UnitOfWorkProposal

Problem Statement

April 16 2004, between 1.0-alpha-3 and alpha-4.

Some discussion on the dev mailing list has brought into question some aspects of how HiveMind interacts with threads. This is an attempt to summarize.

HiveMind explicitly addresses the issues of threading and multi-tasking in several ways. Service creation is explicitly threadsafe, through traditional synchronized blocks. In addition, HiveMind has structures in place to keep concurrent processing in separate threads separate. The threaded and pooled service models are used to create singletons that are accessible only to the current thread. Literally, the proxy will use a JDK ThreadLocal to store the service and create (or acquire) a new instance, specific to that thread, as necessary. Such a proxy can be shared between two threads and invoking methods on the proxy will result in two different implementations receiving the method invocations, each restricted to a single thread. Additional infrastructure, ThreadLocalStorage and ThreadEventNotifier, are available (and used internally by HiveMind).

The entire reason to have thread-local singletons (rather than the default, JVM-wide singletons) is so that the thread-local singletons can have internal state specific to a single "client". In this case, a client may represent a single HTTP request, a single web-service request, a single JMS notification, etc. In all these cases, the request finishes, at which point client-specific state must be discarded before another request, for another client, is procssed.

Thus, it is important that `ThreadEventNotifier.fireThreadCleanup()` is invoked consistently.

At issue here is whether this is the correct metaphor.

What's really at stake here is not the thread, or the specific state information in the thread, but a *unit of work*.

In other systems, a Transaction (or Session, or whatever you might call it) will be explicitly created or otherwise obtained, used (by a single thread, typically), and then closed, committed or otherwise finalized (before being discarded).

Solution Overview

I'm not nearly certain we want to pursue this, now or ever. Regardless, here's my vision of it.

Obviously, all naming here is not final — just off the top of my head.

The Registry would no longer have methods for obtaining services. Instead, it would have a createServiceSession(). In fact, Registry may no longer be an accurate name for what this object does; ServiceSessionFactory may be more accurate.

```
public interface ServiceSession
{
   Object getService(String name, Class castType);

   void close();
}
```

The proxies returned by `getService()` would be specific to the ServiceSession.

When the session is closed, services with the threaded or pooled service models receive notifications. Pooled implementations are returned to a global pool.

All proxies for the session are then deactivated (invoking service methods on them will result in ApplicationRuntimeException).

Using this approach, multiple threads can share the session. If this isn't desired, then the threads should create their own independent sessions.

The need for thread-specific data will be eliminated; instead it will be specific to a ServiceSession, which is easier to manage. There are still issues, because a session can be shared between threads. The threaded and pooled service models would be more efficient (currently, each service method requires an access to a ThreadLocal).

Discussion

As stated above, I don't know that this is *necessary* or *desirable*. There may be a performance impact; more object creation of at least the proxies.

I have some nagging problems as well, with respect to DependencyInjection. Since each proxy belongs to a ServiceSession, that mucks up the works with collaborating services. For example, inside session #1, we access service A which is injected with service B. We get a proxy to service A, and A's implementation gets a proxy to service B. What happens when we close our ServiceSession? The proxies for A and B are disabled. What do we do with A's implementation? It's no longer useable, since it can't invoke any methods on B (the proxy to B has been disabled).

My first thought was that we could have a special, global ServiceSession that keeps long-lived versions of A's implementation. But what if service B has a threaded or pooled service model?

So I'm beginning to think that to re-architected around sessions, we have to ditch service models ... and I think the service models are useful, and a key distinguishing feature of HiveMind. Perhaps I'll come up with a way to reconcile the two ... but I suspect any solution will be a house of cards.