Atmosphere Framework White Paper

Jeanfrancois Arcand (jfarcand@apache.org)

Version 0.6

Introduction

The Atmosphere Framework is designed to make it easier to build asynchronous/Comet-based Web applications that include a mix of Comet and RESTful behavior. The Atmosphere Framework is portable and can be deployed on any Web Server that supports the Servlet Specification 2.3. This document introduces the framework and its module. For any questions or feedback, post them at users@atmosphere.dev.java.net

Terminology

- **Suspend**: The action of suspending consist of telling the underlying Web Server to not commit the response, e.g. to not send back to the browser the final bytes the browser is waiting for before considering the request completed.
- **Resume**: The action of resuming consist of completing the response, e.g. committing the response by sending back to the browser the final bytes the browser is waiting for before considering the request completed.
- **Broadcast**: The action of broadcasting consists of producing an event and distributing that event to one or many suspended response. The suspended response can then decide to discard the event or send it back to the browser.
- **Long Polling**: Long polling consists of resuming a suspended response as soon as event is getting broadcasted.
- **Http Streaming**: Http Streaming, also called forever frame, consists of resuming a suspended response after multiples events are getting broadcasted.
- **Native Asynchronous API**: A native asynchronous API means an API that is proprietary, e.g. if you write an application using that API, the application will not be portable across Web Server.

Note

If you have used previous version or Atmosphere like 0.3.1, we have reformulated and red-designed the AtmosphereHandler. Please read the document titled Migrating your Atmosphere Application from 0.3.x to 0.4.

What is the Atmosphere Framework?

The Atmosphere Framework contains several modules, which can be used depending on your needs:

- **Atmosphere Runtime** (Comet Portable Runtime): This module can be used to write POJO written in Java, JRuby or Groovy. The main component of this module is an AtmosphereHandler. An AtmosphereHandler can be used to suspend, resume and broadcast and allow the use of the usual HttpServletRequest and HttpServletResponse set of API.
- **Atmosphere Annotations:** This module defines the set of annotations that can be implemented to support Atmosphere concepts. By default, the Atmosphere Jersey module implements them, but any framework can also add an implementation.
- **Atmosphere Jersey:** This module can be used to write REST & Asynchronous Web application. The REST engine used is Jersey (Sun's JAX RS implementation).
- **Atmosphere Meteor**: This module can be used with existing Servlet or Servlet based technology like JSP, JSF, Struts, etc. The main component is a Meteor that can easily be looked up from any Java Object.
- **Atmosphere Bayeux**: This module support the work done by the comed.org group: The Bayeux Protocol.
- **Atmosphere Guice**: This module allow the use of Google Guice to configure your Atmosphere application

This tutorial will cover in details all of them. The Framework also contains plug-in that can be added on the fly:

- **Shoal** Plug-in: Allow the creation of Atmosphere Cluster using the Shoal Framework.
- **JGroups** Plug-in: Allow the creation of Atmosphere Cluster using the JGroups Framework
- **JMS** Plug In: Allow the creation of Atmosphere Cluster using JMS Queue/Topics
- **Grizzly** Plug In: Allow Atmosphere application to be natively embedded within the Grizzly Framework. A single builder API is available to help embedding an Atmosphere based application.

The Framework also contains it's own ready to use Web Server named the Atmosphere Spade Server, which consist of an end to end stack containing the Grizzly Web Server, Jersey and all Atmosphere modules and Plug-in.

The Atmosphere Framework supports natively the following Web Server asynchronous API:

- WebLogic's AbstractAsyncServlet

- Tomcat's CometProcessor
- GlassFish's CometHandler
- Jetty's Continuation
- JBoss' HttpEvent
- Grizzly's CometHandler
- Servlet 3.0's AsyncListener
- Google App Engine restricted environment.

If Atmosphere fails to detect the above native API, it will instead use its own asynchronous API implementation, which will consist of blocking a Thread per suspended connections. That means Atmosphere applications are guarantee to work on any Web Server supporting the Servlet specification version 2.3 and up. Note that it is also possible to write your own native implementation and replace the one used by default in Atmosphere by providing an implementation of the CometSupport SPI. See next section for more information.

Getting started

Independent of the module you decide to use, the following WAR file structure is required when using Atmosphere. Since Atmosphere uses Maven as build system, you can use Maven to generate the skeleton for your application:

```
%  mvn archetype:create -DgroupId=sample -DartifactId=<YOUR_APP_NAME> -
DarchetypeArtifactId=maven-archetype-webapp
```

The web.xml here will be used to either define the AtmosphereServlet or MeteorServlet. In order to make your application portable amongst all Web Servers, a file named context.xml is required. The file contains:

```
<Context>
      <Loader delegate="true"/>
</Context>
```

This file is required in order to run inside the Tomcat and JBoss' Web Server. Tomcat will looks under META-INF/context.xml, JBoss under WEB-INF/context.xml. If you are not planning to use those Web Server, then you don't need to add that file. But we strongly recommend you always include it so you application is portable across Web Servers.

To add support for the Atmosphere Framework, you need to define the AtmosphereServlet or AtmosphereFilter inside your application web.xml:

```
    <display-name>Atmosphere Servlet</display-name>
    <servlet>
        <description>AtmosphereServlet</description>
        <servlet-name>AtmosphereServlet</servlet-name>
        <servlet-
class>org.atmosphere.cpr.AtmosphereServlet|AtmosphereFilter</servlet-
class>
```

```
        <!-- Uncomment if you want to use Servlet 3.0 Async Support
        <async-supported>true</async-supported>
        -->
        <!—init-param are optional
        <init-param>org.atmosphere.*</init-param>
        <init-value>                </init-value>
        -->
        <load-on-startup>0</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>AtmosphereServlet</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
```

Mainly, all you need to do is to define the **AtmosphereServlet** or **AtmosphereFilter** depending on how you want to use the framework. You can optionally configure some init-param the framework will use:

- **org.atmosphere.useNative**: You can tell the framework which Asynchronous API: the Web Server native one or if available, the Servlet 3.0 Async API.
- **org.atmosphere.useBlocking**: You can tell the framework to use blocking I/O approach instead of using the underlying Asynchronous Native API or Servlet 3.0. When using blocking I/O, the calling Thread will block as soon as you suspend the response, and will finish its execution once the response gets resumed.
- **org.atmosphere.jersey.servlet-mapping:** define the context-root of a class/resources, which use atmosphere-core.

