# Reverse Ajax

## Introduction

Web development has evolved considerably in the past few years. We're beyond the static web pages linked together, which caused browser refreshing and waiting for pages to load. Now, the demand is for completely dynamic applications accessible from the web. These applications often need to be as fast as possible and provide nearly real-time components. In this document, learn how to develop event-driven web applications using Reverse Ajax techniques.

## Ajax, Reverse Ajax, and WebSockets

**Asynchronous JavaScript and XML (Ajax),** a browser feature accessible in JavaScript, allows a script to make an HTTP request to a website behind the scenes, without the need for a page reload. Ajax has been around more than a decade. Though the name includes XML, you can transfer nearly anything in an Ajax request. The most commonly used data is JSON, which is close to JavaScript syntax and consumes less bandwidth.

**Reverse Ajax** is essentially a concept: being able to send data from the server to the client. In a standard HTTP Ajax request, data is sent to the server. Reverse Ajax can be simulated to issue an Ajax request, in specific ways that are covered in this article, so the server can send events to the client as quickly as possible (low-latency communication).

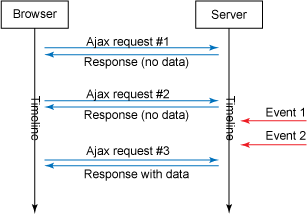
**WebSockets**, which comes from HTML5, is a much more recent technique. Many browsers already support it (Firefox, Google Chrome, Safari, and others). WebSockets enables bidirectional, full-duplex communication channels. The connection is opened through a sort of HTTP request, called a WebSockets handshake, with some special headers. The connection is kept alive, and you can write and receive data in JavaScript, as if you were using a raw TCP socket.

## Reverse Ajax techniques

The goal of Reverse Ajax is to let the server push information to the client. Ajax requests are stateless by default, and can only be opened from the client to the server. You can bypass this limitation by using the techniques to simulate responsive communication between the server and client.

## Compare communications between the server and client

### HTTP polling and JSONP polling



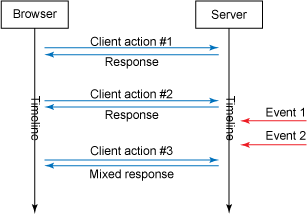
Polling involves issuing a request from the client to the server to ask for some data. This is obviously a mere Ajax HTTP request. To get the server events as soon as possible, the polling interval (time between requests) must be as low as possible. There's a drawback: if this interval is reduced, the client browser is going to issue many more requests, many of which won't return any useful data, and will consume bandwidth and processing resources for nothing.

JSONP polling is essentially the same as HTTP polling. The difference, however, is that with JSONP you can issue cross-domain requests (requests not in your domain)

Polling in JavaScript has advantages and disadvantages.

* **Advantages**: It's really easy to implement and does not require any special features on the server side. It also works in all browsers.
* **Disadvantage**: This method is rarely employed because it does not scale at all. Imagine the quantity of lost bandwidth and resources in the case of 100 clients each issuing polling requests for 2 seconds, where 30% of the requests returned no data.

### Piggyback



Piggyback polling is a much more clever method than polling since it tends to remove all non-needed requests (those returning no data). There is no interval; requests are sent when the client needs to send a request to the server. The difference lies in the response, which is split into two parts: the response for the requested data and the server events, if any occurred.

When implementing the piggyback technique, typically all Ajax requests targeting the server might return a mixed response.

* **Advantages**: With no requests returning no data, since the client controls when it sends requests, you have less resource consumption. It also works in all browsers and does not require special features on the server side.
* **Disadvantage**: You have no clue when the events accumulated on the server side will be delivered to the client because it requires a client action to request them.

### Comet

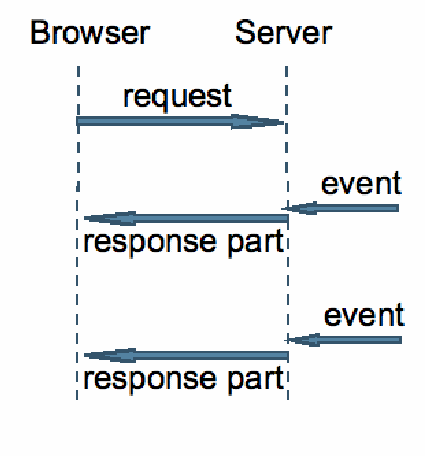
Reverse Ajax with polling or piggyback is very limited: it does not scale and does not provide low-latency communication (when events arrive in the browser as soon as they arrive on the server). Comet is a web application model where a request is sent to the server and kept alive for a long time, until a time-out or a server event occurs. When the request is completed, another long-lived Ajax request is sent to wait for other server events. With Comet, web servers can send the data to the client without having to explicitly request it.

The big advantage of Comet is that each client always has a communication link open to the server. The server can push events on the clients by immediately committing (completing) the responses when they arrive, or it can even accumulate and send bursts. Because a request is kept open for a long time, special features are required on the server side to handle all of these long-lived requests

Implementations of Comet can be separated into two types: those using **streaming** mode, and those using **long polling**.

### Comet using HTTP streaming

In streaming mode, one persistent connection is opened. There will only be a long-lived request (#1 in Figure 3) since each event arriving on the server side is sent through the same connection. Thus, it requires on the client side a way to separate the different responses coming through the same connection. Technically speaking, two common techniques for streaming include Forever Iframes (hidden IFrames) or the multi-part feature of the XMLHttpRequest object used to create Ajax requests in JavaScript.



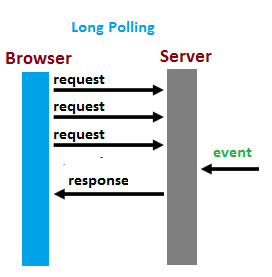
**Client:** Support: Firefox 3.x to 8.x, Firefox 9.x to 11.x, Chrome 12.x and lower, Chrome 13.x and higher, IE 6.x to 9.x, IE 10.x, Opera 10.x and lower, Opera 11.x, Safari 4.x, Safari 5.x

**Server:** Support: Tomcat 5.x, Tomcat 6.x, Tomcat 7.0.26 and lower, Tomcat 7.0.27 and up, Jetty, GlassFish 2.x, GlassFish 3.x to 3.1.1, GlassFish 3.1.2

* **Advantages**: Only one persistent connection is opened. This is the Comet technique that saves the most bandwidth usage.
* **Disadvantage**: The multi-part flag is not supported by all browsers. Some widely used libraries, such as CometD in Java, reported issues in buffering. For example, chunks of data (multi-parts) may be buffered and sent only when the connection is completed or the buffer is full, which can create higher latency than expected.

### Comet using HTTP long polling

Creates connection to server like AJAX does, but keep-alive connection open for some time (not long though), during connection open client can receive data from server. Client have to reconnect periodically after connection is closed due to timeouts. On server side it still treated like HTTP request same as AJAX.



**Client:**

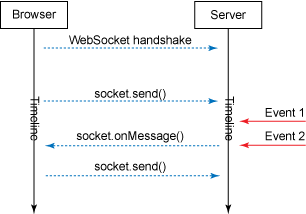
Support: Firefox 3.x to 8.x, Firefox 9.x to 11.x, Chrome 12.x and lower, Chrome 13.x and higher, IE 6.x to 9.x, IE 10.x, Opera 10.x and lower, Opera 11.x, Safari 4.x, Safari 5.x

**Server:**

Support: Tomcat 5.x, Tomcat 6.x, Tomcat 7.0.26 and lower, Tomcat 7.0.27 and up, Jetty, GlassFish 2.x, GlassFish 3.x to 3.1.1, GlassFish 3.1.2

* **Advantages**: It's easy to implement on the client side with a good error-handling system and timeout management. This reliable technique also allows a round-trip between connections on the server side, since connections are not persistent (a good thing, when you have a lot of clients on your application). It also works on all browsers; you only make use of theXMLHttpRequest object by issuing a simple Ajax request.
* **Disadvantages**: There is no main disadvantage compared to other techniques. But, like all techniques we've discussed, this one still relies on a stateless HTTP connection, which requires special features on the server side to be able to temporarily suspend it.

### Websockets



**WebSockets,** which emerged in HTML5, is a much more recent Reverse Ajax technique than Comet. WebSockets enables bi-directional, full-duplex communication channels, and many browsers (Firefox, Google Chrome, and Safari) already support it. The connection is opened through an HTTP request, called a WebSockets handshake, with some special headers. The connection is kept alive, and you can write and receive data in JavaScript as if you were using a raw TCP socket.

A WebSocket URL is started by typing *ws://* or *wss://* (on SSL).

