Developing with Comet and Java
Implementing the Servlet 3.0 specification

Michael Galpin
May 26, 2009

Explore the different implementations of developing with Comet. See how popular Java™ Web servers like Jetty and Tomcat have enabled Comet applications, and learn how to program with each server. And finally, learn about the standardization proposals for Comet in Java that are part of the upcoming Servlet 3.0 and JavaEE 6 specifications.

Getting started

In this article I will show you how to build some simple Comet-style Web applications using a variety of Java technologies. Some light knowledge of Java servlets, Ajax, and JavaScript will be useful. We will look at the Comet-enabling features of both Tomcat and Jetty, so you will want the latest versions of each. Tomcat 6.0.14 and Jetty 6.1.14 were used in this article. You will also need a Java Development Kit supporting Java 5 or higher. JDK 1.5.0-16 was used in this article. You also might want to look at a pre-release version of Jetty 7, as it implements the Servlet 3.0 specification that we will examine in this article. See Related topics for download links.

Understanding Comet

You may have heard of Comet, given its recent share of buzz. Comet is sometimes called reverse Ajax or server-side push. The idea is pretty simple: push data from the server to the browser without the browser requesting it. This sounds simple, but if you are familiar with Web applications, and in particular the HTTP protocol, then you know it is anything but simple. Implementing Comet-style Web applications in a way that scales on both the browser and the server is something that has only become generally feasible in recent years. Later in this article we will look at how some popular Java Web servers enable scalable Comet architectures, but first let's look at why you would create Comet applications and the general design patterns used to implement them.

Motivations for using Comet

It is hard to argue with the success of the HTTP protocol. It is the foundation of the bulk of the information exchange on the Internet. However, it has several limitations. In particular, it is a
stateless, one-way protocol. A request is sent to a Web server, and that server turns around and issues a response—end of story. The request has to be initiated by the client, and the server can only send data in response to a request. This can make a number of types of Web applications a bit impractical to say the least. The classic example is a chat program. Other examples are sports scores, stock tickers, or e-mail programs.

The limitations of HTTP are also responsible for some of its success. The request/response cycle lends itself to the classic one thread per connection model. As long as you can service the request quickly, this approach can scale massively. A huge number of requests per second can be handled, and a relatively low number of servers can handle a very large number of users. This is a perfect fit for many of the classic types of Web applications such as content management systems, search applications, and e-commerce sites, just to name a few. In each case, the server provides the data that the user requests and can then close the connection, allowing that thread to be freed up to serve other requests. If interaction was possible after the initial data was served, and the connection was kept open, then the thread could not be released and the server would not be able to serve many users.

But what if you want to keep interacting with the user after the initial data is sent in response to a request? In the early days of the Web, this used to be accomplished using a meta refresh. This would automatically instruct the browser to reload the page after a specified number of seconds, thus enabling a crude form of polling. This is not only a bad user experience, but it is generally a terribly inefficient way of doing things. What if there is no new data to show on the page? The exact same page just gets re-rendered. What if the change to the page is very small, and most of the page does not change? Again, everything gets re-requested and re-drawn, if it needs to be or not.

The advent and popularity of Ajax changed the above scenario. Now the server can be contacted asynchronously, thus the entire page does not have to be re-requested. An incremental update is now possible. All you need to do is poll the server using an XMLHttpRequest. This technique is commonly known as Comet. There are several variations on this technique, each with different performance and scalability characteristics. Let's take a look at these different styles of Comet.

Comet styles

The viability of Ajax made Comet possible. The one-way nature of HTTP could be effectively worked around. As it turns out there are actually several different ways to pull this off. As you have probably guessed, the easiest way to enable Comet is to poll. You make a call to your server using XMLHttpRequest, and as soon as it comes back, wait a fixed amount of time (usually using JavaScript's setTimeout function), and then call it again. This is a very common technique. For example, this is how most webmail applications show you new e-mail messages when they arrive.

There are some pros and cons to this technique. In this setup, you expect the response to come back quickly, like any other Ajax request. Having the pause in between requests like this is essential. Otherwise you would bombard your server with constant requests, and clearly that would not scale. This pause introduces a latency in your application. The longer your pause, the longer it takes for new data from the server to arrive on the client. If you shorten up the pause,
then you run back into the risk of overwhelming your server. On the other hand, this is definitely the simplest way to implement Comet.

