

Suspendable Requests

Asynchronous Servlets: Suspendable Requests.

Most non trivial web applications need to wait at some stage during the processing of a HTTP request, examples of waiting request handling include:

- waiting for a resource (eg. thread, JDBC Connection) to be available before processing the request.
- waiting for an application event in an [Ajax Comet](#) application (eg Chat message, price change, etc.)
- waiting for a response from a remote service (eg RESTful or SOAP call to a web service).

The current servlet API (<=2.5) supports only a synchronous call style, so that any waiting that a servlet needs to do must be with blocking. Unfortunately this means that the thread allocated to the request must be held during that wait with all its resources including kernel thread, stack memory and often pooled buffers, character converters, EE authentication context, etc. It is wasteful of system resources to be hold these resources while waiting.

Jetty's solution

Jetty provides support for Asynchronous Servlets by supporting Suspendable Requests, either via the Jetty [Continuations](#) or the proposed standard [servlet 3.0 API](#).

Blocking Waiting Examples

JDBC Connection Pool

Consider a web application handling on average 400 requests per second, with each request interacting with the database for 50ms. To handle this average load, $400 \times 50 / 1000 = 20$ JDBC connections are need on average. However, requests do not come at the even rate and there are often bursts and pauses. To protect a database from bursts, often a JDBC connection pool is applied to limit the simultaneous requests made on the database. So for this application, it would be reasonable to apply a JDBC pool of 30 connections, to provide for a 50% margin.

If momentarily the request rate doubled, then the 30 connections would only be able to handle 600 requests per second, and 200 requests per second would join those waiting on the JDBC Connection pool. If the servlet container had a thread pool with 200 threads, then that would be entirely consumed by threads waiting for JDBC connections in 1 seconds of this request rate. After 1s, the web application would be unable to process any requests at all because no threads would be available. Even requests that do not use the database would be blocked due to thread starvation. To double the thread pool would require an additional 100MB of stack memory and would only give the application another 1s of grace under load!.

This thread starvation situation can also occur if the database runs slowly or is momentarily unavailable. Thread starvation is a very frequently reported problem, that causes an entire web service to lock up and become unresponsive!

If the web container was able to threadlessly suspend the requests waiting for a JDBC connection, then thread starvation would not occur, as only 30 threads would be consumed by requests accessing the database and the other 470 threads would be available to process the request that do not access the database.

For an example of Jetty's solution, see the [Quality of Service Filter](#).

Remote Web Service

Consider a web application that accesses a remote web service (eg SOAP service or RESTful service). Typically a

remote web service can take hundreds of milliseconds to produce a response (eg ebays RESTful web service frequently takes 350ms to respond with a list of auctions matching a given keyword), while only a few 10s of milliseconds of CPU time are needed to locally process a request and generate a response.

To handle 400 requests per second, such a webapp would need $400 \cdot (350 + 20) / 1000 = 148$ threads on average, but would also be vulnerable to thread starvation if bursts occurred or the web service became slow.

If the web container was able to threadlessly suspend the request while waiting for the web service response, then even if 5ms of additional processing was required, the web app would need only $400 \cdot (20 + 5) / 1000 = 10$ threads on average and would not be vulnerable to thread starvation during bursts or if the web service is slow or unavailable. Moreover, the reduction of in the number of threads required would free up over 70MB of memory (unused stacks) that could be used for the application.

For an example of Jetty's solution, see the [Asynchronous Rest](#) example.

Ajax Comet Portfolio application

[Ajax Comet](#) web applications use the long polling technique so that a server may at any time send a response to a browser that is waiting for an application event. Consider a stock portfolio web application, each browser will send a long poll request to the server asking for any of the users stock prices that have changed their price. The server will process the long poll requests by waiting until a price changes and then immediately sending a response to the browser, at which time the browser will send a new long poll request to obtain the next price change.

If a price changes on average every 10s and the long poll response/request takes 100ms to transit the network and 20ms to process, then the request rate for 500 users would be $500 \cdot 1000 / (10 \cdot 1000 + 100 + 20) = 49.4$ requests per second, each taking $(10 \cdot 1000 + 100 + 20) = 10.12$ seconds, so that $49.4 \cdot 10.12 = 500$ threads are required.

If the web container was able to threadlessly suspend the request while waiting for the price change, then even if 10ms of additional processing was required, then only $49.4 \cdot (20 + 10) / 1000 = 2$ threads are needed! This can free up almost 250MB of stack memory for other use by the web application!

For an example of Jetty's solution, see the [Cometd \(aka Bayeux\)](#).

More Information

- [Async servlet use-cases](#)
- [Suspendable request patterns](#)