**Client:** Support: Firefox, Google Chrome, Safari, IE10, Opera

WebSockets is a very powerful way to implement a bi-directional communication with no latency. It is supported by Firefox, Google Chrome, Opera, and other modern browsers.

**Server:** Support: Tomcat, Jetty.

**Advantages:** WebSockets provides powerful, bi-directional, low-latency, and easy-to-handle errors. There isn't a lot of connection, like Comet long polling, and it doesn't have the drawbacks of Comet streaming. The API is also very easy to use directly without any additional layers, compared to Comet, which requires a good library to handle reconnection, timeout, Ajax requests, acknowledgments, and the optionally different transports (Ajax long polling and jsonp polling).

**Disadvantages:** Drawbacks of WebSockets include:

* It is a new specification from HTML5, so it isn't yet supported by all browsers.
* No request scope. Since WebSockets is a TCP socket and not an HTTP request, request-scoped services, like Hibernate'sSessionInViewFilter, cannot be used easily. Hibernate is a persistence framework that provides a filter to surround an HTTP request. When the request begins, it sets up a contest (containing transactions and JDBC connection) bound to the request thread. When the request finishes, the filter destroys this contest.
* Every new technology comes with a new set of problems. In the case of WebSocket it is the compatibility with proxy servers which mediate HTTP connections in most company networks. The WebSocket protocol uses the HTTP upgrade system (which is normally used for HTTP/SSL) to "upgrade" an HTTP connection to a WebSocket connection. Some proxy servers do not like this and will drop the connection. Thus, even if a given client uses the WebSocket protocol, it may not be possible to establish a connection

### HTML5 Server sent event

**Client:** Support: Firefox 9.x to 12.x, Chrome 12.x and lower, Chrome 13.x and higher

**Server:** Support: Tomcat 5.x, Tomcat 6.x, Tomcat 7.0.26 and lower, Tomcat 7.0.27 and up, Jetty.

**Advantages:** Server-Sent Events are real-time events emitted by the server and received by the browser. They’re similar to WebSockets in that they happen in real time, but they’re very much a one-way communication method *from* the server.

These events are similar to ordinary JavaScript events that occur in the browser — like *click* events — except we can control the name of the event and the data associated with it

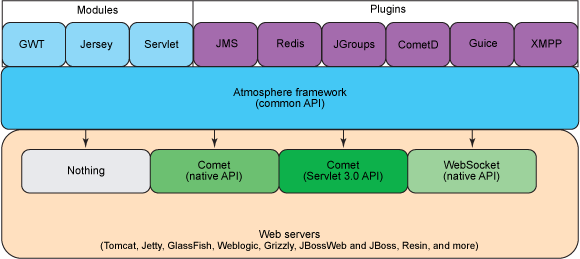
**Disadvantages:** HTML5 Server-Sent Events (SSE) are getting more and more adopted and support for it starts to appear. As an example, the GlassFish Application Server recently added support for it, the upcoming release of the Jersey Framework is also adding some sort of support, and framework like jQuery-Socket has sample supporting SSE as well. Both GlassFish and Jersey suffer **major issues**: First, you need to use non portable API to start using SSE (will only work in GlassFish or Jersey) and second, they expose special API to support SSE, which is a major mistake in my opinion. Just take a look at how simple it can be to implement SSE using the jQuery-Socket sample. Why would you use heavyweight API like GlassFish or Jersey to achieve something that simple? Not only that, but currently Internet Explorer isn't supporting SSE, so if you use either use GlassFish or Jersey, your application will NOT WORK with Internet Explorer. Oups!!!

# Atmosphere framework

## Introduction

In the aboce, we 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. The following, we learn about Atmosphere - one of the best open source Reverse Ajax libraries for Java technology servers.

Atmosphere is a Java technolog 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.



*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 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 server side

### AtmoSpringControllerResolver

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.

Config web.xml

<servlet>

<servlet-name>nms</servlet-name>

<servlet-class>org.atmosphere.cpr.MeteorServlet</servlet-class>

<init-param>

<param-name>org.atmosphere.servlet</param-name>

<param-value>org.springframework.web.servlet.DispatcherServlet</param-value>

</init-param>

<!-- Bunch of Atmosphere specific properties -->

<init-param>

<param-name>org.atmosphere.cpr.broadcasterClass</param-name>

<param-value>org.atmosphere.cpr.DefaultBroadcaster</param-value>

</init-param>

<init-param>

<param-name>org.atmosphere.useNative</param-name>

<param-value>true</param-value>

</init-param>

<init-param>

<param-name>org.atmosphere.useWebSocket</param-name>

<param-value>true</param-value>

</init-param>

<init-param>

<param-name>org.atmosphere.useStream</param-name>

<param-value>true</param-value>

</init-param>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>WEB-INF/nms-servlet.xml</param-value>

</init-param>

<init-param>

<param-name>org.atmosphere.useBlocking</param-name>

<param-value>true</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>nms</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

Config app-servlet.xml

<mvc:annotation-driven>

<mvc:argument-resolvers>

<beans:bean class=*"com.tma.support.AtmoSpringControllerResolver"*></beans:bean>

</mvc:argument-resolvers>

</mvc:annotation-driven>

Class **AtmoSpringControllerResolver**

**public** **class** AtmoSpringControllerResolver **implements**

HandlerMethodArgumentResolver {

@Override

**public** **boolean** supportsParameter(MethodParameter parameter) {

**return** AtmosphereResource.**class**.isAssignableFrom(parameter

.getParameterType());

}

@Override

**public** Object resolveArgument(MethodParameter parameter,

ModelAndViewContainer mavContainer,

NativeWebRequest webRequest,

WebDataBinderFactory binderFactory) **throws** Exception {

**return** Meteor.*build*(

webRequest.getNativeRequest(HttpServletRequest.**class**))

.getAtmosphereResource();

}

}

**HandlerMethodArgumentResolver**: Strategy interface for resolving method parameters into argument values in the context of a given request.

**Meteor:** is a simple class that can be used from a javax.servlet.Servlet to suspend, broadcast and resume a response. A Meteor can be created by invoking the build() method.

*Meteor.build(HttpServletRequest).suspend(-1);*

A Meteor is usually created when an application need to suspend a response. A Meteor instance can then be cached and re-used later for either broadcasting a message, or when an application needs to resume the suspended response.

### AtmosphereResource

The **AtmosphereResource** is the central concept of the Atmosphere Framework. An AtmosphereResource represents a remote connection. You can also see an AtmosphereResource as a communication channel between a browser and an application. An application uses an AtmosphereResource to handle the life cycle of the connection. For example, to tell the Atmosphere Framework to not close its connection and leave it open for later use, call method *suspend().*

An AtmosphereResource gets delivered to an **AtmosphereHandler**, and created every time a new connection is made.

An AtmosphereResource is always associated with one or several channels of communication (read Broadcaster). When created, a Broadcaster is always associated with an AtmosphereResource and can be retrieved using method *getBroadcaster().*

A Broadcaster always initiates a broadcast operation. It can be seen as channel of communication. An application can create many communication channels and retrieve them using the BroadcasterFactory class. An AtmosphereResource is always associated with one or several Broadcaster.You can subscribe to a channel, or Broadcaster, by adding an AtmosphereResource to it.

*broadcaster.addAtmosphereResource(atmosphereResource);*

Once added, messages or events will be delivered to an AtmosphereResource every time Broadcaster.broadcast gets invoked. An AtmosphereResource is associated with an AtmosphereResourceEvent, which always contains the last broadcasted events.

**Conclusion,** there is a remote connection between the server and client. That is AtmosphereResource. The server side suspend (not close connection) leave it open for later use. The broadcaster add an AtmosphereResource. Once added, messages or events will be delivered to an AtmosphereResource every time Broadcaster.broadcast gets invoked.

### AtmosphereHandler

The **AtmosphereHandler** is the central concept of an Atmosphere Framework application. AtmosphereHandler are the lowest component an application can implement. For example, the atmosphere-jersey and atmosphere-gwt build on top of AtmosphereHandler. The interface is defined as:

**public** **interface** **AtmosphereHandler** **{**

**void** **onRequest(**AtmosphereResource resource**)** **throws** IOException**;**

**void** **onStateChange(**AtmosphereResourceEvent event**)** **throws** IOException**;**

**void** **destroy();**

**}**

#### onRequest

The AtmosphereHandler.onRequest is invoked every time a new connection is made to an application. An application must take action and decide what to do with the AtmosphereResource, e.g. suspend, resume or broadcast events. You can also write String or bytes back to the client from that method.