Now, it should be pointed out that many people do not consider polling to be Comet. Instead they consider Comet to be the solutions to the limitations of polling. The most common “true” Comet technique is a variant of polling called long polling. The main difference between polling and long polling is how long it takes the server to respond. A long poll generally keeps the connection open for a long time—usually several seconds, but it could be a minute or longer. When an event happens on the server, the response is sent and closed, and the poll immediately begins anew.

The advantage of a long poll over normal polling is that data goes from the server to the client as soon as it is available. The request may wait a long time with nothing to send back, but once there is something new, it is immediately sent to the client. There is no latency. If you have used a Web-based chat program, or anything that claims to be "real-time", it probably used this technique.

There is a variation of the long poll, a third flavor of Comet that is worth mentioning. This is usually referred to as streaming. In this flavor, the server pushes data back to the client, but does not close the connection. The connection stays open until it times out and causes the request to get re-initiated. The XMLHttpRequest specification states that you can check for a readyState of 3 or Receiving (as opposed to the usual readyState of 4 or Loaded) and get the data that is "streaming" from the server. This is similar to long polling in that there is no latency. When the data is ready on the server, it is sent to the client. It has an added advantage of making far fewer requests to the server, thus avoiding the overhead and latency associated with setting up the connection to the server. Unfortunately, there is great variance in the implementation of XMLHttpRequest across browsers. This technique only works reliably on the newer versions of Mozilla Firefox. For Internet Explorer or Safari, you need to stick to long polling.

At this point you might be thinking that there is a major problem with both long polling and streaming. The request stays alive on the server for a long time. This breaks the one thread per request model, as the thread for a request never gets freed up. Even worse, this thread will be sitting idle until there is data to send back. This definitely will not scale. Luckily, modern Java Web servers have ways of addressing this.

**Comet on Java**

There have always been a lot of Web servers built with Java. One reason for this is that Java has a rich, native thread model. Thus it is relatively straightforward to implement the classic one thread per connection model. This model does not work so well for Comet, but fortunately Java once again has a solution. To handle Comet efficiently, you need non-blocking IO, and Java has this via its NIO library. Two of the most popular open source servers, Apache Tomcat and Jetty, have both leveraged NIO to add non-blocking IO and thus enable Comet. However, the implementations are quite different. Let's take a look at both Tomcat and Jetty's support for Comet.

**Tomcat and Comet**

For Apache Tomcat, there are two major things that you need to do to get Comet working. First, you need to make a small change in Tomcat's configuration file, server.xml. The more typical
synchronous IO connector is enabled by default. You just need to switch this to the asynchronous version as shown in Listing 1.

**Listing 1. Modify Tomcat's server.xml**

<!-- This is the usual Connector, comment it out and add the NIO one -->
<!-- Connector URIEncoding="utf-8" connectionTimeout="20000" port="8084" -->
<!-- protocol="HTTP/1.1" redirectPort="8443" -->

This allows for Tomcat to handle many more simultaneous connections, with the caveat that most of those connections are going to be idle much of the time. The easiest way to take advantage of this is to create a servlet that implements the `org.apache.catalina.CometProcessor` interface. This is obviously an interface unique to Tomcat. An example of this is shown in Listing 2.

**Listing 2. Tomcat Comet servlet**

```java
public class TomcatWeatherServlet extends HttpServlet implements CometProcessor {
    private MessageSender messageSender = null;
    private static final Integer TIMEOUT = 60 * 1000;

    @Override
    public void destroy() {
        messageSender.stop();
        messageSender = null;
    }

    @Override
    public void init() throws ServletException {
        messageSender = new MessageSender();
        Thread messageSenderThread =
            new Thread(messageSender, "MessageSender[" + getServletContext().getContextPath() + "]");
        messageSenderThread.setDaemon(true);
        messageSenderThread.start();
    }

    public void event(final CometEvent event) throws IOException, ServletException {
        HttpServletRequest request = event.getHttpServletRequest();
        HttpServletResponse response = event.getHttpServletResponse();
        if (event.getEventType() == CometEvent.EventType.BEGIN) {
            request.setAttribute("org.apache.tomcat.comet.timeout", TIMEOUT);
            log("Begin for session: " + request.getSession(true).getId());
            messageSender.setConnection(response);
            Weatherman weatherman = new Weatherman(95118, 32408);
            new Thread(weatherman).start();
        } else if (event.getEventType() == CometEvent.EventType.ERROR) {
            log("Error for session: " + request.getSession(true).getId());
            event.close();
        } else if (event.getEventType() == CometEvent.EventType.END) {
            log("End for session: " + request.getSession(true).getId());
            event.close();
        } else if (event.getEventType() == CometEvent.EventType.READ) {
            throw new UnsupportedOperationException("This servlet does not accept data");
        }
    }
}
```
The CometProcessor interface requires you to implement the event method. This is a lifecycle method for a Comet interaction. Tomcat will invoke with different CometEvent instances. We check the eventType of the CometEvent to determine where in the lifecycle we are. A BEGIN event happens when the request first comes in. A READ event indicates data being sent in, and is only needed if the request was a POST. The request can terminate in either an END or ERROR event.

