Sakai's Extension of the Spring Framework to Implement our Component Management Features

Glenn R. Golden, Sakai Architect
January 30, 2005, Version 1.1

Sakai provides a component management system – a way to define APIs and build components that implement the APIs; to select at runtime the set of components that will be available to meet each API; and to discover or inject this selection into any client of the API dynamically without prior client knowledge of the selection.

Multiple Webapp Cooperation

Sakai's component management is unique in that it allows components to be loaded in separate webapps within the servlet container. The combined set of components from all webapps work together to provide a single integrated component system. Components from one webapp may be needed for injection into components from another webapp. This gives Sakai two distinct advantages:

1) Sakai component packaging and selection is done at the webapp level; picking the set of components to implement needed APIs becomes a selection of webapps that provide different components. To change a particular component, the Sakai integrator simply removes one webapp and replaces it with another.

2) Sakai components in different webapps are loaded each in their own class loader, the one created for the webapp. This isolates each component from other components with regard to their configurations and third party .jar dependencies. We do not need to try to reconcile different components that need different versions of same third party package; we simply place each component in a separate webapp, and load into each webapp the proper set and versions of dependent jars.

Spring Framework

The Spring Framework (http://www.springframework.org) has within its many features a rich support for component management in its "bean factories". Sakai currently uses these features as the base of our implementation of our ComponentManager API.

Spring has many more features that are not used in Sakai. Spring is not part of the public Sakai APIs; its use is isolated to our implementation of component management.

Spring's bean factories do not directly support cooperation among all webapps that is a feature and requirement of Sakai. For this, we have extended Spring. Our extension to Spring is a single java class; we do not need to make any modifications to Spring itself.

Sakai Component Manager Components

Sakai's component manager is made up of 4 main classes and the ComponentManager API and static cover. These are:

1) ComponentsServlet – a servlet used to initialize a webapp's component support.

2) SpringComponentManager – an implementation of the Sakai ComponentManager API implemented using our bean factory extended from Spring.

3) SakaiBeanFactory – our extension to Spring's bean factory that implements our cooperating webapp behavior.

4) ComponentMap – to provide the available beans in a Map form by interface name, used to make our beans more available (this is used to make bean injection available in our JavaServer Faces integration).

Sakai's Bean Factory – Spring Extension

Sakai's extension starts at the level of the Spring BeanFactory. Sakai's SakaiBeanFactory extends one of Spring's workhorse bean factories, the DefaultListableBeanFactory. Sakai creates a global, shared SakaiBeanFactory instance, and creates another SakaiBeanFactory instance in each Sakai webapp. The webapp factories are hooked up as children to the global instance.

So far, this is pretty standard Spring – we end up with a 2 level hierarchy of bean factories. In unmodified Spring, when a component is being created and its dependencies are being injected, the beans are looked for in the set of beans and bean definitions in the local bean factory, and if they are not found there, they are looked for in the parent bean factory. There is no provision for combining the sibling bean factories, the parent's other children, which for Sakai are those factories in the other webapps.