#### onStateChange

This method gets invoked when:

* The remote connection gets closed, either by a browser or a proxy
* The remote connection reach its maximum idle time (AtmosphereResource.suspend))
* Everytime a broadcast operation is executed (broadcaster.broadcast)

#### destroy

When the Atmosphere Framework is stopped.

## The client side

For the client we are using the atmosphere.js jQuery plugin. The javascript looks like:

<script type=*"text/javascript"*>

$(**function**() {

**var** content = $('#content');

//The first things we need to so is to get a pointer to atmosphere.js main

element.For the chat we will call it socket.

**var** socket = $.atmosphere;

**var** subsocket;

**var** transport = 'websocket';

**var** websocketUrl = "${pageContext.request.contextPath}/test/events/";

//Here we define the request we want to make. As the preferred transport we will

favor WebSocket and fall-back to long-polling in case the server or client does

not support WebSocket.

**var** request = {

url : websocketUrl,

//contentType : "application/json",

logLevel : 'debug',

transport : transport,

fallbackTransport : 'long-polling',

//Here we are defining a function that will be invoked when the connection is

opened and ready to execute request. The onOpen function will be invoked once

the server suspend operation successfully completed.

onOpen : **function**(response) {

console.log('Atmosphere onOpen: Atmosphere connected using'

+ response.transport);

content.html($('<p>', { text: 'Atmosphere connected using '

+ response.transport }));

transport = response.transport;

$('#transportType').html(response.transport);

},

//Here we are defining a function that will be invoked

when a Broadcast operation occurs on the server.

onMessage : **function** onMessage(response) {

console.log("Receive response form server");

**var** result = response.responseBody;

**var** jsonobject;

**try** {

//The JSON message will be parsed

jsonobject = $.parseJSON(result);

} **catch** (e) {

console.log("An error ocurred while parsing the JSON Data: "

+ "; Error: " + e);

**return**;

}

console.log(jsonobject);

//Displayed on the screen.

content.append('<p> ' + jsonobject.id+'|'+

jsonobject.message+'|'+

jsonobject.severity+'|'+

jsonobject.deviceId+'|'+

jsonobject.portId+'|'+ '</p>');

},

//Here we are defining a function that will be invoked when reconnect

onReconnect : **function**(request, response) {

console.log("Atmosphere onReconnect: Reconnecting");

},

//Here we are defining a function that will be invoked when the connection

is closed

onClose : **function**(response) {

console.log('Atmosphere onClose executed');

},

//Here we are defining a function that will be invoked in case an error occurs

(the connection is dropped, the server goes down, etc.).

onError : **function**(response) {

console.log('Atmosphere onError: Sorry, but there is some problem with your '

+ 'socket or the server is down');

}

};

//Init socket

subSocket = socket.subscribe(request);

});

</script>

## Demo

1. Dependency in “pom.xml”

<dependency>

<groupId>org.atmosphere</groupId>

<artifactId>atmosphere-runtime</artifactId>

<version>1.0.16</version>

</dependency>

1. Import jquery

[*http://cdnjs.cloudflare.com/ajax/libs/jquery.atmosphere/2.0.0.RC2/jquery.atmosphere.min.js*](http://cdnjs.cloudflare.com/ajax/libs/jquery.atmosphere/2.0.0.RC2/jquery.atmosphere.min.js)

[*http://cdnjs.cloudflare.com/ajax/libs/jquery/1.7.2/jquery.min.js*](http://cdnjs.cloudflare.com/ajax/libs/jquery/1.7.2/jquery.min.js)

*http://ajax.microsoft.com/ajax/jquery.templates/beta1/jquery.tmpl.js*

1. Config Web.xml.

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:applicationContext.xml</param-value>

</context-param>

<listener>

<listener-class>

org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener>

<listener>

<listener-class>

org.springframework.web.context.request.RequestContextListener

</listener-class>

</listener>

<!-- MeteorServlet -->

<servlet>

<servlet-name>nms</servlet-name>

<servlet-class>org.atmosphere.cpr.MeteorServlet</servlet-class>

<init-param>

<param-name>org.atmosphere.servlet</param-name>

<param-value>org.springframework.web.servlet.DispatcherServlet</param-value>

</init-param>

<!-- Bunch of Atmosphere specific properties -->

<init-param>

<param-name>org.atmosphere.cpr.broadcasterClass</param-name>

<param-value>org.atmosphere.cpr.DefaultBroadcaster</param-value>

</init-param>

<init-param>

<param-name>org.atmosphere.useNative</param-name>

<param-value>true</param-value>

</init-param>

<init-param>

<param-name>org.atmosphere.useWebSocket</param-name>

<param-value>true</param-value>

</init-param>

<init-param>

<param-name>org.atmosphere.useStream</param-name>

<param-value>true</param-value>

</init-param>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>WEB-INF/nms-servlet.xml</param-value>

</init-param>

<init-param>

<param-name>org.atmosphere.useBlocking</param-name>

<param-value>true</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>nms</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

1. nms-servlet.xml

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:beans=*"http://www.springframework.org/schema/beans"*

xmlns:mvc=*"http://www.springframework.org/schema/mvc"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:p=*"http://www.springframework.org/schema/p"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xsi:schemaLocation=*"*

*http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.1.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-3.1.xsd*

*http://www.springframework.org/schema/mvc*

*http://www.springframework.org/schema/mvc/spring-mvc-3.1.xsd"*>

<context:component-scan base-package=*"com.tma.controller"* />

<mvc:annotation-driven>

<mvc:argument-resolvers>

<beans:bean class=*"com.tma.support.AtmoSpringControllerResolver"*>

</beans:bean>

</mvc:argument-resolvers>

</mvc:annotation-driven>

<mvc:resources mapping=*"/css/\*\*"* location=*"/css/"* />

<mvc:resources mapping=*"/js/\*\*"* location=*"/js/"* />

<bean id=*"viewResolver"*

class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>

<property name=*"prefix"* value=*"/WEB-INF/"* />

<property name=*"suffix"* value=*".jsp"* />

</bean>

</beans>

1. Create package “com.tma.support”
2. Create class AtmoSpringControllerResolver in package “com.tma.support”

**public** **class** AtmoSpringControllerResolver **implements**

HandlerMethodArgumentResolver {

@Override

**public** **boolean** supportsParameter(MethodParameter parameter) {

**return** AtmosphereResource.**class**.isAssignableFrom(parameter

.getParameterType());

}

@Override

**public** Object resolveArgument(MethodParameter parameter,

ModelAndViewContainer mavContainer,

NativeWebRequest webRequest,

WebDataBinderFactory binderFactory) **throws** Exception {

**return** Meteor.*build*(

webRequest.getNativeRequest(HttpServletRequest.**class**))

.getAtmosphereResource();

}

}

1. Create class controller “EvenRSrc”

@Controller

@RequestMapping(value = "/test")

**public** **class** EventRSrc {

**private** **final** Logger logger = Logger.*getLogger*(EventRSrc.**class**);

**private** ObjectMapper mapper = **new** ObjectMapper();

**public** **static** **final** String *EVENT\_BROADCAST* = "/events";

@RequestMapping(value = "/", method = RequestMethod.*GET*)

**public** String test() {

**return** "home";

}

@RequestMapping(value = "/events", method = RequestMethod.*GET*)

@ResponseBody

**public** **void** registerClientGet(AtmosphereResource atmosphereResource) {

atmosphereResource.suspend();

Broadcaster broadcaster = BroadcasterFactory.*getDefault*().lookup(

*EVENT\_BROADCAST*, **false**);

**if** (broadcaster == **null**) {

logger.info("Create broadcaster /events");

broadcaster = BroadcasterFactory.*getDefault*().get(*EVENT\_BROADCAST*);

}

logger.info("Add client to broadcaster :" + atmosphereResource.uuid());

broadcaster.addAtmosphereResource(atmosphereResource);

}

// Insert alarm

@RequestMapping(value = "/alarms/", method = RequestMethod.*POST*)

@ResponseBody

**public** **void** insertAlarm(@RequestBody Alarm alarm) **throws** Exception {

AlarmDAO alarmDAO = AppContext.*getService*(AlarmDAO.**class**, "alarmDAO");

alarmDAO.insertAlarm(alarm);

logger.info("Insert alarm success");

// Broadcaster

BroadcasterFactory bF = BroadcasterFactory.*getDefault*();

**if** (bF != **null**) {

Broadcaster broadcaster = bF.lookup(EventRSrc.*EVENT\_BROADCAST*,

**false**);

**if** (broadcaster != **null**) {

broadcaster.broadcast(mapper.writeValueAsString(alarm));

logger.info("Broadcast alarm");

