Sakai's Component Manager API, Component Packaging, and the Underlying Spring Implementation

March 23, 2005

Please direct questions or comments about this document to:
Glenn R. Golden, ggolden@umich.edu

Component Management

One of Sakai's most important framework features is the component management system. This is 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.

Client software of the API will usually use declarative injection to make available the component that satisfies the API at runtime. Where injection is not possible, the client will use discovery to find a component, by calling on Sakai's ComponentManager API directly.

Components are usually made available to the component management system by declaration. A bit of special declaration text, in an XML dialect, associates the class of the component with the class of the API it satisfies, and also declares the components dependencies on other APIs (for injection) and other configuration parameters.

In rare cases, a component instance already created can be registered without being declared, by calling on Sakai's ComponentManager API.

Component Packaging

Sakai's component management is unique in that it allows components to be loaded in separate independent packages. The combined set of components from all components packages work together to provide a single integrated component system. Components from one package may be needed to satisfy API dependencies in components from other packages. This gives Sakai two distinct advantages:

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

2) Sakai components in different packages are loaded each in their own class loader, created just for the package. This isolates each components package from other packages 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 package, and load into each package the proper set and versions of dependent jars.

The components package system is much like the Servlet webapp system in its modularity and isolation.

In addition to the components packages, Sakai supports components hosted in the shared class area, and also components hosted in particular webapps.

Spring Framework

Sakai's component management system is built over a network of Spring Framework ApplicationContext (AC) objects. These act as a network of "bean factories" managing our component instances. Spring's bean definition language is used to declare each Sakai component's availability and component injection and configuration needs.

Sakai uses Spring in a certain way to meet our component packaging needs, but does so without the need to make any changes to or significant extensions to the Spring software. Sakai's use of Spring is done in a "Spring friendly" way, so that developers familiar with Spring can leverage that experience, and developers who want to can take advantage of other Spring features within Sakai webapps.

Writing Portable Code

Sakai's use of spring makes it possible for an application developer to directly use the Spring APIs to discover components and do other things in standard Spring ways. This code will interoperate with the rest of Sakai; components discovered can be hosted in components packages, or locally in the webapp. Applications can also register non-component beans in the local Spring AC for other needs (such as for Spring-Hibernate).

But for the purpose of component management, when there is need for discovery of a component, we encourage developers to use the Sakai ComponentManager API rather than the Spring APIs directly. This will result in more portable code. We encourage the minimization of any dependency on Spring in the Sakai framework and applications. If in the future Sakai is changed to use another component management support layer instead of Spring, code written to the Sakai APIs will not have to change, while the code written directly to Spring will be broken.

ComponentManager Cover

The ComponentManager API, like many other Sakai APIs, includes a "cover" for the API. A cover is a static class that has methods with the same signatures as the API, but

the methods are static so are directly callable. The methods called in the cover use discovery to find the component for the covered API and pass on the calls.

A cover for the ComponentManager is a different, since it cannot call on the ComponentManager API's cover to discovery its component. Instead, it has the responsibility for creating and holding the singleton instance of the ComponentManager implementation. This is the only cover class in Sakai that actually knows the class of the implementation:

```
private static
org.sakaiproject.service.framework.component.ComponentManager
m_componentManager = new SpringCompMgr(null);
```

This cover is part of the API, so it is in `/shared/lib/` with the API files. This makes the component manager singleton shared among all webapps and components packages.

Sakai's Spring Implementation

Sakai implements the ComponentManager API as façade over the component management capabilities of the Spring. We create a 2-tiered network of Spring ApplicationContext objects, and use these:

a) to populate, inject and configure the set of available components, and

b) to satisfy the ComponentManager API.

The top tier of the network is a single AC shared by all webapps and components packages. This AC is used to hold the implementation components of all the shared Sakai APIs. These shared components are those hosted and registered by the many components packages (and perhaps webapps) that make up a Sakai system.

The second tier of the network is a set of ACs, one in each Sakai webapp, each using the shared AC as a parent. These local ACs are used by Spring aware applications to access the Spring component management system (and other Spring features). Any request that cannot be satisfied by the local AC will automatically be sent up to the shared AC parent by Spring. Beans that are not components, or are components for local APIs, can be registered in the local AC. Each local AC, in each webapp, is isolated from the others, so there's no worry of name space collision between webapps.