In the example in Listing 2, the servlet uses a class called MessageSender to send in data. This is set up in its own thread during the init method of the servlet and taken down in the destroy method of the servlet. The MessageSender is shown in Listing 3.

Listing 3. The MessageSender

```java
private class MessageSender implements Runnable {

    protected boolean running = true;
    protected final ArrayList<String> messages = new ArrayList<String>();
    private ServletResponse connection;
    private synchronized void setConnection(ServletResponse connection)
    {
        this.connection = connection;
        notify();
    }
    public void send(String message) {
        synchronized (messages) {
            messages.add(message);
            log("Message added #messages=" + messages.size());
            messages.notify();
        }
    }
    public void run() {
        while (running) {
            try {
                synchronized (messages) {
                    messages.wait();
                }
            } catch (InterruptedException e) {
                // Ignore
            }
            String[] pendingMessages = null;
            synchronized (messages) {
                pendingMessages = messages.toArray(new String[0]);
                messages.clear();
            }
            try {
                if (connection == null) {
                    try{
                        synchronized(this)
                        {
                        wait();
                        }
                    } catch (InterruptedException e){
                        // Ignore
                    }
                    PrintWriter writer = connection.getWriter();
                    for (int j = 0; j < pendingMessages.length; j++) {
                        final String forecast = pendingMessages[j] + "<br>";
                        writer.println(forecast);
                        log("Writing:" + forecast);
                    }
                    writer.flush();
                    writer.close();
                    connection = null;
                }
            }
```

This class is mostly boilerplate code, not directly relevant to Comet. There are a couple of things to notice, though. The class takes a ServletResponse object. If you look back at Listing 2 in the event method, when the event is a BEGIN, the response object is passed in to the MessageSender. In the MessageSender's run method, it uses the ServletResponse to send data back to the client. Notice that once it has sent all of the queued up messages, it closes the connection. Thus this implements a long poll. If you wanted to implement a streaming style of Comet, you would want to keep the connection open, but still flush the data.

If you look back at Listing 2, you will see that there is a weatherman class being created. This class is what uses the MessageSender to send data back to the client. It is a class that uses a Yahoo RSS feed to get weather information about various zip codes and sends this to the client. This is a contrived example designed to simulate a data source that sends data in an asynchronous manner. Its code is shown in Listing 4.

**Listing 4. The Weatherman**

```java
private class Weatherman implements Runnable{
    private final List<URL> zipCodes;
    private final String YAHOO_WEATHER = "http://weather.yahooapis.com/forecastRSS?p=";

    public Weatherman(Integer... zips) {
        zipCodes = new ArrayList<URL>(zips.length);
        for (Integer zip : zips) {
            try {
                zipCodes.add(new URL(YAHOO_WEATHER + zip));
            } catch (Exception e) {
                // dont add it if it sucks
            }
        }
    }

    public void run() {
        int i = 0;
        while (i >= 0) {
            int j = i % zipCodes.size();
            SyndFeedInput input = new SyndFeedInput();
            try {
                SyndFeed feed = input.build(new InputStreamReader(zipCodes.get(j).openStream()));
                SyndEntry entry = (SyndEntry) feed.getEntries().get(0);
                messageSender.send(entryToHtml(entry));
                Thread.sleep(30000L);
            } catch (Exception e) {
                // just eat it, eat it
            }
            i++;
        }
    }

    private String entryToHtml(SyndEntry entry){
        StringBuilder html = new StringBuilder("<h2>");
    }
}
```
This class uses the Project Rome library for parsing the RSS feed from Yahoo Weather. If you need to produce or consume RSS or Atom feeds, this is a very useful library. The only other point of interest in this code is that it spawns another thread that sends in weather data every 30 seconds. Finally, we have one last thing to look at: the client code for using this servlet. In this case a simple JSP with a small amount of JavaScript is sufficient. This is shown in Listing 5.