}

}

}

}

1. Create “home.jsp”.

<html>

<head>

<meta charset=*"utf-8"*>

<title>Atmosphere WebSocket Chat</title>

<script type=*"text/javascript"* src=*"/AtmoDemo/js/jquery/jquery.js"*></script>

<script type=*"text/javascript"*

src=*"/AtmoDemo/js/jquery/jquery.tmpl.min.js"*></script>

<script type=*"text/javascript"*

src=*"/AtmoDemo/js/jquery/jquery.atmosphere.js"*></script>

<style>\* {font-family: *tahoma*;font-size: *12px*;padding: *0px*;margin: *0px*;}

**p** { line-height: *18px*;}

**div** {width: *500px*;margin-left: *auto*;margin-right: *auto*;}

*#content* {padding: *5px*;background: *#ddd*;border-radius: *5px*;border: *1px solid #CCC*;margin-top: *10px*;}

*#header* {padding: *5px*;background: *#f5deb3*;border-radius: *5px*;border: *1px solid #CCC*;margin-top: *10px*;}

*#input* {border-radius: *2px*;border: *1px solid #ccc*;margin-top: *10px*;padding: *5px*;width: *400px*;}

*#status* {width: *88px*;display: *block*;float: *left*;margin-top: *15px*;}

</style>

</head>

<body>

<h2>Home</h2>

<div id=*"header"*>

<h3>Atmosphere Demo</h3>

</div>

<div id=*"stats"* class=*"prepend-1 span-4 append-1 prepend-top last"*>

<table id=*"asynchHttpStats"*>

<tbody>

<tr>

<td>Protocol:&nbsp;&nbsp;</td>

<td id=*"transportType"*>N/A</td>

</tr>

</tbody>

</table>

</div>

<div id=*"content"*></div>

<script type=*"text/javascript"*>

$(**function**() {

**var** content = $('#content');

//The first things we need to so is to get a pointer to atmosphere.js main element.

//For the chat we will call it socket.

**var** socket = $.atmosphere;

**var** subSocket;

**var** transport = 'websocket';

**var** websocketUrl = "${pageContext.request.contextPath}/test/events/";

//Here we define the request we want to make

//As the preferred transport we will favor WebSocket

//and fall-back to long-polling in case the server or client does not support WebSocket.

**var** request = {

url : websocketUrl,

//contentType : "application/json",

logLevel : 'debug',

transport : transport,

fallbackTransport : 'long-polling',

//Here we are defining a function that will be invoked

//when the connection is opened and ready to execute request.

//The onOpen function will be invoked once the server suspend operation successfully completed.

onOpen : **function**(response) {

console

.log('Atmosphere onOpen: Atmosphere connected using '

+ response.transport);

content.html($('<p>', {

text : 'Atmosphere connected using '

+ response.transport

}));

transport = response.transport;

$('#transportType').html(response.transport);

},

//Here we are defining a function that will be invoked

//when a Broadcast operation occurs on the server.

onMessage : **function** onMessage(response) {

console.log("Receive response form server");

**var** result = response.responseBody;

**var** jsonobject;

**try** {

//The JSON message will be parsed

jsonobject = $.parseJSON(result);

} **catch** (e) {

console

.log("An error ocurred while parsing the JSON Data: "

+ "; Error: " + e);

**return**;

}

console.log(jsonobject);

//Displayed on the screen.

content.append('<p> ' + jsonobject.id + '|'

+ jsonobject.message + '|' + jsonobject.severity

+ '|' + jsonobject.deviceId + '|'

+ jsonobject.portId + '|' + '</p>');

},

//Here we are defining a function that will be invoked when reconnect

onReconnect : **function**(request, response) {

console.log("Atmosphere onReconnect: Reconnecting");

},

//Here we are defining a function that will be invoked when the connection is closed

onClose : **function**(response) {

console.log('Atmosphere onClose executed');

},

//Here we are defining a function that will be invoked

//in case an error occurs (the connection is dropped, the server goes down, etc.).

onError : **function**(response) {

console

.log('Atmosphere onError: Sorry, but there is some problem with your '

+ 'socket or the server is down');

}

};

//Init socket

subSocket = socket.subscribe(request);

});

