Page Life Cycle

This is an advanced topic. Most users won't ever need to know anything about the page life cycle.

In Tapestry, you are free to develop your presentation objects, page and components classes, as ordinary objects, complete with instance variables and so forth.

This is somewhat revolutionary in terms of web development in Java. By comparison, using traditional servlets, or Struts, your presentation objects (Servlets, or Struts Actions, or the equivalent in other frameworks) are *stateless singletons*. That is, a *single* instance is created, and all incoming requests are threaded through that single instance. Because multiple requests are handled by many different threads, this means that the singleton's instance variables are useless ... any value written into an instance variable would immediately be overwritten by a different thread. Thus, it is necessary to use the Servlet API's HttpServletRequest object to store per-request data, and the HttpSession object to store data between requests.

Tapestry takes a very different approach.

In Tapestry, each page is a singleton, but with a *per thread* map of field names & values that Tapestry invisibly manages for you.

With this approach, all the difficult, ugly issues related to multi-threading go by the wayside. Instead, familiar, simple coding practices (using ordinary methods and fields) can be used.

Tapestry 5.0 and 5.1 used page pooling, rather than a singleton page with a per-thread map, to achieve the same effect.

Related Articles

- Page Life Cycle
- Component Rendering
- Component Events
- Page Navigation
- Request Processing
- Component Events FAQ

The page life cycle is quite simple:

1. When first needed, a page is loaded. Loading a page involves instantiating the components of the page and connecting them together.
2. Once a page is loaded, it is *attached* to the current request. Remember that there will be many threads, each handling its own request to the same page.
3. At the end of a request, after a response has been sent to the client, the page is *detached* from the request. This is a chance to perform any cleanup needed for the page.

Page Life Cycle Methods

There are rare occasions where it is useful for a component to perform some operations, usually some kind of initialization or caching, based on the life cycle of the page.

As with component rendering, you have the ability to make your components "aware" of these events by telling Tapestry what methods to invoke for each.

Page life cycle methods should take no parameters and return void.

You have the choice of attaching an annotation to a method, or simply using the method naming conventions:

Annotation	Method Name	When Called
@PageLoaded	pageLoaded()	After the page is fully loaded
@PageAttached	pageAttached()	After the page is attached to the request.
@PageReset	pageReset()	After the page is *activated*, except when requesting the same page
@PageDetached	pageDetached()	AFter the page is detached from the request.

The @PageReset life cycle (only for Tapestry 5.2 and later) is invoked on a page render request when the page is linked to from some *other* page of the application (but *not* on a link to the same page), or upon a reload of the page in the browser. This is to allow the page to reset its state, if any, when a user returns to the page from some other part of the application.

Comparison to JavaServer Pages

JSPs also act as singletons. However, the individual JSP tags are pooled.

This is one of the areas where Tapestry can significantly outperform JSPs. Much of the code inside a compiled JSP class concerns getting tags from a tag pool, configuring the properties of the tag instance, using the tag instance, then cleaning up the tag instance and putting it back in the pool.

The operations Tapestry does once per request are instead executed dozens or potentially hundreds of times (depending the complexity of the page, and if any nested loops occur).

Pooling JSP tags is simply the wrong granularity.

Tapestry can also take advantage of its more coarse grained caching to optimize how data moves, via parameters, between components. This means that Tapestry pages will actually speed up after they render the first time.

Page Pool Configuration

This section is related to versions of Tapestry prior to 5.2. Modern Tapestry uses an alternate approach that allows a single page instance to be shared across many request processing threads.

In Tapestry 5.0 and 5.1, a page pool is used to store page instances. The pool is "keyed" on the name of the page (such as "start") and the *locale* for the page (such as "en" or "fr").

Within each key, Tapestry tracks the number of page instances that have been created, as well as the number that are in use (currently attached to a request).

When a page is first accessed in a request, it is taken from the pool. Tapestry has some configuration values that control the details of how and when page instances are created.

- If a free page instance is available, the page is marked in use and attached to the request.
- If there are fewer page instances than the *soft limit*, then a new page instance is simply created and attached to the request.
- If the soft limit has been reached, Tapestry will wait for a short period of time for a page instance to become available before creating a new page instance.
- If the hard limit has been reached, Tapestry will throw an exception rather than create a new page instance.
- Otherwise, Tapestry will create a new page instance.
  Thus a busy application will initially create pages up-to the soft limit (which defaults to five page instances). If the application continues to be pounded with requests, it will slow its request processing, using the soft wait time in an attempt to reuse an existing page instance.

A truly busy application will continue to create new page instances as needed until the hard limit is reached.

Remember that all these configuration values are per key: the combination of page name and locale. Thus even with a hard limit of 20, you may eventually find that Tapestry has created 20 start page instances for locale "en" *and* 20 start page instances for locale "fr" (if your application is configured to support both English and French). Likewise, you may have 20 instances for the start page, and 20 instances for the newaccount page.

Tapestry periodically checks its cache for page instances that have not been used recently (within a configurable window). Unused page instances are release to the garbage collector.

The end result is that you have quite a degree of tuning control over the process. If memory is a limitation and throughput can be sacrificed, try lowering the soft and hard limit and increasing the soft wait.

If performance is absolute and you have lots of memory, then increase the soft and hard limit and reduce the soft wait. This encourages Tapestry to create more page instances and not wait as long to re-use existing instances.

Localization   User Guide   Request Processing