The Sakai ContextLoaderListener is used in each webapp to created the local AC, link it to the shared parent AC, and populated them from application context (bean) definition files. This is used instead of the Spring ContextLoaderListener.

Sakai's ContextLoaderListener

The Servlet API (2.4) defines a way to trigger code to run whenever a webapp (context) is created, the context listener. Sakai provides a listener to use in Sakai application webapps.

This listener extends the Spring ContextLoaderListener. The basic function of the listener invokes the standard Spring context loader (the same code that the Spring context listener uses) to load the local AC's definition file. This file defaults to `applicationContext.xml` in the webapp's `WEB-INF` folder, but can be set to read one or more files by configuring the servlet context.

After the local AC is defined, the extra features of the Sakai listener are invoked. The shared AC is located (using the Sakai ComponentManager API), and created if needed. The local AC sets the shared AC as its parent.

Next, the Spring listener looks for another set of component definition files, which list the beans to be loaded from the webapp context into the shared AC. This is for the components that the webapp hosts that satisfy shared Sakai APIs for use by the entire Sakai system. If this is used, the shared APIs must be put into a jar and installed into `/shared/lib/`. There are some disadvantages of using this method that are covered later.

ComponentManager Initialization

On the first access to the Sakai ComponentManager (through the cover), the implementation singleton is created.

This class creates the shared AC, and locates a special component definition file to load up Sakai framework components and any other components that are in shared and not hosted by a components package or webapp. This file is located on the class path at `org.sakaiproject.component.shared_components.xml`. This can be a file in the `/shared/classes/` hierarchy, or be in a jar file. This file distributed with Sakai lists the standard framework components to pre-load from shared

Our ComponentManager implementation then finds the ComponentsLoader, and uses it to load up all the available components packages.

After shared components and components packages are loaded, it instantiates all the registered singletons, and the component manager is "ready" to serve requests.

Listener Configuration

The Sakai ContextLoaderListener can be configured to name the file or files to load into the local AC using the `contextConfigLocation` context parameter specified in the `web.xml` for the webapp. The default is to load a single file called `/WEB-INF/applicationContext.xml`. This can be overridden by setting a list of file names with space, comma, tab or new line separation in the context parameter like this:

```
    <!-- list of components to load locally -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/local.xml</param-value>
```

```
    </context-param>
```

Sakai's listener can also be configured to load one or more files into the shared AC using the `contextSharedLocation` parameter like this:

```
    <!-- list of components to load into the shared components -->
    <context-param>
        <param-name>contextSharedLocation</param-name>
        <param-value>/WEB-INF/shared.xml</param-value>
    </context-param>
```

If this is missing nothing will be loaded from this webapp into the shared context.

The listener is declared in the webapp in the `web.xml` like this:

```
<listener>
  <listener-class>
    org.sakaiproject.util.ContextLoaderListener
  </listener-class>
</listener>
```

Locate this after the servlets and servlet mappings, and before the welcome file list.

Packaging Component Packaging Options

There are three ways to make a component available to Sakai.

1) Host the component in a webapp.

2) Put the component into `/shared/lib/` and the component manager's shared components declaration file.

3) Host the component in a Sakai components package.

There are problems with the first two approaches. We recommend that shared components be hosted in components packages.

When considering the various component hosting methods, keep in mind how Spring loads component definitions and instantiates the beans. Spring uses the thread's context class loader at the time that beans are registered to find the bean's implementation class. The class loader that located the class at registration time is used later to instantiate the bean, without regard to the thread context class loader or any other class loader at bean instantiation time.

Webapp Component Hosting

Components that are used locally within a Sakai application can simply be packaged in the application's webapp. The API interfaces and implementation classes can be

included with the webapp, and the components declared to be loaded into the webapp's local AC.

Components that are shared by many applications could be hosted in a webapp. The API interfaces have to be placed into jar files and put into `/shared/lib/`. The components would be declared in the webapp to be loaded into the shared AC. These components would be loaded in the webapp's class loader, so their configuration and third party dependencies are isolated from all other webapps and components packages.