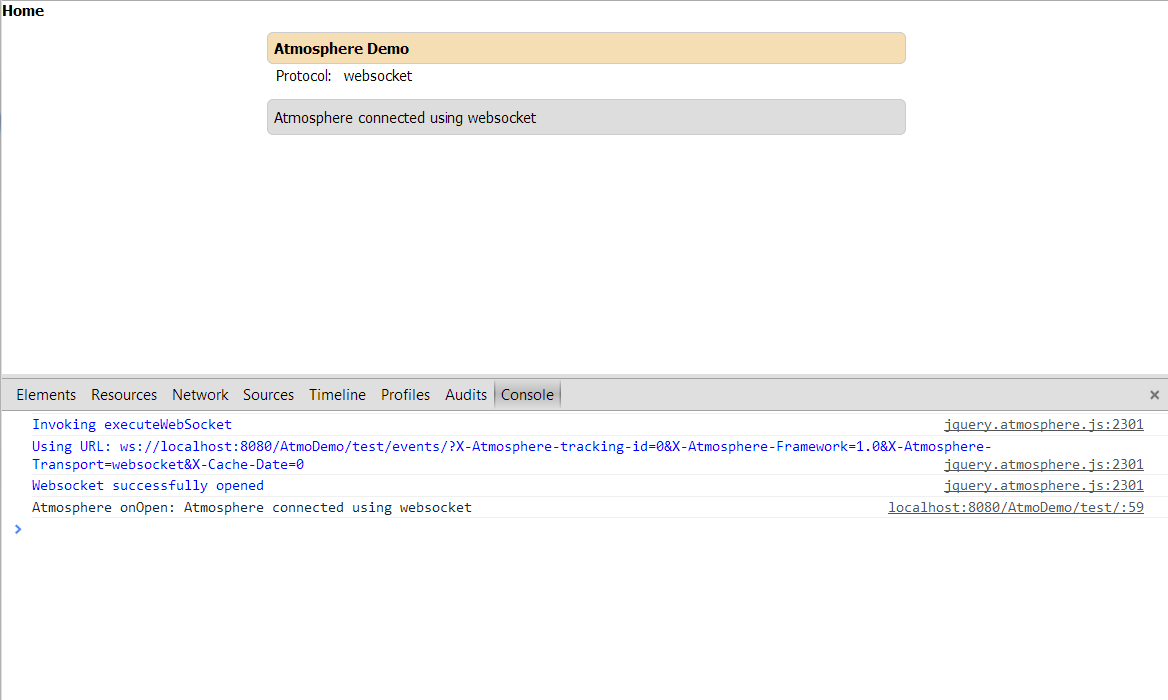
</script>

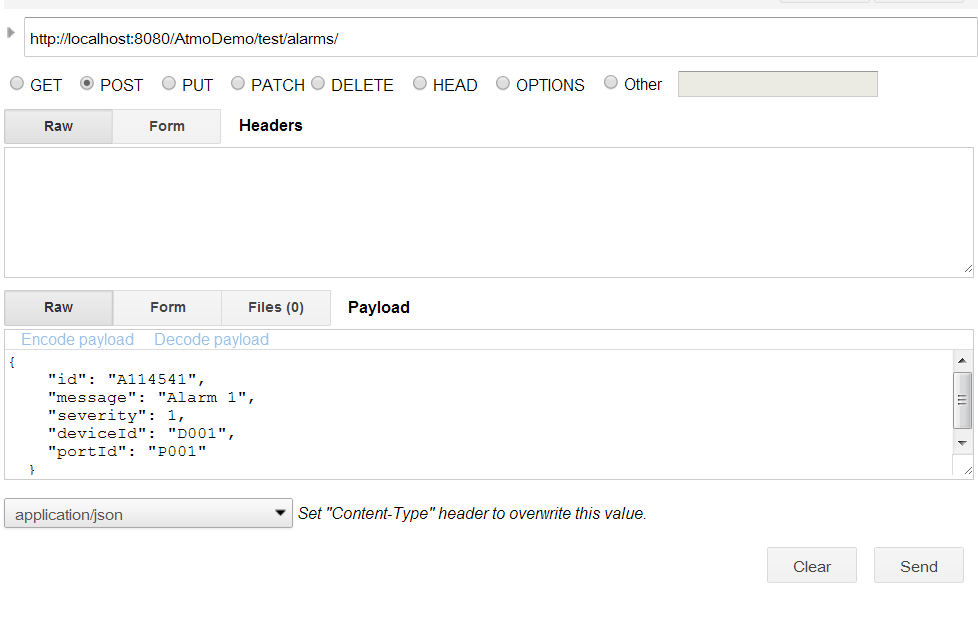
</body>

</html>

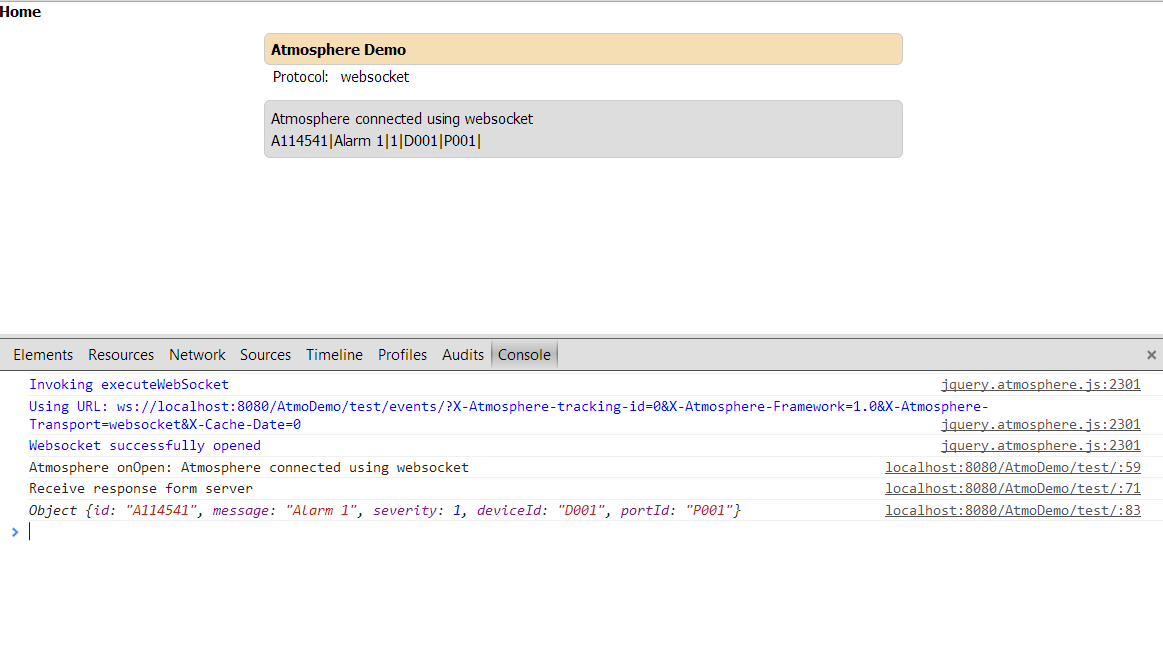
1. Run server & test

<http://localhost:8080/AtmoDemo/test/>



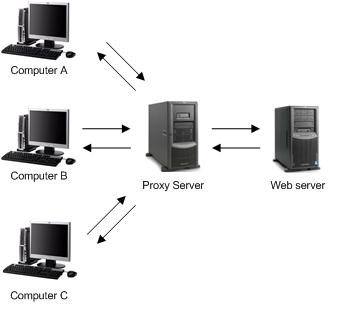
Insert Alarms: 

Result:



# Proxy server

A proxy server is a specific type of application server that routes HTTP requests to content servers that perform the work. You can classify a proxy server according to the role that it plays in a system. This specific proxy server is classified as a reverse proxy server because the main function is to act as the first point of contact, not including the firewall, for client requests into the enterprise server. By contrast, a forward proxy server acts as the first point of contact for outbound traffic.



## Use proxy server Nginx

* 1. Download nginx and unrar to C:\

<http://nginx.org/en/download.html>

* 1. Configure “nginx.conf”

#user nobody;

worker\_processes 1;

#error\_log logs/error.log;

#error\_log logs/error.log notice;

#error\_log logs/error.log info;

#pid logs/nginx.pid;

events {

worker\_connections 1024;

}

http {

include mime.types;

default\_type application/octet-stream;

#log\_format main '$remote\_addr - $remote\_user [$time\_local] "$request" '

# '$status $body\_bytes\_sent "$http\_referer" '

# '"$http\_user\_agent" "$http\_x\_forwarded\_for"';

#access\_log logs/access.log main;

sendfile on;

#tcp\_nopush on;

#keepalive\_timeout 0;

keepalive\_timeout 65;

#gzip on;

server {

listen localhost:80;

root html/localhost;

index index.php index.html;

charset utf-8;

location ~ /\. {deny all;}

location / {

}

location /AtmoDemo/test/ {

proxy\_pass http://localhost:8080/AtmoDemo/test/;

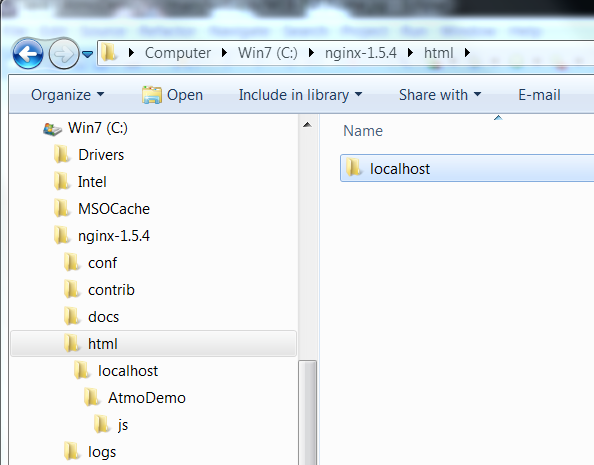
}

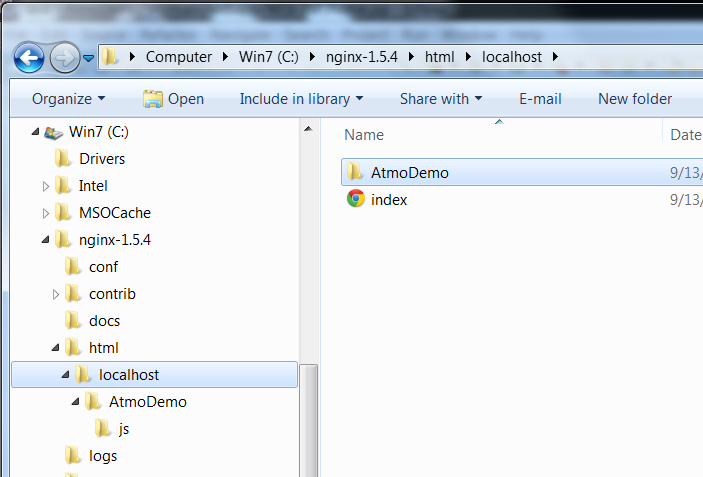
}

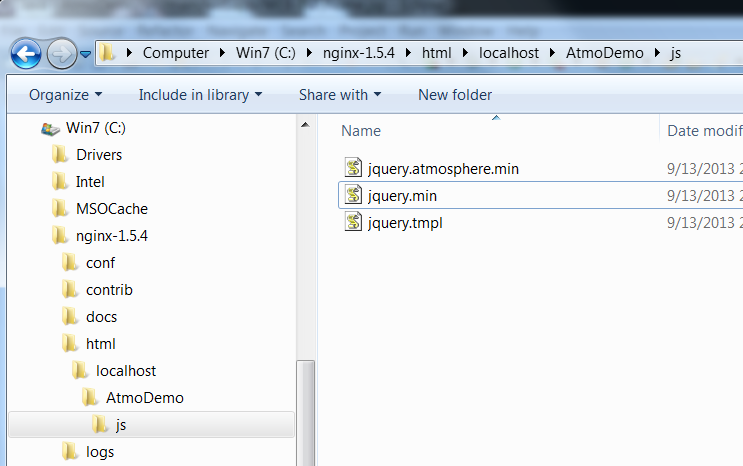
}

* 1. Create new folder “localhost” in “html”, new folder “AtmoDemo” in “localhost”, new folder “js” in “AtmoDemo”.