IMPORTANT: If you decide to use the framework using the AtmosphereFilter, be aware that only GlassFish, Grizzly and Jetty will take advantages of their asynchronous native API. Tomcat, JBoss, WebLogic etc. will use the blocking I/O approach. If you need to use the Asynchronous Native API all the time, make sure you write your Atmosphere application use the AtmosphereServlet.

Note that if you deploy your application inside a Web Server that doesn't have any Asynchronous Native API, the framework will detect it and use a blocking I/O approach (ex: Resin). Thus you are always guaranteed that your application can be deployed inside any Web Server that support the Servlet Specification 2.3 and up.

An Atmosphere application can be configured using a file called atmosphere.xml, located under META-INF/. The file is optional and the framework will use default value if not specified. When not specified, the framework will scan classes under WEB-INF/classes and load and register the one that implement the AtmosphereHandler interface. The atmosphere.xml contains information about AtmosphereHandler and their URL mapping, Broadcaster, Servlet, etc. The file takes the form of:

```
<atmosphere-handlers>
    <atmosphere-handler context-root="/twitter" class-
name="org.atmosphere.samples.twitter.TwitterAtmosphereHandler"/>
```

```
         <property name=”” value=””/>
    <atmosphere-handler …/>
</atmosphere-handlers>
```

From that file, Atmosphere can configure for an application:

- **context-root**: The context-root used to map the request with the AtmosphereHandler. As an example, if you want to map a request that takes the form of /foo/bar, you will maps the AtmosphereServlet to /foo, and set the context-root to /bar.

- **class-name**: The full qualified name of your AtmosphereHandler implementation

- **broadcaster**: The full qualified name of your AtmosphereHandler implementation

- **property**: A name/value pair that can be used to configure an AtmosphereHandler. As an example, defining <property name="servletClass" value="foo"/> will invokes on the AtmosphereHandler.setServletClass method with the value defined. Any property can then be configured at runtime using property.

- **comet-support**: An implementation of the CometSupport SPI. You can define your own Comet implementation that will be used by your application.

- **session-support**: By default, the Atmosphere Runtime uses Sessions to store some info, but Atmosphere Jersey by default doesn’t uses Sessions. You can turn on Sessions by setting the value to true.

Atmosphere Main Concept

There are really three main operations an asynchronous application might require. The first one is the ability to ***suspend*** the execution of a request/response processing until an asynchronous events occurs. Once the event occurs, you may want to ***resume*** the normal request/response processing, depending on the technique your application support (long polling or http streaming). The third operation consists of being able to ***broadcast*** or *push* asynchronous events and deliver them to suspended responses.

Chapter 1: Atmosphere Jersey (Asynchronous RESTful web Application)

The Concepts

Atmosphere Jersey is a module that allows the creation of Asynchronous RESTFul web service. The module extends Project Jersey, which is the implementation of JSR 311 (JAX-RS). It is strongly recommended to take a look at Jersey's Getting Started Tutorial before reading this section. This section will not explain how Jersey works and what JAX-RS/RESTful web services are.

An application that defines resources mapped to the root ("/") isn't by default required to have an META-INF/atmosphere.xml file. The Atmosphere Framework will auto-detect the Jersey runtime and start it automatically, and will register the resources defined to the "/" context-root (see atmosphere.xml explanation in chapter 2). You can also specify the context-root by adding the following init-param for the AtmosphereServlet|Filter:

```
<init-param>org.atmosphere.core.servlet-mapping</init-param>
<init-value>some value</init-value>
```

You can also define the following atmosphere.xml file to enable atmosphere-jersey:

```
<atmosphere-handlers>
  <atmosphere-handler context-root="/dispatch"
    class-name="org.atmosphere.handler.ReflectorServletProcessor">
      <property name="servletClass"
        value="com.sun.jersey.spi.container.servlet.ServletContainer"/>
    </atmosphere-handler>
</atmosphere-handlers>
```

As explained in more details in the Chapter 2, the ReflectorServletProcessor is used to manage the lifecycle of the Jersey's ServletContainer. The Atmosphere runtime will load that Servlet, and will extends Jersey internal with a set of annotations that will allow the creation of Asynchronous RESTful application. The operations we are the ability to suspend, resume, schedule and broadcast event.

@Suspend

The @Suspend annotation allow suspending the processing of the response. You can annotate any Resource's method with the @Suspend annotation, which is defined as:

```
public @interface Suspend {

    int period() default -1;

    enum SCOPE { REQUEST, APPLICATION, VM }

    SCOPE scope() default SCOPE.APPLICATION;

    boolean outputComments() default true;

    public boolean resumeOnBroadcast() default false;

    public Class<? extends AtmosphereResourceEventListener>[]
listeners() default {};
```

The **period** represent the time in second a response will stay suspended without any events occurring, or without resuming the response.

The **scope** concept is explained in more details later in this chapter: can a Broadcaster be used for broadcasting events to its associated request, to all suspended response within the application or to all suspended responses available on the VM. Note that the Broadcaster will be created after the annotated method gets executed.