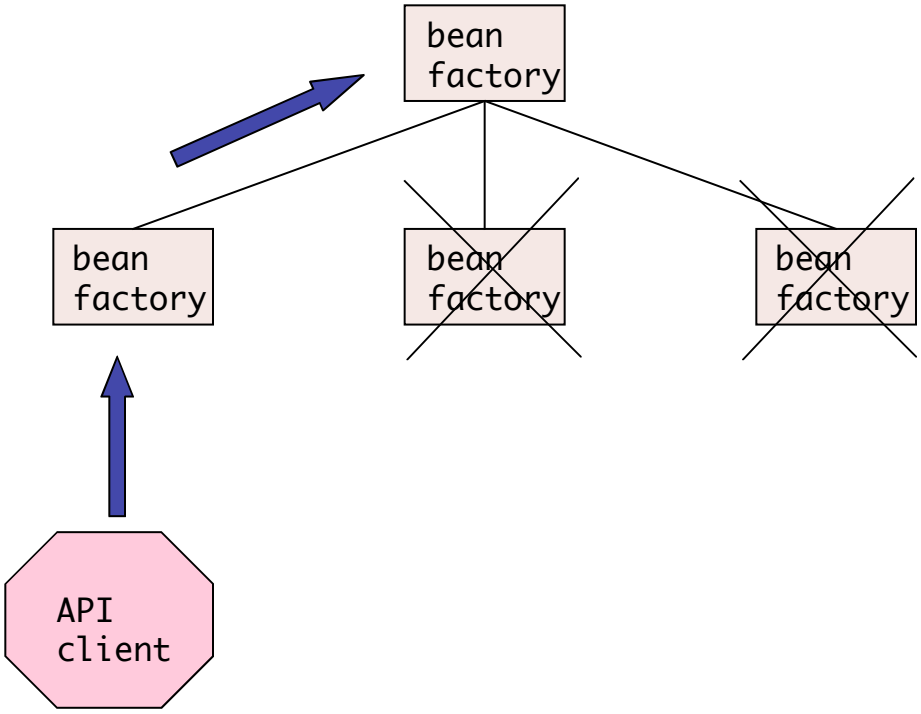

Figure 1: standard Spring "up looking" bean hierarchy

Sakai's bean factory extension solves this problem by using an "up/down" approach. The normal parent / child relationship is disabled, disabling the normal Spring up-looking approach. Instead, each request for a bean is promoted all the way *up* to the parent, and once there, it is applied *down* to each child, in turn, until it is satisfied.

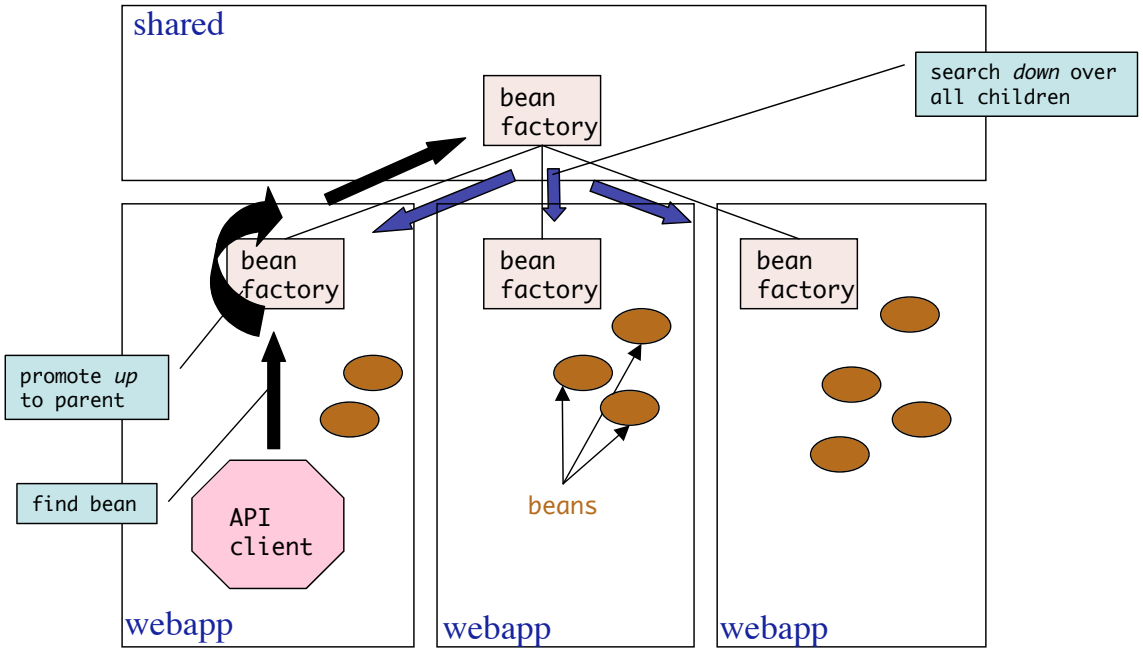

Figure 2: Sakai enhanced "up / down" looking bean hierarchy

For example, if a bean in webapp A needs a bean that happens to be provided by webapp B, the bean search starts out in A's bean factory, is promoted up to the global bean factory, then is sent down to all the webapp bean factories. Once it is sent to B's factory, the bean definition is found and that bean factory creates the bean. Because these bean factories have been created in each webapp, they use the webapp's class loader to locate the bean and its dependencies. But once he bean is created, it can be used safely from any other webapp.

It is possible that, in the above example, bean B will need further beans to be injected, so the process repeats. It's possible that B's dependency will be satisfied by webapp A, or C, or even locally in B.