Download jquery.asmosphere.min.js, jquery.min.js, jquery.tmpl.js to folder “js”







* 1. Edit home.jsp: add scripts

<head>

<meta charset=*"utf-8"*>

<title>Atmosphere WebSocket Chat</title>

<script type=*"text/javascript"* src=*"/AtmoDemo/js/jquery/jquery.js"*></script>

<script type=*"text/javascript"*

src=*"/AtmoDemo/js/jquery/jquery.tmpl.min.js"*></script>

<script type=*"text/javascript"*

src=*"/AtmoDemo/js/jquery/jquery.atmosphere.js"*></script>

<script type=*"text/javascript"* src=*"/AtmoDemo/js/jquery.min.js"*></script>

<script type=*"text/javascript"* src=*"/AtmoDemo/js/jquery.tmpl.js"*></script>

<script type=*"text/javascript"* src=*"/AtmoDemo/js/jquery.atmosphere.min.js"*></script>

<style>\* {font-family: *tahoma*;font-size: *12px*;padding: *0px*;margin: *0px*;}

**p** { line-height: *18px*;}

**div** {width: *500px*;margin-left: *auto*;margin-right: *auto*;}

*#content* {padding: *5px*;background: *#ddd*;border-radius: *5px*;border: *1px solid #CCC*;margin-top: *10px*;}

*#header* {padding: *5px*;background: *#f5deb3*;border-radius: *5px*;border: *1px solid #CCC*;margin-top: *10px*;}

*#input* {border-radius: *2px*;border: *1px solid #ccc*;margin-top: *10px*;padding: *5px*;width: *400px*;}