The **outputComments** is used to tells the framework to not output any comments when the response gets suspended. By default, Atmosphere always output some comments to make browsers based on WebKit (Chrome, Safari) works properly when a response is suspended.

The **resumeOnBroadcast** tells Atmosphere to resume the response as soon as a Broadcast operations occurs. You usually set it to true if your planning to use the long polling Comet technique.

The listeners() is a set of class that implements that AtmosphereResourceEventListener, which can be used to monitor Atmosphere events like disconnect, resume and broadcast. The API is defined as:

```
void onResume(AtmosphereResourceEvent event);

void onDisconnect(AtmosphereResourceEvent event);

void onBroadcast(AtmosphereResourceEvent event);
```

The **onResume** will be invoked when the response is resuming, either because the period time out or the @Resume annotation was been executed. The **onDisconnect** will be invoked when the remote client close the connection. Note that not all Web Server support client disconnection detection. Refer to Annex A for more information. The **onBroadcast** will be invoked every time a broadcast operations occurs.

Usually you use that annotation by doing:

```
@Suspend(default=60)
```

@Resume

The @Resume annotation is the equivalence of *AtmosphereResource.resume*. You can annotate any resource's method with the @Resume annotation, which is defined as:

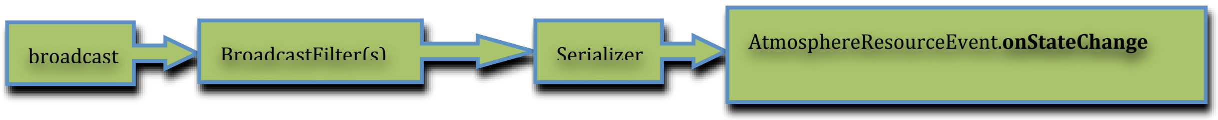
```
public @interface Resume {

    int count() default 1; // long polling
}
```

The count represent the number of Broadcast event that need to happens before the suspended response gets resumed. Usually you use that annotation by doing:

```
@Resume
```

@Broadcast

Once of the key concept of the Atmosphere Framework is called Broadcaster. A Broadcaster can be used to broadcast (or push back) asynchronous events to the set or subset of suspended responses. A Broadcaster can be used when events are ready to be written back to the browser. A Broadcaster can contain zero or more BroadcastFilter, which can be used to transform the events before they get written back to the browser. For example, any malicious characters can be filtered before they get written back. A Broadcaster can contain zero or one Serializer. A Serializer allows an application to decide how the event will be serialized and written back to the browser. When no Serializer is defined, the HttpServletResponse's output stream will be used and the event written as it is. Hence, firing a broadcast will produce the following chain of invocation:



The API looks like:

```
public interface Broadcaster {

        public Future<Object> broadcast(Object o);

        public Future<Object> delayBroadcast(Object o);

        public Future<Object> delayBroadcast(Object o, long delay, TimeUnit t);

        public Future<?> scheduleFixedBroadcast(Object o, long period, TimeUnit t);

        public Future<Object> broadcast(Object o, AtmosphereResource event);

        public Future<Object> broadcast(Object o, Set<AtmosphereResource> subset);
```

Internally, a Broadcaster uses an ExecutorService to execute the above chain of invocation. That means a call to *Broadcaster.broadcast(..)* will not block unless you use the returned Future API, and will use a set of dedicated threads to execute the broadcast. By default, an ExecutorServices will be created and the number of threads will be based on the OS' core/processor (usually 1 or 2). You can also configure your own ExecutorServices using a BroadcasterConfig.

A Broadcaster can also be used to broadcast delayed events, e.g. an application can decide to delay an events until another events happens or after a delay. This is particularly useful when you need to aggregate events and write them all in once.

A Broadcaster can also be used to broadcast periodic events. It can be configured using an ScheduledExecutorService to produce periodic events. As with the ExecutorService, the number of OS' core/processor will be used to determine the default number of threads. You can also configure your own SchedulerExecutorService via the BroadcastConfig.

One final word on Broadcaster: by default, a Broadcaster will broadcast using all AtmosphereResourceEvent on which the response has been suspended, e.g. *AtmosphereResource.suspend()* has been invoked. This behavior is configurable and you can configure it by invoking the *Broadcaster.setScope()*:

- REQUEST: broadcast events only to the AtmosphereResourceEvent associated with the current request.
- APPLICATION: broadcast events to all AtmosphereResourceEvent created for the current web application
- VM: broadcast events to all AtmosphereResourceEvent created inside the current virtual machine.

The default is APPLICATION. Broadcaster are retrieved using an *AtmosphereResource.getBroadcaster()* or can also be looked up from any Java objects using BroadcasterLookup.

An application can define its own implementation and tell the framework to use it by declaring it inside atmosphere.xml:

```
<atmosphere-handlers>
    <atmosphere-handler..
      broadcaster="org.atmosphere.samples.twitter.TwitterBroadcaster"/>
```

The **@Broadcast** annotation is the equivalence of AtmosphereResource.getBroadcaster's. You can annotate any resource's method with the @Broadcast annotation, which is defined as:

```
public @interface Broadcast {

    public Class<? extends BroadcastFilter>[] value()
      default {};

    public boolean resumeOnBroadcast() default false;

    public int delay() default -1;
}
```

The **value** field defines which BroadcastFilter needs to be added to the Broadcaster associated with the current suspended response. The **resumeOnBroadcast** means that the response will be resumed as soon as an event is broadcasted. The **delay** can be used to delay the execution of the events, which is the same as doing Broadcaster.delay(…).