Listing 5. Client Comet code

```html
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/html4/loose.dtd">

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Comet Weather</title>
<SCRIPT TYPE="text/javascript">
function go(){
    var url = "http://localhost:8484/WeatherServer/Weather"
    var request = new XMLHttpRequest();
    request.open("GET", url, true);
    request.setRequestHeader("Content-Type","application/x-javascript;");
    request.onreadystatechange = function() {
        if (request.readyState == 4) {
            if (request.status == 200){
                if (request.responseText) {
                    document.getElementById("forecasts").innerHTML = request.responseText;
                }
                go();
            }
        }
    request.send(null);
    }
</SCRIPT>
</head>
<body>
<h1>Rapid Fire Weather</h1>
<input type="button" onclick="go()" value="Go!">
<div id="forecasts"></div>
</body>
</html>
```

This code simply starts the long poll when the user clicks the Go button. Notice that it uses the XMLHttpRequest object directly, so this will not work in Internet Explorer 6. You probably want to use an Ajax library to smooth over the browser differences. The only other important thing to note is the callback function, or, the closure created for the request's onreadystatechange function. This function pastes in the new data from the server and then re-invokes the go function.

Now we have seen what a simple Comet application looks like on Tomcat. There were two very Tomcat-centric things we had to do: configure its connector and implement a Tomcat specific
interface in the servlet. So you may be wondering just how difficult it would be to "port" this code to Jetty. Let's take a look at that next.

Jetty and Comet

The Jetty server uses a slightly different technique to enable a scalable implementation of Comet. Jetty supports the programming construct known as continuations. The idea is simple enough. A request is suspended and continued at some point in the future. The resumption could happen either because of a timeout or some other, meaningful event. While the request is suspended, its thread is freed up.

You can use Jetty's org.mortbay.util.ajax.ContinuationSupport class to create an instance of org.mortbay.util.ajax.Continuation for any HttpServletRequest. This allows for a very different approach to Comet. However, continuations can be used to implement a logically identical style of Comet. Listing 6 shows the weather servlet from Listing 2 after it has been "ported" to Jetty.

Listing 6. Jetty Comet servlet

```java
public class JettyWeatherServlet extends HttpServlet {
    private MessageSender messageSender = null;
    private static final Integer TIMEOUT = 5 * 1000;
    public void begin(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
        request.setAttribute("org.apache.tomcat.comet", Boolean.TRUE);
        request.setAttribute("org.apache.tomcat.comet.timeout", TIMEOUT);
        messageSender.setConnection(response);
        Weatherman weatherman = new Weatherman(95118, 32408);
        new Thread(weatherman).start();
    }
    public void end(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
        synchronized (request) {
            request.removeAttribute("org.apache.tomcat.comet");
            Continuation continuation = ContinuationSupport.getContinuation
            (request, request);
            if (continuation.isPending()) {
                continuation.resume();
            }
        }
    }
    public void error(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
        end(request, response);
    }
    public boolean read(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
        throw new UnsupportedOperationException();
    }
    @Override
    protected void service(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
        synchronized (request) {
            Continuation continuation = ContinuationSupport.getContinuation
            (request, request);
            if (!continuation.isPending()) {
                begin(request, response);
            }
            Integer timeout = (Integer) request.getAttribute
            ("org.apache.tomcat.comet.timeout");
            boolean resumed = continuation.suspend(timeout == null ? 10000 :
```
The most important thing to notice here is how the structure mimics the Tomcat version of the code. The begin, read, end, and error methods match up to the same events in Tomcat. The servlet's service method is overridden to create a continuation when the request first comes in and suspends it until either the timeout is hit or another event causes it to resume. The init and destroy methods are not shown above because they are identical to the Tomcat version. This servlet uses the same MessageSender as the Tomcat version. No modification is needed. Notice how the begin method creates a Weatherman instance. That class is also used exactly as in the Tomcat version. Even the client code is identical. Only the servlet is changed at all. Its changes are significant, but with a straightforward mapping back to the event model in Tomcat.

I hope this is encouraging. The exact same code does not work in both Tomcat and Jetty, but it is very similar. Of course one of the appeals of JavaEE is portability. Most code that runs in Tomcat will run in Jetty with no modification and vice versa. Thus it should come as no surprise that the next version of the Java Servlet specification includes a standardization of asynchronous request processing, or, the underlying technology behind Comet. Let's take a look at that spec, the Servlet 3.0 spec.