*#status* {width: *88px*;display: *block*;float: *left*;margin-top: *15px*;}

</style>

</head>

* 1. Edit “server.xml” of tomcat server

<Connector connectionTimeout=*"20000"*

port=*"8080"*

protocol=*"HTTP/1.1"*

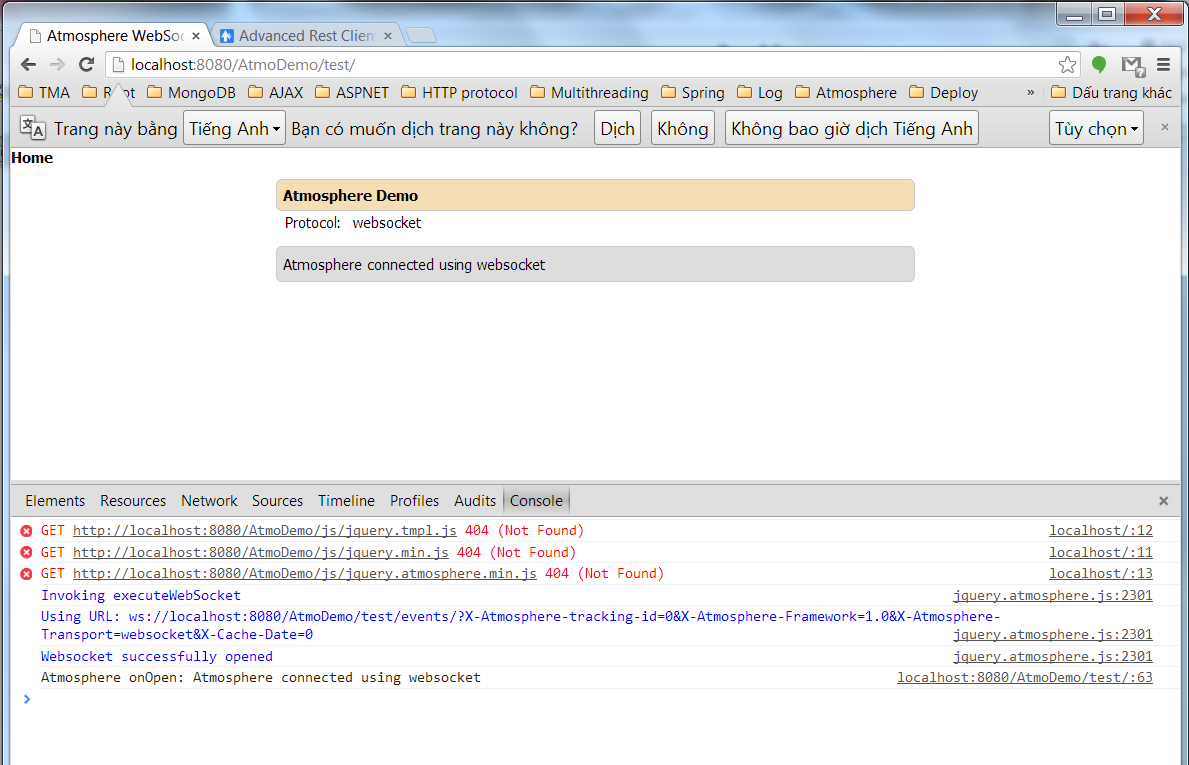
proxyName=*"localhost"*

proxyPort=*"80"*

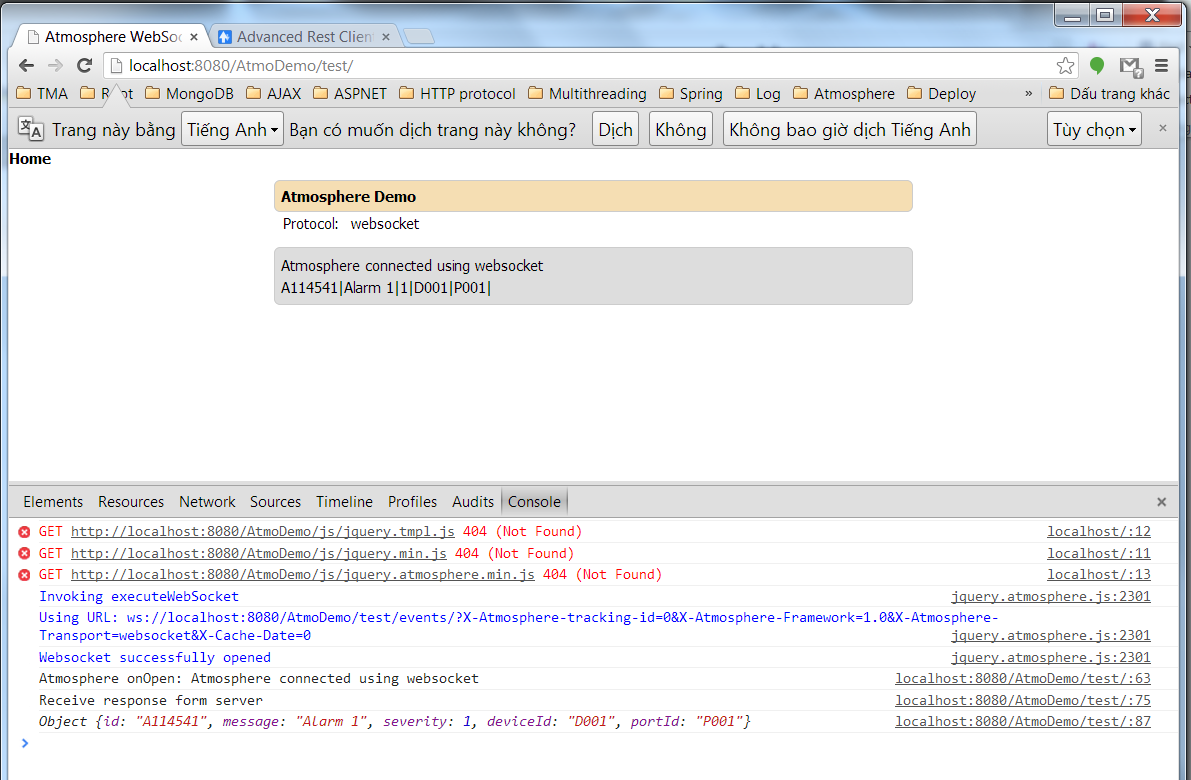
redirectPort=*"8443"*/>

* 1. Run tomcat server

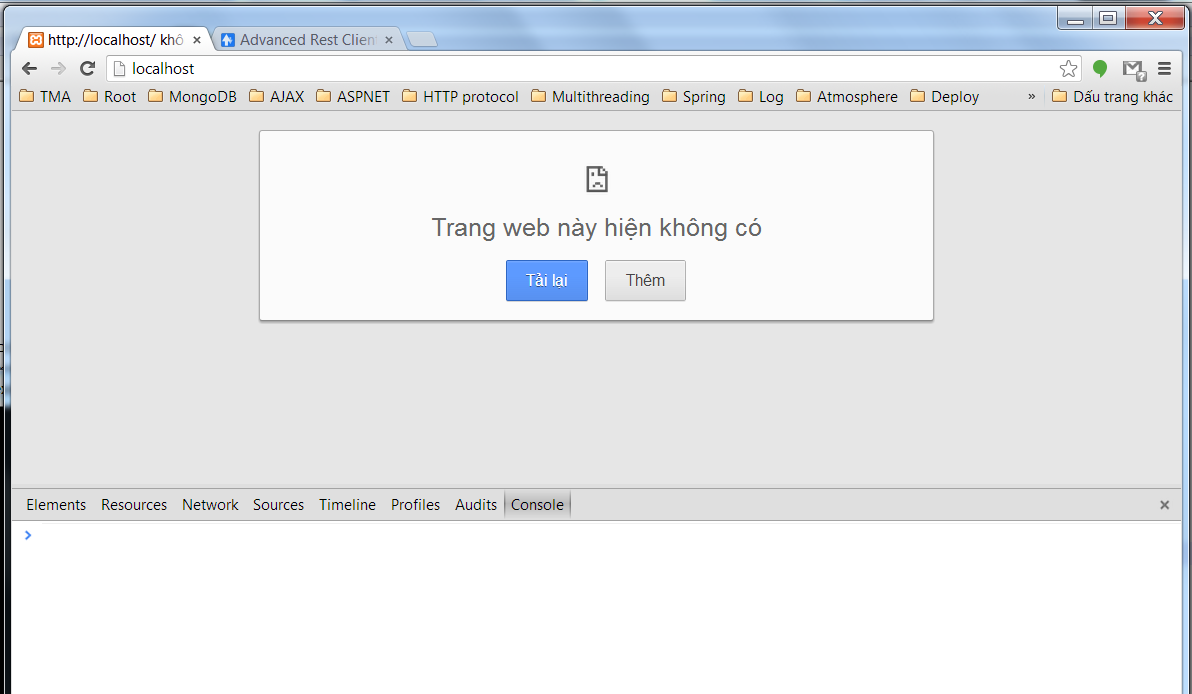
<http://localhost:8080/AtmoDemo/test/>



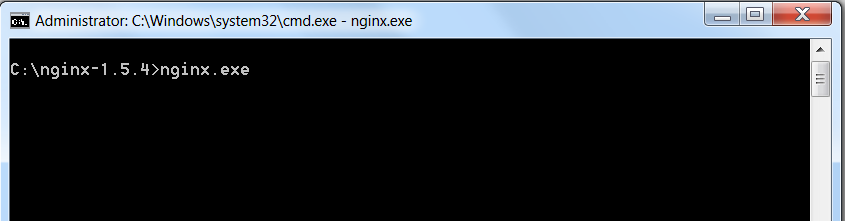
Error because javascripts not load. Don’t problem. Websocket until run ok.



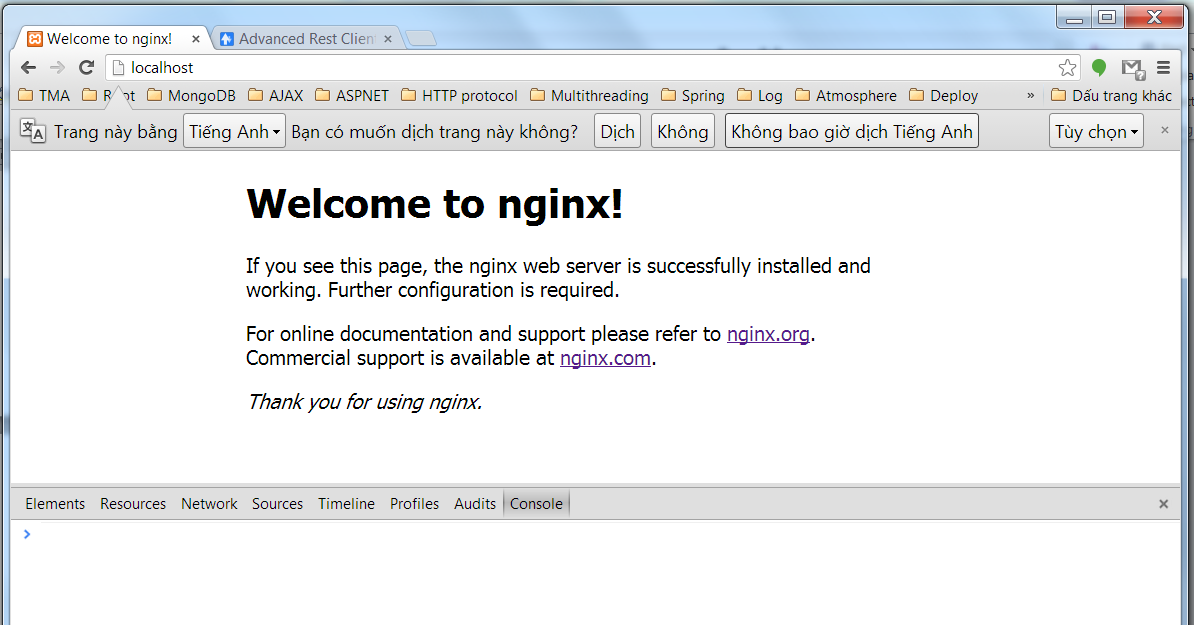
Server sent event ok.

<http://localhost/>

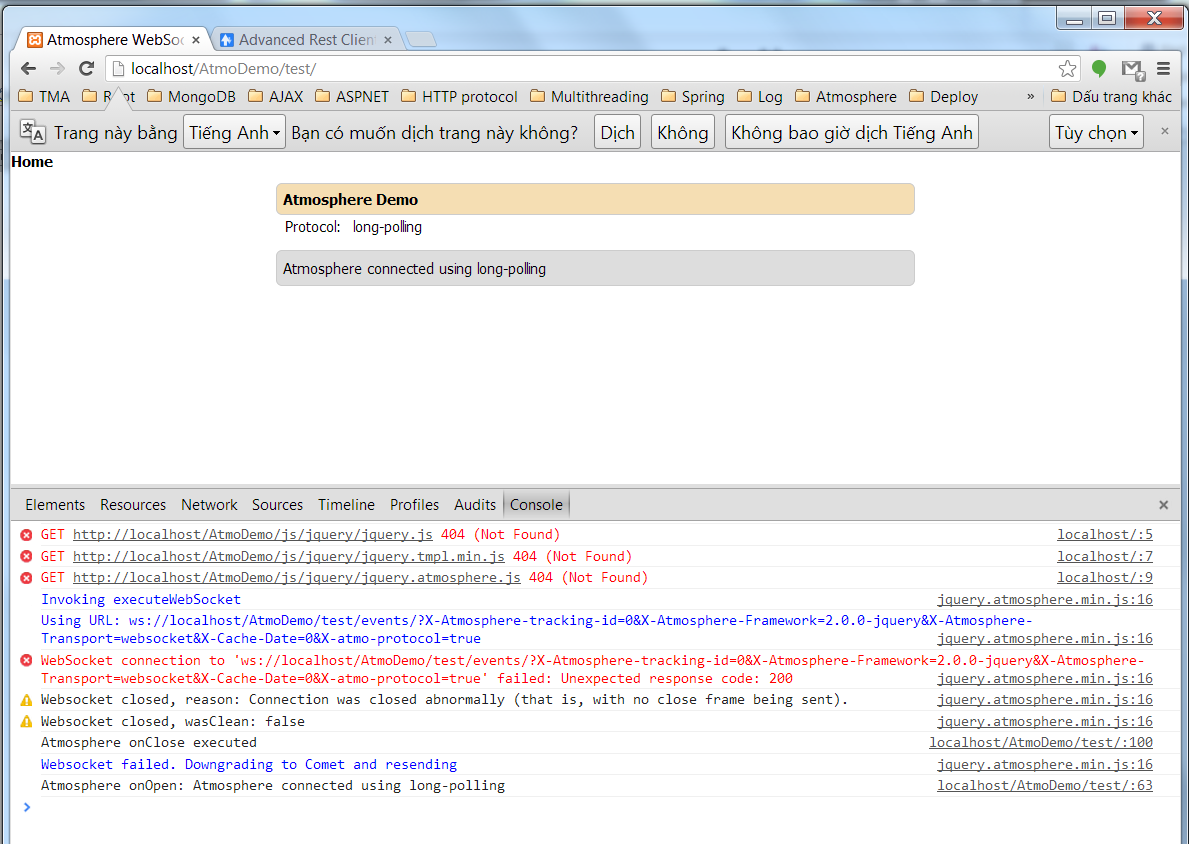
* 1. Start proxy server



<http://localhost/>



<http://localhost:80/AtmoDemo/test/>



## Problem

The problem was that WebSocket server was running on port 5000. It would work when connecting to it from localhost.

ws://localhost:8080

But when we wanted to push it to production and try it on my company’s network it wouldn’t work!

Obviously that was to be expected.

Our company’s network only allows a couple of ports through their firewall. Obviously port 80 was one of them and port 8080 was not.

Now the problem is that server was already running Nginx on port 80. So obviously WebSocket server couldn’t run on port 80.

## Solution

All we had to do was configure Nginx to start proxying WebSockets from port 80 (external) to port 8080 (internal).