Usually you use that annotation by doing:

```
@Broadcast({XSSHtmlFilter.class, JsonpFilter.class})
```

You can also broadcast any type of object by using a Broadcastable. A Broadcastable will tell the framework to broadcast event using the Broadcaster and the Object a Broadcastable represents:

```
@POST
@Consumes(MediaType.APPLICATION_FORM_URLENCODED)
@Produces("text/html")
@Broadcast
public Broadcastable onPush()
    Broadcastable b = new Broadcastable(m, bc);
    return b;
}
```

Note that you can broadcast any type returned by the @Produced annotation:

```
@POST
@Path("{counter}")
@Produces({"application/xml", "application/json"})
public MyEvent resume(){
    return new MyEvent("POST");
}
```

Injectable Objects

You can also inject at runtime Broadcaster, AtmosphereResourceEvent and BroadcasterLookup instance by using the Jersey's @Context annotation:

```
@Context Broadcaster bc
@Context AtmosphereResourceEvent e
@Context BroadcasterLookup b
```

See the PubSub sample for an easy to understand sample.

Writing a simple Chat application using Atmosphere Jersey

Let's re-write the Chat application we described using atmosphere-runtime. The complete source of the following sample can be downloaded from atmosphere.dev.java.net. First, let's generate the project using Maven.

```
%  mvn archetype:create -DgroupId=org.atmosphere.samples  -
DartifactId=chat -DarchetypeArtifactId=maven-archetype-webapp
```

Which will create the following structure:

```
./chat
./chat/pom.xml
./chat/src
./chat/src/main
./chat/src/main/resources
./chat/src/main/webapp
./chat/src/main/webapp/index.jsp
./chat/src/main/webapp/WEB-INF
./chat/src/main/webapp/WEB-INF/web.xml
```

Next let's edit our pom.xml and defines the atmosphere-core and it's dependencies

```
 <dependency>
      <groupId>org.atmosphere</groupId>
      <artifactId>atmosphere-core</artifactId>
      <version>${atmosphere-version}</version>
  </dependency>
```

As for any Atmosphere application, you define the AtmosphereServlet inside the web.xml:

```
<servlet>
    <description>AtmosphereServlet</description>
    <servlet-name>AtmosphereServlet</servlet-name>
    <servlet-class>org.atmosphere.cpr.AtmosphereServlet</servlet-class>
    <!-- Uncomment if you want to use Servlet 3.0 Async Support
    <async-supported>true</async-supported>
    -->
</servlet>
<servlet-mapping>
    <servlet-name>AtmosphereServlet</servlet-name>
    <url-pattern>/resource/*</url-pattern>
</servlet-mapping>
```

Next we can either define the atmosphere.xml:

```
<atmosphere-handlers>
  <atmosphere-handler context-root="/resources"
```

```
        class-name="org.atmosphere.handler.ReflectorServletProcessor">
          <property name="servletClass"
            value="com.sun.jersey.spi.container.servlet.ServletContainer"/>
        </atmosphere-handler>
    </atmosphere-handlers>
```

Or add the org.atmosphere.core.servlet-mapping init-param under the AtmosphereServlet defined above

```
      <init-param>
        <param-name>org.atmosphere.core.servlet-mapping</param-name>
        <param-value>/resources</param-value>
      </init-param>
```

Now let's use the annotation defined in the previous section:

```
@Suspend
@GET
@Produces("text/html;charset=ISO-8859-1")
public String suspend() {
     return “”;
}

@Broadcast({XSSHtmlFilter.class, JsonpFilter.class})
@Consumes("application/x-www-form-urlencoded")
@POST
@Produces("text/html;charset=ISO-8859-1")
public String publishMessage(MultivaluedMap<String, String> form) {
    String action = form.getFirst("action");
    String name = form.getFirst("name");

    if ("login".equals(action)) {
        return ("System Message" + "__" + name + " has joined.");
    } else if ("post".equals(action)) {
        return name + "__" + form.getFirst("message");
    } else {
        throw new WebApplicationException(422);
    }
}

@Schedule(period=30)
@POST
@Path("/ping")
public String pingSuspendedClients(){
    return "Atmosphere__ping";
}
```

The way it works is as soon the suspend method will be invoked, its returns value will be written back (here empty) to the client and the response suspended. When the publishMessage will be invoked, its returned value will be broadcasted, e.g. all suspended responses will be invoked and the value written as it is to the client.

We also added support for “ping”, which is enabled by the @Schedule annotation of the pingSuspendedClients. The complete code can be browsed from here.

Chapter 2: Atmosphere Runtime (Comet Portable Runtime)

The concepts

The Atmosphere runtime is the foundation of the Framework. All others modules build on top of it. The main component is called an **AtmosphereHandler** and it is defined as:

```
public interface AtmosphereHandler<F,G> {
      public void onRequest(AtmosphereResource<F,G> event)
            throws IOException;

      public void onStateChange(AtmosphereResourceEvent<F,G> event)
            throws IOException;
}
```

The *onRequest* is invoked every time a request uri match the context-root of the AtmosphereHandler defined within the definition of the AtmosphereHandler in atmosphere.xml:

```
<atmosphere-handlers>
    <atmosphere-handler context-root="/twitter" class-
name="org.atmosphere.samples.twitter.TwitterAtmosphereHandler"/>
</atmosphere-handlers>
```

AtmosphereResource

The main object to interact with when the onRequest is executed is an **AtmosphereResource.** An AtmosphereResource can be used to manipulate the current request and response and used to suspend or resume a response, and also broadcast events.

```
      public interface AtmosphereResource<E,F> {

            public void resume();

            public void suspend();

            public void suspend(long timeout);

            public E getRequest();

            public F getResponse();

            public AtmosphereConfig getAtmosphereConfig();

            public Broadcaster getBroadcaster();

            public void setBroadcaster(Broadcaster broadcaster);

            public void setSerializer(Serializer s);

            public void write(OutputStream os, Object o) throws IOException;

            public Serializer getSerializer();
```

AtmosphereResourceEvent

An AtmosphereResourceEvent contains the state of the response, e.g. has the response been suspended, resumed, etc. API looks like:

```
        public interface AtmosphereResourceEvent<E,F> {
                public void resume();

                public boolean isResumedOnTimeout();

                public boolean isCancelled();

                public boolean isSuspended();

                public boolean isResuming();

                public Object getMessage();

                public void write(OutputStream os, Object o) throws IOException;
```

The AtmosphereHandler's *onStateChange* is invoked when:

- An event is broadcasted using a Broadcaster (more on the topic below)
- An event is about to resume based on a timeout
- A browser closes the remote connection. Not all Web Server supports that mechanism. See Annex A for more info.