Once a bean is created in any of the webapp bean factories, it is available for discovery or injection for any other webapp. The bean will use the class loader from the time it was created to load in dependent classes, so it will look only locally (or in the shared area) for these classes, not in other webapps. In this way conflicting version dependencies are avoided.

Clients of these API component beans of course know nothing about the specific class they are using. That's the basis of API programming. They only know the API. In order for the API to be truly shared among code in different webapps, it must live in the shared area. This is not usually a problem, since the API does not have third party dependencies like the implementation components do.

The SakaiBeanFactory achieves this up/down approach in two ways:

1) It breaks the normal parent / child bookkeeping, to keep Spring from doing the normal thing. It replaces this with its own parent / child bookkeeping.

2) In overrides many BeanFactory methods with code that sends up, if there is an up, then iterates over the children, sending down.

Here's an example:

```
public String[] getBeanDefinitionNames()
{
	// move this call to the root
	if (m_parent != null)
	{
		return ((DefaultListableBeanFactory) m_parent).getBeanDefinitionNames();
	}
	else
	{
		return getBeanDefinitionNamesDown(true);
	}
}

protected String[] getBeanDefinitionNamesDown(boolean children)
{
```

```
        List rv = new Vector();
        String[] names = super.getBeanDefinitionNames();
        rv.addAll(Arrays.asList(names));

        if (children)
        {
               for (Iterator iChildren = m_children.iterator(); iChildren.hasNext();)
               {
                      SakaiBeanFactory child = (SakaiBeanFactory) iChildren.next();
                      names = child.getBeanDefinitionNamesDown(children);
                      rv.addAll(Arrays.asList(names));
               }
        }
        return (String[]) rv.toArray(new String[rv.size()]);
}
```

When getLocalBeanDefinitionNames() is called in the normal course of Spring's bean work, this method is overridden. If there is a parent, this method on the parent is called. When the global bean factory (the parent of the webapp's bean factory) gets this call, it calls the new method getBeanDefinitionDown().

The new method checks locally for the answer (the super call, directly invoking the DefaultListableBeanFactory method), and then augments the result with the result from each child's getBeanDefinitionDown() method call. In this way each of the webapps is polled for the results.

Other methods that are searching (rather than listing) will poll the children webapp bean factories until the answer is found, rather than calling them all.

Webapp Load Order and Reloading

One consequence of having our components distributed over the webapps is that we must assure that all the webapps have been loaded, and all the child bean factories created, initialized with their definitions, and linked to the global parent, before we start constructing beans. This means:

1) We must not in any webapp initialization code invoke any service components or attempt discovery, or

2) We must wait till we get a signal that all the webapps are loaded before we start using the component system.

Sakai's ComponentManager API is used to discover components and otherwise interact with the component management system. It has a method, isReady(), that may be polled to wait for the condition that all the webapps have been loaded and it is safe to start using the component system.

How the component manager knows that things are all loaded is another issue, and we have a weak solution for it now, that depends on the load order of webapps. Tomcat seems to usually load the webapps in alphabetical order ("seems", "usually" – I warned

you this is weak). We have designed a special webapp called "z-last" to be last to load (we hope). When it loads its ComponentsServlet to create a local ComponentManager with its local BeanFactory, it sets the "set-ready" parameter. The ComponentsServlet then tells the component manager hierarchy to act as if all is ready.

The "z-last" webapp's ComponentsServlet is also configured with the "preload" parameter. This signals the ComponentsServlet to load all the registered components in all the webapps that are registered with valid shared API interface names. That starts up all the services at server startup, rather than waiting for them to get their first invocations, so that:

1) The system is ready for business at startup, rather than waking up slowly as the various first invocations occur.

2) Components that run background threads and may have no other calls from the outside get to startup with server startup.

The startup only of beans named with shared available interfaces allows us to define very local beans, and beans that are not named by their interfaces, such as are used by Spring-Hibernate. These are not pre-loaded.

Once the webapps are all loaded, we cannot dynamically stop and replace them without a total restart of our servlet container. This would require hot swapping of components; something we might in the future support.

Load Order Trouble

We have seen cases where Tomcat does not load z-last last, and that just breaks our system. These seem to be platform related. We will address this in Sakai 2, with a possible back port to Sakai 1.5.