Configure “nginx.conf”:

#user nobody;

worker\_processes 1;

#error\_log logs/error.log;

#error\_log logs/error.log notice;

#error\_log logs/error.log info;

#pid logs/nginx.pid;

events {

worker\_connections 1024;

}

http {

include mime.types;

default\_type application/octet-stream;

#log\_format main '$remote\_addr - $remote\_user [$time\_local] "$request" '

# '$status $body\_bytes\_sent "$http\_referer" '

# '"$http\_user\_agent" "$http\_x\_forwarded\_for"';

#access\_log logs/access.log main;

sendfile on;

#tcp\_nopush on;

#keepalive\_timeout 0;

keepalive\_timeout 65;

#gzip on;

server {

listen localhost:80;

root html/localhost;

index index.php index.html;

charset utf-8;

location ~ /\. {deny all;}

location / {

}

location /AtmoDemo/test/ {

proxy\_pass http://localhost:8080/AtmoDemo/test/;

proxy\_http\_version 1.1;

proxy\_set\_header Upgrade $http\_upgrade;

proxy\_set\_header Connection "upgrade";

}

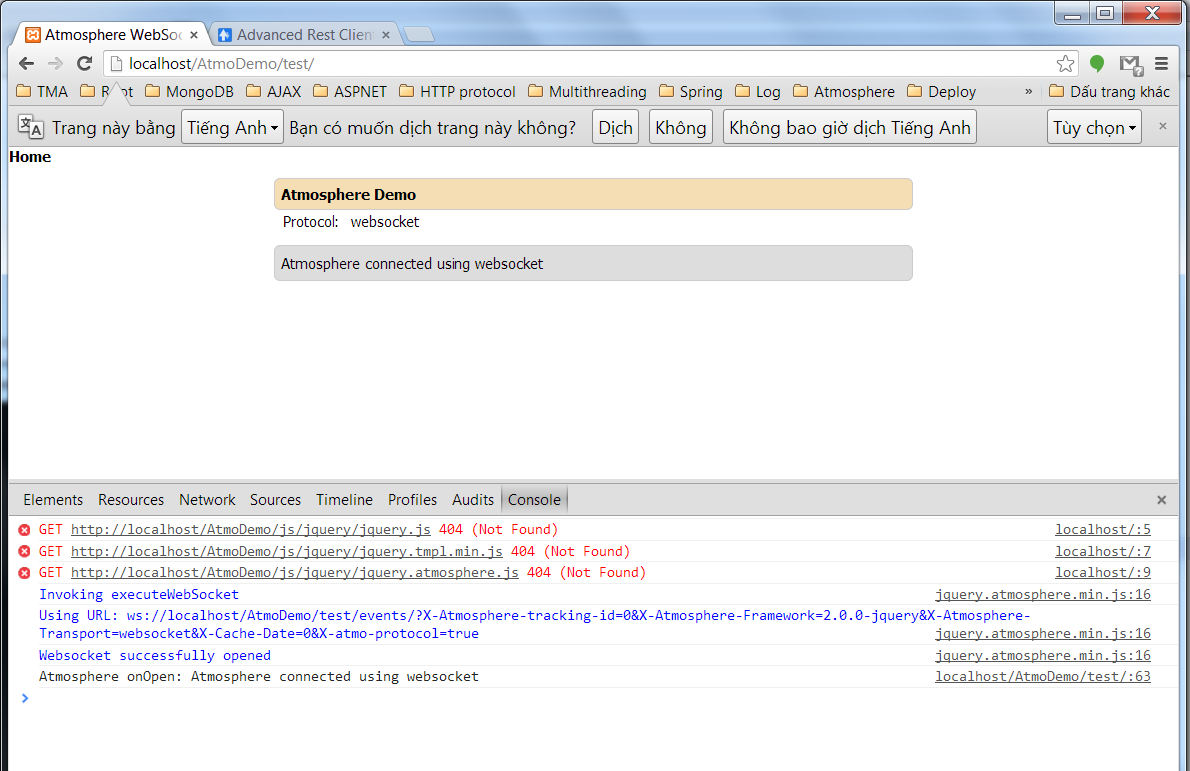
}

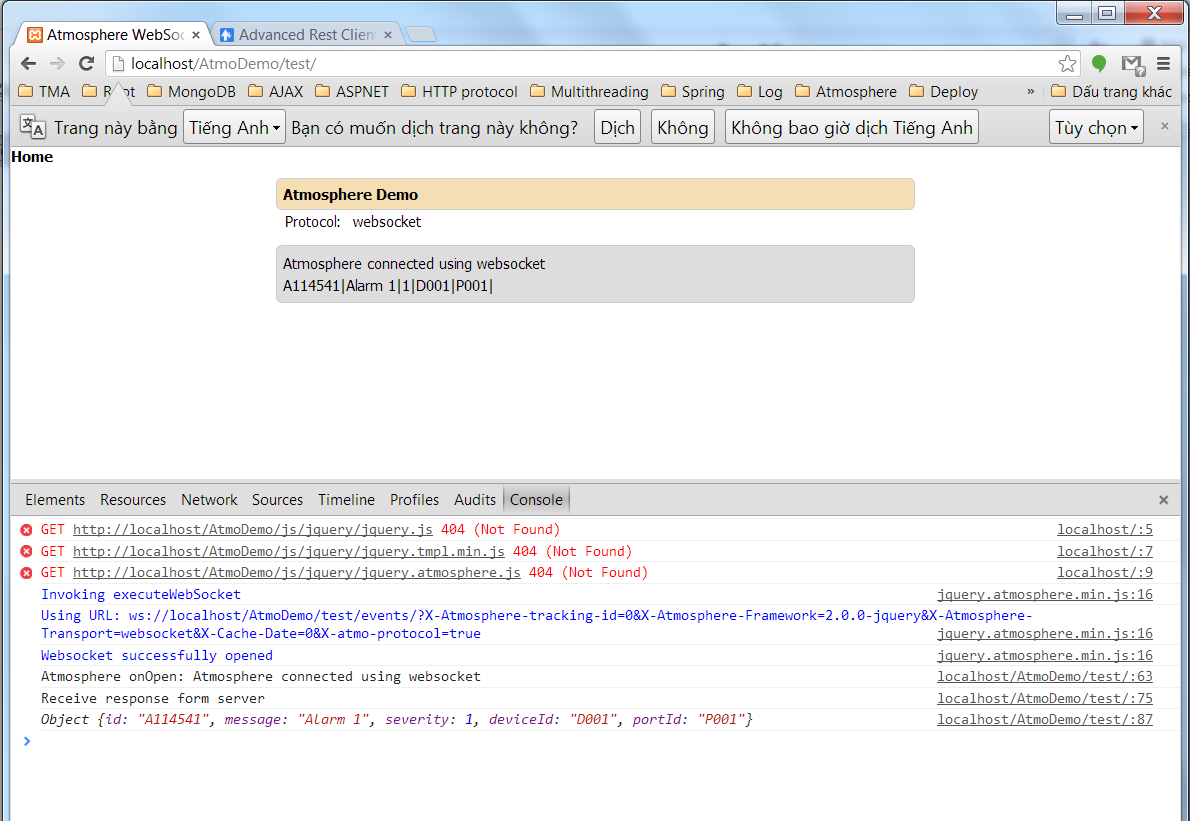
}

Run again:

Notice: To stop nginx proxy server, you “end process” nginx in “Windows Task Manager”

<http://localhost:80/AtmoDemo/test/>





# References

1. <http://www.ibm.com/developerworks/web/library/wa-reverseajax1/index.html>
2. <http://www.ibm.com/developerworks/web/library/wa-reverseajax2/index.html>
3. <http://www.html5rocks.com/en/tutorials/eventsource/basics/>
4. <http://html5doctor.com/server-sent-events/>
5. <https://github.com/Atmosphere/atmosphere/wiki/Getting-started-with-HTML5-Server-Sent-Events-(SSE)>
6. <http://jfarcand.wordpress.com/2012/04/19/websockets-or-comet-or-both-whats-supported-in-the-java-ee-land/>
7. <https://github.com/Atmosphere/atmosphere/wiki/Supported-WebServers-and-Browsers>
8. <https://github.com/Atmosphere/atmosphere/wiki/Understanding-AtmosphereHandler>
9. <http://michieldemey.be/blog/proxying-websockets-with-nginx-and-socket-io/>
10. <http://en.wikipedia.org/wiki/Proxy_server>
11. <https://library.linode.com/web-servers/nginx/configuration/basic>
12. [http://docs.spring.io/spring/docs/3.1.x/javadoc-api/org/springframework  
    /web/method/support/HandlerMethodArgumentResolver.html](http://docs.spring.io/spring/docs/3.1.x/javadoc-api/org/springframework/web/method/support/HandlerMethodArgumentResolver.html)