Ready to use AtmosphereHandler

An Atmosphere application can consist of one or more AtmosphereHandler. The Framework contains two implementations of that interface. The first is called *AbstractReflectorAtmosphereHandler* and contains a default implementation for the *onMessage* method. The *onMessage* in that case reflect the broadcasted event, e.g. it writes the broadcasted event without any modification. If a Serializer has been configured, the Serializer will be used for writing the event. If none, the HttpServletResponse writer (or output stream) will be used.

The second implementation is RefectorServletProcessor and can be used to execute Servlet from an AtmosphereHandler. A ReflectorServletProcessor will forward the request to the Servlet.service() method of a Servlet and also make available the associated AtmosphereResourceEvent to the Servlet. That way any existing Servlet can use the Atmosphere Framework by retrieving the AtmosphereResourceEvent from the HttpServletRequest.getAttribute using:

org.atmosphere.runtime.AtmosphereResource

as a key:
HttpServletRequest.getAttribute("org.atmosphere.runtime.AtmosphereResource").

This AtmosphereHandler is helpful when you need to run another framework on top of Atmosphere. If you are planning to use the framework from an existing application, take a look at Chapter 3 for the recommended way. You define the ReflectorServletProcessor in atmosphere.xml and it's associated Servlet by doing:

```
<atmosphere-handlers>
    <atmosphere-handler context-root="/dispatch"
      class-name="org.atmosphere.handler.ReflectorServletProcessor"
    <property name="servletClass"
value="com.sun.jersey.spi.container.servlet.ServletContainer"/>
    </atmosphere-handler>
</atmosphere-handlers>
```

You need to define a property called **servletClass** and set your Servlet's fully qualified class name.

Writing a simple Chat application using Atmosphere runtime

The complete source of the following sample can be downloaded from atmosphere.dev.java.net. First, let's generate the project using Maven.

```
%  mvn archetype:create -DgroupId=org.atmosphere.samples  -
DartifactId=chat -DarchetypeArtifactId=maven-archetype-webapp
```

Which will create the following structure:

```
./chat
./chat/pom.xml
./chat/src
./chat/src/main
./chat/src/main/resources
./chat/src/main/webapp
./chat/src/main/webapp/index.jsp
./chat/src/main/webapp/WEB-INF
./chat/src/main/webapp/WEB-INF/web.xml
```

Next let's edit our pom.xml and defines the atmosphere-runtime and it's dependencies

```
<dependency>
      <groupId>org.atmosphere</groupId>
      <artifactId>atmospher-runtime</artifactId>
      <version>${atmosphere-version}</version>
  </dependency>
```

We are now ready to write our first AtmosphereHandler, which is the central piece of any Atmosphere runtime application. Let's just implement this interface:

```
public void onRequest
(AtmosphereResource<HttpServletRequest,HttpServletResponse>
                event) throws IOException {

      HttpServletRequest req = event.getRequest();
      HttpServletResponse res = event.getResponse();

      if (req.getMethod().equalsIgnoreCase("GET")) {
            event.suspend();
            Broadcaster bc = event.getBroadcaster();
            bc.getBroadcasterConfig().addFilter(new XSSHtmlFilter());

            Future<Object> f =   bc.broadcast(
               event.getAtmosphereConfig().getWebServerName()
                    + "**has suspended a connection from "
                    + req.getRemoteAddr());

            try {
                // Wait for the push to occurs.
                // This block the current Thread
                f.get();
            } catch (Throwable t) {

            }

            bc.scheduleFixedBroadcast(req.getRemoteAddr()
                  + "**is connected", 30, TimeUnit.SECONDS);
            bc.delayBroadcast("Underlying Response now suspended");
```

The central piece is the AtmosphereResource, from which we can retrieve the request and response object. Next we do some setup and then once we are ready we just need to invoke the AtmosphereResourceEvent.suspend, which will automatically tell Atmosphere runtime to not commit the response. Not committing the response means we can re-use it later for writing. In the current sample, we will use the suspended response when someone enter join or enter sentence inside the chat room. Now let's assume when a user logs in or enter sentences, the browser set a POST (posting some data). So when a user logs in:

```
      } else if (req.getMethod().equalsIgnoreCase("POST")) {
            res.setCharacterEncoding("UTF-8");
            String action = req.getParameterValues("action")[0];
            String name = req.getParameterValues("name")[0];

            if ("login".equals(action)) {
                  event.getBroadcaster().broadcast(WELCOME_MSG
                  + event.getAtmosphereConfig().getWebServerName()

                  + "**" + name + " has joined.");

                res.getWriter().write("success");
                res.getWriter().flush();
```