There are problems with this approach. Webapps are loaded in a non-specified order, and there is no standard way to know when the webapps are all loaded. This means that components hosted in a webapp will not be pre-instantiated at server startup; they will be created the first time they are referenced. This also means that these component definitions are not going to be available at server startup; we know they are available only after the server has started. No startup process can rely on them. This means that they are not available for component injection, and not available for Servlet init() calls.

Because of these restrictions at server startup, we do not recommend that shared components be hosted in webapps.

Component Living in Shared

The Sakai ComponentManager implementation's shared Spring AC is configured with a bean definition file located on the `/shared/` classpath. Component implementation classes and their dependencies can be placed into `/shared/classes/` or `/shared/lib/`, and added to this file. These will be pre-loaded when the component manager starts up, pre-instantiated at server startup, and available to any component injection or webapp init() at server startup.

Certain standard Sakai framework and common components may be distributed this way.

Shared-hosted components do not have the startup problems that webapp-hosted components do. But they have a major problem of their own. Components that are hosted in shared need to have their dependencies and their implementation classes in shared, so they compete with each other and all the webapps for jar versions and configurations. There is no isolation that we have in webapp hosting or components package hosting.

Also, the `/shared/` space is shared among all applications, so adding components here "pollutes" this shared space. The components and their dependencies are not really shared – they are commonly used across the Sakai system, but only the shared API interfaces need to be visible in `/shared/`.

Components hosted in shared are not so easy to pick and choose, to install and replace, because they are mixed in with all the other files in `/shared/`, and share a single bean definition file.

Because of this lack of isolation and packaging ease, we do not recommend hosting components in the /shared/ area.

Components Packages

Sakai's components packages offers a system that has the isolation and packaging of webapps, but none of the server startup problems.

A components package holds the classes and jars of one or more components, all the dependent classes and jars, and the bean definition file that declares, configures and injects these components.

Each components package is given a webapp-like class loader to isolate the classes from other components packages and webapps. Components packages can see the /shared/ classes and above in addition to their own classes and jars.

Components packages are built as war files that have a `components.xml` file instead of a `web.xml` file. Components packages are expanded into a special `/components/` directory instead of `/webapps/`. And of course they don't take http requests as webapps do, they just host components that respond to API calls.

The source code project for a components package looks much like that for a webapp, but instead of a `web.xml` in the `src/webapp/WEB-INF` folder, there's a `components.xml` file. The maven `project.xml` declares this as building a "components" rather than a "war":

```
<project>
      <pomVersion>3</pomVersion>
      <name>Sakai Component Manager Test Components 1</name>
      <groupId>sakaiproject</groupId>
      <id>sakai2-comp-test-components1</id>
      <currentVersion>sakai.2.0.0</currentVersion>

      <properties>
            <!-- deploy as a components -->
            <deploy.type>components</deploy.type>
      </properties>

      <dependencies>
      …
```

The sakai-maven-plugin knows to use the war building code to build a components package, and knows how to deploy these to the `/components/` folder. These are expanded on installation.

To select a set of components to include in a Sakai, simply include the components package project in a Sakai source module, and build. To replace a component, remove one package project, put in the desired package project, and build.

Although components packages are ideal for implementations of the Sakai framework and common shared APIs, they may also be used by a Sakai application for their own service APIs. To do this, the application module would have at least 3 projects:

1) the application tool / user interface code, building a webapp

2) the application API interfaces, building a `/shared/` jar

3) the application components, building a components package

This helps with separation of concerns in the application, and also makes the APIs available to other applications, which is sometimes useful.

The `/components/` root location will be able to be configured – for now it's assumed to be a peer folder to Tomcat's `/webapps/`folder.

Components packages are the preferred way to host components in Sakai.

Components Package Class Loader

A class loader is created for each components package. The parent of this class loader is the shared class loader, the same parent used for the webapp class loaders. The class loader can see the `/WEB-INF/classes/` classes, and all the jar files in `/WEB-INF/lib/` in the components package.

The class loader is much like the webapp class loader, with one difference. The components package uses a standard URLClassLoader, and like other standard class loaders, the loader will **first** ask the parent class loader for the class, and only look at its target locations if the class is not defined in a parent.

