Continuations

> ⚠ Continuations will be replaced by standard Servlet-3.0 suspendable requests once the specification is finalized. Early releases of Jetty-7 are now available that implement the proposed standard suspend/resume API

Background

With most web applications today, the number of simultaneous users can greatly exceed the number of connections to the server. This is because connections can be closed during the frequent pauses in the conversation while the user reads the content or completes in a form. Thousands of users can be served with hundreds of connections.

But AJAX based web applications have very different traffic profiles to traditional webapps. While a user is filling out a form, AJAX requests to the server will be asking for for entry validation and completion support. While a user is reading content, AJAX requests may be issued to asynchronously obtain new or updated content. Thus an AJAX application needs a connection to the server almost continuously and it is no longer the case that the number of simultaneous users can greatly exceed the number of simultaneous TCP/IP connections.

If you want thousands of users you need thousands of connections and if you want tens of thousands of users, then you need tens of thousands of simultaneous connections. It is a challenge for java web containers to deal with significant numbers of connections, and you must look at your entire system, from your operating system, to your JVM, as well as your container implementation.

Thread per connection

One of the main challenges in building a scalable servlet server is how to handle Threads and Connections. The traditional IO model of java associates a thread with every TCP/IP connection. If you have a few very active threads, this model can scale to a very high number of requests per second.

However, the traffic profile typical of many web applications is many persistent HTTP connections that are mostly idle while users read pages or search for the next link to click. With such profiles, the thread-per-connection model can have problems scaling to the thousands of threads required to support thousands of users on large scale deployments.

Thread per request

The NIO libraries can help, as it allows asynchronous IO to be used and threads can be allocated to connections only when requests are being processed. When the connection is idle between requests, then the thread can be returned to a thread pool and the connection can be added to an NIO select set to detect new requests. This thread-per-request model allows much greater scaling of connections (users) at the expense of a reduced maximum requests per second for the server as a whole (in Jetty 6 this expense has been significantly reduced).

AJAX polling problem

But there is a new problem. The advent of AJAX as a web application model is significantly changing the traffic profile seen on the server side. Because AJAX servers cannot deliver asynchronous events to the client, the AJAX client must poll for events on the server. To avoid a busy polling loop, AJAX servers will often hold onto a poll request until either there is an event or a timeout occurs. Thus an idle AJAX application will have an outstanding request waiting on the server which can be used to send a response to the client the instant an asynchronous event

occurs. This is a great technique, but it breaks the thread-per-request model, because now every client will have a request outstanding in the server. Thus the server again needs to have one or more threads for every client and again there are problems scaling to thousands of simultaneous users.

	Formula	**Web 1.0**	**Web 2.0 + Comet**	**Web 2.0 + Comet + Continuations**
Users	u	10000	10000	10000
Requests/Burst	b	5	2	2
Burst period (s)	p	20	5	5
Request Duration (s)	d	0.200	0.150	0.175
Poll Duration (s)	D	0	10	10
Request rate (req/s)	rr=u*b/20	2500	4000	4000
Poll rate (req/s)	pr=u/d	0	1000	1000
Total (req/s)	r=rr+pr	2500	5000	5000
Concurrent requests	c=rr*d+pr*D	500	10600	10700
Min Threads	T=c T=r*d	500 -	10600 -	- 875
Stack memory	S=64*1024*T	32MB	694MB	57MB

Jetty 6 Continuations

The solution is Continuations, a new feature introduced in Jetty 6. A java Filter or Servlet that is handling an AJAX request, may now request a Continuation object that can be used to effectively suspend the request and free the current thread. The request is resumed after a timeout or immediately if the resume method is called on the Continuation object. In the Jetty 6 chat room demo, the following code handles the AJAX poll for events:

```
private void doPoll(HttpServletRequest
request, AjaxResponse response)
{
```

```
    HttpSession session =
request.getSession(true);

    synchronized (mutex)
    {
        Member member =
(Member)chatroom.get(session.getId());

        // Is there any chat events ready to
send?
        if (!member.hasEvents())
        {
            // No - so prepare a
continuation
            Continuation continuation =
ContinuationSupport.getContinuation(request,
mutex);

member.setContinuation(continuation);

            // wait for an event or timeout
            continuation.suspend(timeoutMS);
        }
        member.setContinuation(null);

        // send any events
```

```
            member.sendEvents(response);
        }
    }
```

So the request handling is "suspended" to wait for available chat events. When another user says something in the chat room, the event is delivered to each member by another thread calling the method:

```
class Member
    {
        // ...
        public void addEvent(Event event)
        {
            synchronized (mutex)
            {
                _events.add(event);
                if (getContinuation()!=null)

getContinuation().resume();
            }
        }
        // ...
    }
```

How it works

Behind the scenes, Jetty has to be a bit sneaky to work around Java and the Servlet specification as there is no mechanism in Java to suspend a thread and then resume it later. The first time the request handler calls continuation.suspend(timeoutMS) a RetryRequest runtime exception is thrown. This exception propagates out of all the request handling code and is caught by Jetty and handled specially. Instead of producing an error response, Jetty places the request on a timeout queue and returns the thread to the thread pool.

When the timeout expires, or if another thread calls continuation.resume(event) then the request is retried. This time, when continuation.suspend(timeoutMS) is called, either the event is returned or null is returned to indicate a

timeout. The request handler then produces a response as it normally would.

Thus this mechanism uses the stateless nature of HTTP request handling to simulate a suspend and resume. The runtime exception allows the thread to legally exit the request handler and any upstream filters/servlets plus any associated security context. The retry of the request, re-enters the filter/servlet chain and any security context and continues normal handling at the point of continuation.

Furthermore, the API of Continuations is portable. If it is run on a non-Jetty6 server it will simply use wait/notify to block the request in getEvent. If Continuations prove to work as well as I hope, I plan to propose them as part of the 3.0 Servlet JSR.