Performance and Clustering

Tapestry has a great **performance** story to tell. It's designed to take advantage of the speed of the modern JVM: no reflection, built to support a high level of concurrency without contention, and clean, lightweight code paths. In addition, there is built-in integrated GZIP content compression, far-future expires headers on static resources, JavaScript aggregation and minification, and an intentionally lightweight use of the HTTPSession. The result is a blistering fast framework. See Tapestry's Performance Tested for some objective numbers.

Related Articles

- Performance and Clustering
- Persistent Page Data
- Session Storage

Performance Tips

But even with all of Tapestry's built-in speediness, to really get top performance you'll need to be sure you're not hamstringing Tapestry. As a start, use the following checklist:

- Ensure (be absolutely sure) that Production Mode is turned on in production.
- Minimize the use of the HTTPSession (see below), especially if you're using clustering.
- Set tapestry.clustered-sessions to "false" if you aren't using clustering.
- Organize your JavaScript files into JavaScriptStacks.
- Ensure that your static resources (images, CSS, JavaScript) are being cached by the browser.
  - Use "asset:" or "context:" binding prefixes
  - for all links to static resources (images, CSS, JavaScript).
  - Make sure that your firewall, proxy server, load balancer, front-end web servers, and app servers all allow caching of static resources.
  - Ensure "cache-control" and "vary" HTTP headers are set correctly for your static resources.
  - Use a client-based tool (like Firebug) to examine the requests that your browser makes as you navigate through the site. You should *not* see repeated requests for static resources.
- Consider using a Content Delivery Network for static parts of your site.

After all of the above issues are addressed, if you still have performance problems, they probably aren't related to Tapestry.

Clustering versus Sticky Sessions

For web applications, **clustering** is a load-balancing technique in which multiple application servers are set up to behave as one big server. Generally this requires replicating HttpSession data across the servers, to ensure that a user's web interactions will continue without interruption regardless of which server handles the next request. Session replication achieves very high reliability, but it incurs an extra performance cost (due to the serializing and deserializing of session data and the extra network traffic required).

In contrast, **Sticky Sessions** (also called *session persistence* or *sticky persistence*) is a load balancing technique in which each session is assigned to a particular server for the duration of the session. This approach doesn't require copying HTTPSession data between servers, so it's very scalable. But if a server goes down, all of its sessions are lost.

In general, the sticky sessions approach is the way to go when possible (that is, when performance is more important than session survival). It represents a much more efficient use of resources ... you are scaling *out* not *up*, which is always cheaper. It also means that you don't have to be as careful about what goes into the HTTPSession.

For details on setting up clustering and sticky sessions, see the documentation of whatever load balancer you are using.

Clustering

Tapestry is designed to be "a good citizen" of an application server that supports clustering. It is careful about what it writes into the HttpSession. The framework understands that the server that receives a request may not be the same one that rendered the page initially; this knowledge affects many code paths, and it guides the approach Tapestry takes to caching page and component properties.

Your part is to properly manage the objects put into the HttpSession (via @SessionAttribute, @SessionState or @Persist; see Performance and Clustering):

- Don't store anything in the session that you don't have to. Principally this means minimizing the use of @Persist (see Page Activation and Using Select With a List), storing only IDs in the session rather than whole entities.
- Where possible, persist only objects that are immutable (i.e., String, or a primitive or wrapper type).
- Only put *serializable* objects into the session.
- Make use of the @ImmutableSessionPersistedObject annotation and OptimizedSessionPersistedObject interface (both described below).

Again, Tapestry is a good citizen, but from the application server's point of view, it's just another servlet application. The heavy lifting here is application server specific.

Clustering Issues

The Servlet API was designed with the intention that there would be only a modest amount of server-side state, and that the stored values would be individual numbers and strings, and thus, immutable.

However, many web applications do not use the HttpSession this way, instead storing large, mutable objects in the session. This is not a problem for single servers, but in a cluster, anything stored in the session must be serialized to a bytestream and distributed to other servers within the cluster, and restored there.

Most application servers perform that serialization and distribution whenever HttpSession.setAttribute() is called. This creates a data consistency problem for mutable objects, because if you read a mutable session object, change its state, but *don't* invoke setAttribute(), the changes will be isolated to just a single server in the cluster.

Tapestry attempts to solve this: any session-persisted object that is read during a request will be re-stored back into the HttpSession at the end of the request. This ensures that changed internal state of those mutable objects is properly replicated around the cluster.

But while this solution solves the data consistency problem, it does so at the expense of performance, since all of those calls to setAttribute() result in extra session data being replicated needlessly if the internal state of the mutable object hasn't changed.

Tapestry has solutions to this, too:

@ImmutableSessionPersistedObject Annotation

Tapestry knows that Java's String, Number and Boolean classes are immutable. Immutable objects do not require a re-store into the session.

You can mark your own session objects as immutable (and thus not requiring session replication) using the ImmutableSessionPersistedObject annotation.

OptimizedSessionPersistedObject Interface

The OptimizedSessionPersistedObject interface allows an object to control this behavior. An object with this interface can track when its mutable state changes. Typically, you should extend from the BaseOptimizedSessionPersistedObject base class.

SessionPersistedObjectAnalyzer Service

The SessionPersistedObjectAnalyzer service is ultimately responsible for determining whether a session persisted object is dirty or not (dirty meaning in need of a restore into the session). This is an extensible service where new strategies, for new classes, can be introduced.