Webapp class loaders violate this standard behavior, at the request of the Servlet spec, by looking locally first, for most classes. This assures that a webapp will use the classes and jars it has even if there is another version of the same class or jar in `/shared/` or `/common/`.

Sakai's components package follows the java standard, not the "look here first" webapp way. If there are classes or jars in shared or common or the system class loader, they will be used instead of locally packaged jars. We can change this to follow the webapp way as an option in the future if desired.

Shutdown

When a webapp is stopped, usually at server shutdown, the context listener will close the local AC. This will destroy any singleton component that the AC is holding.

This works well for the local ACs, but leaves the shared AC running. If we leave the shared AC running when the server shuts down, none of the shared components will have a chance to stop. If they are running a background thread, the java process may continue to run after the container stops. If they have write-cached information to store at shutdown, this may not occur.

The Sakai listener adds a feature to the context destruction process. Our ComponentManager implementation keeps track of how many child ACs are pointing at the shared AC. The listener reports when a new child AC is created and when one is destroyed.

When the component manager detects that the last child AC is removed, it will call its close method. This closes the shared AC, and destroys the singleton components living there.

If the component management system is used with no webapps, the application must call the ComponentManager.close() method when it wants to shut down the shared components.

Tests

The Sakai Test module has a number of projects designed to exercise and demonstrate our component management system.

The projects that make up this test are:

- comp-test-service : a set of shared service APIs for the test
- comp-test-app1 : one webapp building project
- comp-test-app2 : a second webapp building project
- comp-test-components1 : one components package building project
- comp-test-components2 : a second components package building project
- util-test : a jar building project with a utility class to play with

The service APIs are in the `org.sakaiproject.service.test.components` package, and are `Service1`, `Service2`, etc. These are uninteresting, defining a single method (String method()). This project builds a jar to go into shared, so that we can treat these APIs as shared APIs that can be used in webapps and component packages but hosted (i.e. the component for the API) in any of our hosting locations.

App1 and App2 provide a test Servlet and a local component, and make use of some shared components, in various ways to show that this is all working. There are also components named the same as the actual components in these webapps that should not be invoked unless our class loading is messed up.

Components1 and components2 provide two sets of shared components, packaged into components packages.

App2 hosts a shared component for Service6.

Test 1 – Components1 defines a component for Service1 called Component1, and registers this in the shared AC. App1 defines a simple component-using Servlet, CompTest1Tool, which discovers the Service1 component and calls it in two ways:

- Test 1a – use the Sakai component manager to discover a locally hosted shared component.

- Test 1b – use Spring to discover a locally hosted shared component.

App2's tool code is much like App1's tool code.

We also want to be watching that the Component1 is the same when called each time, that it is truly a singleton. The Component1 has a counter incremented on each call to check this.

Test 2 – Each App1 and App2 define a local implementation of Service2, and register it as a local bean. Service2 need not really be a shared API for this test, but it is. The App tool code uses Spring to discover this bean – they cannot use the Sakai ComponentManager since it is not a shared component. We expect that each App1 and App2 get their own Component2, and they do not get mixed up.

Test 3 – We define some dependencies for the Component3 and Component4 (implementing Service3 and Service4) to test that we can have cross components package hosted dependencies, and that the order of the components loading does not matter.

Components1 hosts Component3, which has a dependency on Service5, with its component hosted in Components2. Components2 hosts Component4, which has a dependency on Service1, which as we already noted has the official implementation hosted in Components1. You can see how these are declared in the `components.xml` files – the declaration and use of the components have no knowledge where these are implemented or by which component.

No matter which components package loads first, one will be registering a dependency on a service that is not yet known to the component management hierarchy. But by the time we get our request in, all components packages are loaded, so everything works out.

Test 4 – App2 hosts Componet6 for Service6 as a shared component. This is discovered using the ComponentManager in both App1 and App2. They should both find the same component. This works because Service6 is not discovered at Servlet init() time, nor is it used as a bean dependency.

There are probably some other cases to test, like circular dependencies (i.e. Component7 needs Service8, and Component8 needs Service7, and they are in different webaps), but that's an exercise left for another day.

We should also see what happens with third party jars loaded into each webapp, to make sure that whenever and however the component get instantiated, it will load up jars from its webapp's class loader, not from any other.