their own  |                                                                                                |
| custom solution.                                                     |                                                                                                |
| CometD supports many required features, such as reconnection,         |                                                                                                |
| reliable timeout detection, back-off, batching, message acknowledgement, and more that you won’t find in other Reverse Ajax frameworks. CometD lets you achieve the most reliable, low-latency communication possible. |                                                                                                |

Conclusion

Both Atmosphere and CometD are solid, open source Reverse Ajax solutions. At Ovea, our choice is CometD because we develop scalable event-driven web applications for mobile devices within a clustered environment, and we have total control over the infrastructure (we use Jetty). However, without additional development, CometD might not be the best choice if you are selling web applications and want your Reverse Ajax features to work on as many servers as possible. But knowing that more and more Web containers now support the Servlet Specification 3.0, CometD’s limitation tends to decrease. Speaking about the transport layer, the main remaining difference now relies on the WebSocket support.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Article source code</td>
<td>reverse_ajaxpt4_source.zip</td>
<td>28KB</td>
</tr>
</tbody>
</table>
Related topics

- Read the previous parts in this series:
  - Part 1: Introduction to Comet
  - Part 2: WebSockets
  - Part 3: Web servers and Socket.IO
- Atmosphere: Learn more about the portable AjaxPush/Comet and WebSocket Framework.
- CometD: Read all about the CometD Project @ The Dojo Foundation.
- JSR 315: Java Servlet 3.0 Specification is an update to the 2.5 specification.
- Read the IBM WebSphere Application Server V8.0 release announcement.
- "Google AppEngine uses Jetty!" describes Google's customization of Jetty.
- Read more about Jetty Continuations and features.
- See a Glassfish WebSockets Sample using Grizzly.
- WebSphere Application Server: Learn more about the flagship product within IBM's WebSphere brand.
- WebSphere 8 (with Comet support): Enables intelligent management of application environments and helps deliver rich user experiences faster.
- Socket.IO: Aims to make realtime apps possible in every browser and mobile device, blurring the differences between the different transport mechanisms.
- The Starting With WebSockets tutorial reviews the process of running a WebSocket server in PHP, and then builds a client to send and receive messages to it over the WebSocket protocol.
- "The WebSocket API" (W3C, August 2011): This specification defines an API that enables Web pages to use the WebSocket protocol for two-way communication with a remote host.
- Read more about Asynchronous processing support in Servlet 3.0.
- Philip McCarthy's weblog has more on Comet & Java: Threaded Vs Nonblocking I/O.
- The Rox Java NIO Tutorial collects the author's experiences using the Java NIO libraries and the dozens of hints, tips, suggestions and caveats that litter the Internet.
- On Wikipedia, read about:
  - AJAX
  - Reverse-Ajax
  - WebSockets
- Exploring Reverse AJAX: Provides an introduction to some Reverse-Ajax techniques.
- "Cross-domain communications with JSONP, Part 1: Combine JSONP and jQuery to quickly build powerful mashups" (developerWorks, February 2009): See how you can combine an obscure cross-domain call technique (JSONP) and a flexible JavaScript library (jQuery) to build powerful mashups surprisingly quickly.
- "Build Ajax applications with Ext JS" (developerWorks, July 2008): Get an overview of the object-oriented JavaScript design concepts behind Ext JS, and shows how to use the Ext JS framework for rich Internet application UI elements.
- "Compare JavaScript frameworks" (developerWorks, February 2010): Get an overview of the frameworks that greatly enhance JavaScript development.
- "Mastering Ajax, Part 2: Make asynchronous requests with JavaScript and Ajax" (developerWorks, January 2006): Learn how to use Ajax and the XMLHttpRequest
object to create a request/response model that never leaves users waiting for a server to respond.
- "Create Ajax applications for the mobile web" (developerWorks, March 2010): Understand how to build cross-browser smartphone Web applications using Ajax.
- "Where and when to use Ajax in your applications" (developerWorks, February 2008): See how you can use Ajax to improve your Web sites while avoiding bad user experiences.
- "Improve the performance of web 2.0 applications" (developerWorks, December 2009): Explore different browser-side cache mechanisms.
- "Introducing JSON" (JSON.org): Get an introduction to JSON syntax.
- developerWorks podcasts: Listen to interesting interviews and discussions for software developers.
- Atmosphere source code: Get Atmosphere, a POJO based framework using Inversion of Control (IoC) to bring push/Comet and Websocket to the masses.
- CometD source code: Get the project that's a scalable comet (server push) implementation for web messaging.
- CometD extensions: Get them for Jetty 8 WebSocket support and Google Guice support from Ovea.
- Jetty: Get Jetty, a web server and javax.servlet container, plus support for WebSockets.
- Grizzly NIO Framework: To help you take advantage of the Java NIO API.
- Grizzly Comet Sample: See the changes you'll need to make to existing applications before building a new one from scratch.
- Glassfish Application Server: Get the main GlassFish Server Open Source Edition.
- Socket.IO Java: Get the original project and the most updated Ovea's fork.
- Apache Maven: Get Maven, a software project management and comprehension tool.
- Java Development Kit, Version 6: Get the Java Platform, Standard Edition (Java SE), which lets you develop and deploy Java applications on desktops and servers, as well as in today's demanding embedded environments.
- Try out IBM software for free. Download a trial version, log into an online trial, work with a product in a sandbox environment, or access it through the cloud. Choose from over 100 IBM product trials.

© Copyright IBM Corporation 2011
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)