**Servlet 3.0 spec**

We could dive into all of the gnarly details of the Servlet 3.0 specification. Instead let's just take a look at what the Comet servlet might look like if it was running inside a Servlet 3.0 container. Notice the word "might." The specification has been released for public review, but has not been finalized as of the time of this writing. Thus Listing 7 shows an implementation compliant with the public review specification.

**Listing 7. Servlet 3.0 Comet**

```java
@WebServlet(asyncSupported=true, asyncTimeout=5000)
public class WeatherServlet extends HttpServlet {
    private MessageSender messageSender;

    // init and destroy are the same as other
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        AsyncContext async = request.startAsync(request, response);
        messageSender.setConnection(async);
        Weatherman weatherman = new Weatherman(95118, 32444);
        async.start(weatherman);
    }
}
```
The nice thing here is that this is much simpler. In all fairness, a similar implementation is possible with Jetty, if we had not tried to adhere to the event model of Tomcat. The event model may have seemed sensible and could be easily implemented in containers other than Tomcat, like Jetty, but there will be no standardization around it.

Looking back at Listing 7, notice that its annotation declares that it supports asynchronous processing and sets the timeout. The startAsync method is a new method on HttpServletRequest, and it returns an instance of the new class javax.servlet.AsyncContext. Notice that the MessageSender is being passed a reference to the AsyncContext instead of the ServletResponse. Instead of closing the response, you should call the complete method on the AsyncContext instance. Also notice that the Weatherman was passed directly to the start method of the AsyncContext instance. This starts a new thread in the current ServletContext.

So, despite being significantly different than either Tomcat or Jetty, it is not too hard to adapt the same style of programming to work with the proposed APIs of the Servlet 3.0 specification. It should be noted that Jetty 7 is intended to implement Servlet 3.0 and is available in a beta form. However, as of the time of writing, it did not implement the latest version of the specification as shown above.

Summary

Comet-style Web applications can bring a whole new level of interactivity to the Web. It presents some complex challenges for implementing these features on a large scale. However, the leading Java Web servers are all providing mature, stable technology for implementing Comet. You have seen in this article the differences and similarities of the current flavors of Comet on Tomcat and Jetty, as well as the ongoing standardization effort for the Servlet 3.0 specification. Tomcat and Jetty make it possible to build scalable Comet applications today, with a clear upgrade path to standardization under Servlet 3.0 in the future.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weather Server Source Code</td>
<td>WeatherServer.zip</td>
<td>347KB</td>
</tr>
</tbody>
</table>
Related topics

- "The allure of Comet" (Erik Burkhart, developerWorks, November 2007): Find out more about the history and motivations behind Comet.
- "High-performance Ajax with Tomcat Advanced I/O" (Adriaan de Jonge, developerWorks, September 2008): Learn even more about Comet on Tomcat.
- "Ajax for Java Developers: Write Scalable Comet applications with Jetty and Direct Web Remoting" (Philip McCarthy, developerWorks, July 2007): Direct Web Remoting (DWR) can make Ajax a lot easier for Java developers, and it can do the same for Comet. Find out how in the developerWorks article.
- "Java EE meets Web 2.0" (Constantine Plotnikov et al., developerWorks, November 2007): Dive into the performance implications of Comet.
- "Developing Rich Internet Applications with WebSphere Application Server Community Edition" (Manu George and Vamsavardhana Reddy Chillakuru, developerWorks, September 2008): Read about how Comet works equally well in a full Java EE stack.
- Asynchronous HTTP Comet architectures: Check out Asynchronous HTTP Comet architectures for even more information on Comet in Java.
- An Introduction to Servlet 3.0: Read about all of the new features of Servlet 3.0.
- Check out the reference information about Tomcat's Advanced IO and Continuations in Jetty.
- "Creating mashups on the Google App Engine" (Michael Galpin, developerWorks, August 2008): See how you can use Python and the Prototype JavaScript library to create a Comet application that can be deployed to the Google App Engine.
- developerWorks podcasts: Listen to interesting interviews and discussions for software developers.
- JDK 1.5.0-16 was used in this article.
- Jetty: Jetty 6.1.14 was used in this article.
- Tomcat: Tomcat 6.0.14 was used in this article.