Tapestry Inversion of Control FAQ

Tapestry Inversion of Control Container

Main article: Tapestry IoC

Contents

Why do I need to define an interface for my services? Why can't I just use the class itself?

First of all: you can do exactly this, but you lose some of the functionality that Tapestry's IoC container provides.

The reason for the split is so that Tapestry can provide functionality for your service around the core service implementation. It does this by creating *proxies*: Java classes that implement the service interface. The methods of the proxy will ultimately invoke the methods of your service implementation.

Related Articles

- Tapestry IoC Overview
- IOC
- Tapestry Inversion of Control FAQ
- IoC cookbook

One of the primary purposes for proxies is to encapsulate the service's life cycle: most services are singletons that are created *just in time*. Just in time means only as soon as you invoke a method. What's going on is that the life cycle proxy (the object that gets injected into pages, components or other service implementations) checks on each method invocation to see if the actual service exists yet. If not, it instantiates and configures it (using proper locking to ensure thread safety), then delegates the method invocation to the service.

If you bind a service class (not a service interface and class), then the service is fully instantiated the first time it is injected, rather than at that first method invocation. Further, you can't use decorations or method advice on such a service.

The final reason for the service interface / implementation split is to nudge you towards always coding to an interface, which has manifest benefits for code structure, robustness, and testability.

My service starts a thread; how do I know when the application is shutting down, to stop that thread?

This same concern applies to any long-lived resource (a thread, a database connection, a JMS queue connection) that a service may hold onto. Your code needs to know when the application has been undeployed and shutdown. This is actually quite easy, by adding some post-injection logic to your implementation class.

MyServiceImpl.java

```
public class MyServiceImpl implements MyService
{
  private boolean shuttingDown;

  private final Thread workerThread;

  public MyServiceImpl()
  {
    workerThread = new Thread(. . .);
  }

  . . .

  @PostInjection
  public void startupService(RegistryShutdownHub shutdownHub)
  {
    shutdownHub.addRegistryShutdownListener(new Runnable()
    {
      public void run()
      {
        shuttingDown = true;

        workerThread.interrupt();
      }
    });
  }
}
```

After Tapestry invokes the constructor of the service implementation, and after it performs any field injections, it invokes post injection methods. The methods must be public and return void. Parameters to a post injection method represent further injections ... in the above example, the RegistryShutdownHub is injected into the PostInjection method, since it is only used inside that one method.

It is **not** recommended that MyServiceImpl take RegistryShutdownHub as a constructor parameter and register itself as a listener inside the constructor. Doing so is an example of unsafe publishing, an unlikely but potential thread safety issue.

This same technique will work for any kind of resource that must be cleaned up or destroyed when the registry shuts down.

Be careful not to invoke methods on any service proxy objects as they will also be shutting down with the Registry. A RegistryShutdownListener should not be reliant on anything outside of itself.

How do I make my service startup with the rest of the application, rather than lazily?

Tapestry services are designed to be *lazy*; they are only fully realized when needed: when the first method on the service interface is invoked.

Sometimes a service does extra work that is desirable at application startup: examples may be registering message handlers with a JMS implementation, or setting up indexing. Since the service's constructor (or @PostInjection methods) are not invoked until the service is realized.

The solution is the @EagerLoad annotation; service implementation classes marked with this annotation are loaded when the Registry is first startup, rather than lazily.