Since Broadcaster's role is to publish data to the suspended responses, as soon as we broadcast data, all suspended responses will be given a chance to write the content of the broadcast when the *AtmosphereHandler.onStateChange* gets invoked. Above we just broadcast the name and also which Web Server we are running on Here let's assume we just reflect (write) what we receive, so we just implements inside the onStateChange:

```
      public void onStateChange(
            AtmosphereResourceEvent<HttpServletRequest,
                    HttpServletResponse> event) throws IOException {

        HttpServletRequest req = event.getResource().getRequest();
        HttpServletResponse res = event.getResource().getResponse();
        String msg = (String)event.getMessage();

        if (event.isCancelled()){
            event.getResource().getBroadcaster().broadcast(
                  req.getSession().getAttribute("name").toString() + "
            has left");
        } else if (event.isResuming() || event.isResumedOnTimeout()) {
            String script =
                  "<script>window.parent.app.listen();\n</script>";

            res.getWriter().write(script);
            res.getWriter().flush();
        } else {
            res.getWriter().write(msg);
            res.getWriter().flush();
        }
        return event;
```

In the above code we make just check if the event has been cancelled (because the browser closed the connection) or resumed, and if not we just write the broadcasted message.

Now let's assume we want to have a more fine grain way to map our AtmosphereHandler to the request. To achieve that, create a file called atmosphere.xml under src/main/webapp/META-INF/ and define the mapping you want:

```
 <atmosphere-handlers>
     <atmosphere-handler context-root="/chat" class-
name="org.atmosphere.samples.chat.ChatAtmosphereHandler">
         <property name="name" value="Chat"/>
     </atmosphere-handler>
 </atmosphere-handlers>
```

Now let's explore the client side. First, let's write a very simple index.html file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
    <head>
        <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8" />
        <title>Atmosphere Chat</title>
        <link rel="stylesheet" href="stylesheets/default.css"
type="text/css" />
        <script type="text/javascript"
src="javascripts/prototype.js"></script>
        <script type="text/javascript"
src="javascripts/behaviour.js"></script>
        <script type="text/javascript"
src="javascripts/moo.fx.js"></script>
        <script type="text/javascript"
src="javascripts/moo.fx.pack.js"></script>
        <script type="text/javascript"
src="javascripts/application.js"></script>
    </head>
    <body>
        <div id="container">
            <div id="container-inner">
                <div id="header">
                    <h1>Atmosphere Chat</h1>
                </div>
                <div id="main">
                    <div id="display">
                    </div>
                    <div id="form">
```

```
                        <div id="system-message">Please input your
name:</div>
                        <div id="login-form">
                            <input id="login-name" type="text" />
                            <br />
                            <input id="login-button" type="button"
value="Login" />
                        </div>
                        <div id="message-form" style="display: none;">
                            <div>
                                <textarea id="message" name="message"
rows="2" cols="40"></textarea>
                                <br />
                                <input id="post-button" type="button"
value="Post Message" />
                            </div>
                        </div>
                    </div>
                </div>
            </div>
        </div>
        <iframe id="comet-frame" style="display: none;"></iframe>
    </body>
</html>
```

Simple form that will send back to the server the login's name and the chat message entered. To update on the fly the interface as soon as our ChatAtmosphereHandler.onStateChange write/send us data, let's use prototype and behavior javascript. Below we are assuming you are either familiar with those frameworks or have basic understanding how they work. This will be defined under application.js. As soon as the user enter its login name, let's do

```
     post: function() {
        var message = $F('message');
        if(!message > 0) {
            return;
        }
        $('message').disabled = true;
        $('post-button').disabled = true;

        var query =
        'action=post' +
        '&name=' + encodeURI($F('login-name')) +
        '&message=' + encodeURI(message);
        new Ajax.Request(app.url, {
            postBody: query,
            onComplete: function() {
                $('message').disabled = false;
                $('post-button').disabled = false;
                $('message').focus();
                $('message').value = '';
            }
        });
    },
```

When the user write new chat message, let's push

```
    post: function() {
        var message = $F('message');
        if(!message > 0) {
            return;
        }
        $('message').disabled = true;
        $('post-button').disabled = true;

        var query =
        'action=post' +
        '&name=' + encodeURI($F('login-name')) +
        '&message=' + encodeURI(message);
        new Ajax.Request(app.url, {
            postBody: query,
            onComplete: function() {
                $('message').disabled = false;
                $('post-button').disabled = false;
                $('message').focus();
                $('message').value = '';
            }
        });
    }
```

Now when we get response, we just update the page using

```
   update: function(data) {
        var p = document.createElement('p');
        p.innerHTML = data.name + ':<br/>' + data.message;

        $('display').appendChild(p);

        new Fx.Scroll('display').down();
    }
```

The way the index.html and application.js interact is simply defined by:

```
      var rules = {
          '#login-name': function(elem) {
              Event.observe(elem, 'keydown', function(e) {
                  if(e.keyCode == 13) {
                      $('login-button').focus();
                  }
              });
          },
          '#login-button': function(elem) {
              elem.onclick = app.login;
          },
          '#message': function(elem) {
              Event.observe(elem, 'keydown', function(e) {
                  if(e.shiftKey && e.keyCode == 13) {
                      $('post-button').focus();
                  }
              });
          },
          '#post-button': function(elem) {
              elem.onclick = app.post;
```

```
        }
    };
    Behaviour.addLoadEvent(app.initialize);
    Behaviour.register(rules);
```

Finally, just deploy your war file into any Web Server and see the final result.

Chapter 3: Atmosphere Meteor

The Concepts

The Atmosphere Meteor module allows any existing Servlet based application (Wicket, JSP, JSF, etc.) to easily add asynchronous support. You enable Atmosphere Meteor by adding the MeteorServlet inside your web.xml:

```
   <servlet>
        <description>MeteorServlet</description>
        <servlet-name> MeteorServlet </servlet-name>
        <servlet-class>org.atmosphere.runtime.MeteorServlet</servlet-class>
        <init-param>
            <param-name>org.atmosphere.servlet</param-name>
            <param-value>…</param-value>
        </init-param>
        <load-on-startup>0</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name> MeteorServlet </servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
```

The MeteorServlet can be configured using two or more init-param:

- org.atmosphere.servlet: The MeteorServlet will invoke the Servlet and allow them to suspend, resume and broadcast event.
- org.atmosphere.filter: The MeteorServlet will invoke those Filters and allow them to suspend, resume and broadcast event.

Unlike atmosphere-runtime and atmosphere-core module, this module doesn't require any atmosphere.xml configuration file.

Now from any Filters and Servlet, all you need to do is to interact with the Meteor API:

```
        public final static Meteor build(
           HttpServletRequest r, List<BroadcastFilter> l,Serializer s)

        public Meteor suspend(long l)

        public Meteor resume()

        public Meteor broadcast(Object o)

        public void addListener(AtmosphereResourceEventListener e)
```

All you need to do is to get an instance of a Meteor by invoking the build method, and passing an instance of the request you eventually want to suspend its associated response. You can optionally pass your set of **BroadcastFilter** as well as a **Serializer**.

Writing a simple Chat application using Atmosphere Meteor

Let's re-write the Chat application we described using atmosphere-runtime. The complete source of the following sample can be downloaded from atmosphere.dev.java.net. First, let's generate the project using Maven.

```
%  mvn archetype:create -DgroupId=org.atmosphere.samples  -
DartifactId=chat -DarchetypeArtifactId=maven-archetype-webapp
```

Which will create the following structure:

```
./chat
./chat/pom.xml
./chat/src
./chat/src/main
./chat/src/main/resources
./chat/src/main/webapp
./chat/src/main/webapp/index.jsp
./chat/src/main/webapp/WEB-INF
./chat/src/main/webapp/WEB-INF/web.xml
```

Next, let's define our MeteorServlet as well as our Servlet inside the web.xml:

```
<description>Atmosphere Chat</description>
<display-name>Atmosphere Chat</display-name>
<servlet>
    <description>MeteorServlet</description>
    <servlet-name>MeteorServlet</servlet-name>
    <servlet-class>org.atmosphere.runtime.MeteorServlet</servlet-class>
    <init-param>
        <param-name>org.atmosphere.servlet</param-name>
        <param-value>org.atmosphere.samples.chat.MeteorChat</param-value>
    </init-param>
    <load-on-startup>0</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>MeteorServlet</servlet-name>
    <url-pattern>/Meteor</url-pattern>
</servlet-mapping>
```

Now we just need to write a simple Servlet like the following:

```
public class MeteorChat extends HttpServlet {

    /**
     * List of {@link BroadcastFilter}
     */
    private final List<BroadcastFilter> list;

    public MeteorChat() {
        list = new LinkedList<BroadcastFilter>();
        list.add(new XSSHtmlFilter());
        list.add(new JsonpFilter());
    }

    @Override
    public void doGet(HttpServletRequest req, HttpServletResponse res) throws
                                                                   IOException{
        Meteor m = Meteor.build(req, list, null);

        req.getSession().setAttribute("meteor", m);

        m.suspend(-1);
        m.broadcast(req.getServerName()
           + "__has suspended a connection from " + req.getRemoteAddr());
    }

    @Override
    public void doPost(HttpServletRequest req, HttpServletResponse res) throws
IOException {
        Meteor m = (Meteor)req.getSession().getAttribute("meteor");
        res.setCharacterEncoding("UTF-8");
        String action = req.getParameterValues("action")[0];
        String name = req.getParameterValues("name")[0];

        if ("login".equals(action)) {
            req.getSession().setAttribute("name", name);
            m.broadcast("System Message from " + req.getServerName() + "__"
                       + name + " has joined.");
            res.getWriter().write("success");
            res.getWriter().flush();
        } else if ("post".equals(action)) {
            String message = req.getParameterValues("message")[0];
            m.broadcast(name + "__" + message);
            res.getWriter().write("success");
            res.getWriter().flush();
        } else {
            res.setStatus(422);

            res.getWriter().write("success");
            res.getWriter().flush();
        }
    }
}
```

As you can see, using a Meteor is simple and should be used from any existing Servlet. Once you have a Meteor, you can suspend, resume and broadcast events like you can do with Atmosphere runtime or core module. The client side for the example above is the same as described in Chapter 1 with atmosphere-runtime.

Chapter 4: Atmosphere Plug In

Atmosphere Grizzly Plug In

Use Project Grizzly Web Framework to embed Atmosphere

This plug in allow you to programmatically add Atmosphere support to the Project Grizzly's Web Server embed API. You can learn more about Project Grizzly and its Web Framework by going to their home page. The Atmosphere implementation is called AtmosphereAdapter and can be used with Grizzly WebServer by doing:

```
GrizzlyWebServer ws = new GrizzlyWebServer();

AtmosphereAdapter a = new AtmosphereAdapter();

a.addAtmosphereHandler(new AtmosphereHandler()..);

ws.addGrizzlyAdapter(a);

ws.start();
```

You can also enable atmosphere-core by doing:

```
GrizzlyWebServer ws = new GrizzlyWebServer();

AtmosphereAdapter a = new AtmosphereAdapter();

a.setResource(“my.resource.package”);

ws.addGrizzlyAdapter(a);

ws.start();
```

Deploying your AtmosphereAdapter using GlassFish

You can also deploy an AtmosphereAdapter inside GlassFish v3 Nucleus distribution without the needs to bundles your application inside a war file and without the need of web.xml. All you need to do is to create a **META-INF/grizzly-glassfish.xml** and register your AtmosphereAdapter:

```
<adapters>
  <adapter context-root="/atmosphere" class-
  name="org.atmosphere.grizzly.AtmosphereAdapter"/>
</adapters>
```

Then you can deploy your jar into GlassFish v3 using the admin tool or simply doing

```
java –jar glassfish.jar yourAtmosphereApplication.jar
```

Atmosphere Cluster Plug In

Any Atmosphere application can be deployed inside a cluster using the Atmosphere Cluster Plug In. The framework currently supports the following cluster framework:

- Shoal
- JGroups
- JMS

Clustering support in Atmosphere is supported using BroadcastFilter. Hence, to add support for clustering, all you need to do add the BroadcastFilter to your Broadcaster:

```
Broadcaster bc = event.getBroadcaster();
bc.addBroadcastFilter(new ShoalFilter());
```

Any Broadcast operations will be broadcasted to all your Atmosphere applications deployed inside your cluster. If you use atmosphere-core, all you need to do is to annotate your method with the @Cluster annotation:

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.runtime)
@Documented
public @interface Cluster {
     String name() default "Atmosphere";

     Class<? extends
org.atmosphere.runtime.ClusterBroadcastFilter>[] value()default
{org.atmosphere.runtime.ClusterBroadcastFilter.class};
```

You use the annotation by doing:

```
@Broadcast
@Consumes("application/x-www-form-urlencoded")
@POST
@Produces("text/html")
@Cluster(name="chat,value=ShoalFilter|JgroupsFilter|JMSFilter")
public String publishMessage(MultivaluedMap form) {
```

Chapter 5: Atmosphere Spade Server

The Atmosphere Space Server is an end-to-end stack that aggregate technology like:

- Grizzly's Servlet Container
- Jersey
- Atmosphere runtime, Jersey and Meteor
- Atmosphere Plug-in

The server is bundled inside a single jar and can be used using the command line:

```
java –jar atmosphere-spade-server.jar …

Usage: org.atmosphere.spade.AtmosphereSpadeLauncher [options]

    -p, --port=port                   Runs Atmosphere on the specified port.
                                      Default: 8080
    -a, --apps=application path       The AtmosphereServlet folder or jar or war
                                     location.
                                      Default: .
    -rp, --respourcespackage=package  The resources package name
                                      Default:
    -sp, --servletPath=path           The path AtmosphereServlet will serve
                                       resources
                                      Default: .
    -h, --help                        Show this help message.
```

The server can also be embedded into any application using the AtmosphereSpadeServer API:

```
public static AtmosphereSpadeServer build(String u)

public static AtmosphereSpadeServer build
              (String u, String resourcesPackage)

public AtmosphereSpadeServer addAtmosphereHandler
              (String mapping,AtmosphereHandler h

public void setResourcePackage(String resourcePackage)

public AtmosphereSpadeServer start() throws IOException

public AtmosphereSpadeServer stop() throws IOException
```

You can build AtmosphereSpadeServer by just doing:

```
AtmosphereSpadeServer.build("http://localhost:8080").start();
```

You can configure the AtmosphereSpadeServer to deploy your AtmosphereHandler:

```
AtmosphereSpadeServer.addAtmosphereHandler
                              ("/chat",new ChatAtmoshereHandler);
```

AtmosphereSpadeServer can also be used with atmosphere-core by either passing the package name of your resource:

```
AtmosphereSpadeServer.build(“http://localhost”,”org.atmosphere”);
```

Chapter 6: Support for the Bayeux Protocol

The Bayeux Protocol is a pub/sub protocol developed by the Cometd.org. If you are not familiar with the Bayeux protocol, I recommend you read the specification before reading that chapter.

By default, the Cometd.org implementation uses a blocking thread approach when suspending responses, except when deployed on Jetty. The reason is the main developer of Cometd.org is also the lead of Jetty. The Atmosphere Bayeux Plug In fixes that issue by running on top of atmosphere-runtime, which always uses native Comet Implementation before blocking a thread to suspend the response. Blocking a thread per request may cause serious performance issue when too may threads need to be created. Atmosphere Bayeux Plug In the rescue!

To deploy your Bayeux application using Atmosphere, you first need to define the following atmosphere.xml

```
<atmosphere-handlers>
    <atmosphere-handler context-root="/cometd" class-
name="org.atmosphere.handler.ReflectorServletProcessor">
        <property name="servletClass"
value="org.atmosphere.plugin.bayeux.AtmosphereBayeuxServlet"/>
    </atmosphere-handler>
</atmosphere-handlers>
```

Here you just tell the ReflectorServletProcessor to use the AtmosphereBayeuxServlet, which is an extension to the normal Cometd.org Servlet implementation. Next is to define in web.xml the AtmosphereServlet the same way you would have normally defined the Cometd.org's ContinuationCometdServlet:

```
    <servlet>
        <servlet-name>cometd</servlet-name>
       <servlet-class>org.atmosphere.cpr.AtmosphereServlet</servlet-class>
        <init-param>
            <param-name>filters</param-name>
            <param-value>/WEB-INF/filters.json</param-value>
        </init-param>
        ….
        <load-on-startup>1</load-on-startup>
    </servlet>
```

You can download from atmosphere.dev.java.net the Cometd.org samples.

Annex A - Web Server differences

All the Web Servers aren't supporting the same functionality when their native Comet API is used. Below is a table explaining who support what.

	Jetty	Tomcat	GlassFish	WebLogic	JBossWeb
Optional atmosphere.xml	X	X	X		X
Optional web.xml init-param com.sun.jersey.config.property.packages	X	X	X		X
Auto-detect client remote disconnection (client disconnect)		X (*)	X		X

(*) You must configure the CometConnectionManagerValve Valve

The Servlet 3.0 CometSupport SPI supports all the above, and the Blocking I/O CometSupport SPI supports all except the client disconnect.