Sakai Kernel Bundle
Service Manual – Components

Date: November 9, 2005
Version: 2

Table of Contents

1 Overview

This document describes the Sakai Component Manager and related functions. Requirements for this service are defined in *Sakai Kernel Bundle Requirements* [1]. This document is based, in part, on an earlier description of components by Glenn Golden [3].

Web applications managed by a container like Tomcat are intended to run in isolation; that is, they do not share code or resources. There are a lot of reasons for this, security among them. Sakai, however, has a need to share both resources and code between web applications (Sakai tools).

Sakai refers to any collection of resources and/or code intended to be shared between Sakai tools as *components*.

The Sakai Architecture defines a layered, service-based stack of capabilities. Each service is defined by an application programming interface (API). All access to the capabilities provided by a service should be done through its API. This is sometimes referred to as "coding to the interface" and allows the implementation of a service to be separated from its definition and access points.

Shared Sakai service APIs are deployed to the shared/lib directory of Tomcat as JARs. Since this directory is automatically in the class path of all Tomcat web applications, interfaces are readily accessible to them. Implementations of these interfaces must be loaded at startup time. The Sakai Component Manager (and associated listeners and loaders) are responsible for loading service implementations and making them accessible to web applications that desire to use them.

1.1 Component Definition

A *Component* is a collection of software and/or other resources intended to be accessed by using the component manager and possibly shared across Sakai applications. A component is identified by the full name of the API that it implements.

1.2 Component Access

Access to a Sakai service is accomplished using one of three mechanisms:

- Covers
- Component Manager Access
- Injection

Service covers provide access to a service by defining a static version of the service manager that is globally visible. In general, this static cover includes a getInstance() method that returns the currently defined implementation of the manager. Depending on how it is coded, getInstance() may use the Component Manager Access method described below. The Component Manager cover is an exception to this convention, since it is itself responsible for managing components (this avoids circularity).

Normally, the Component Manager can be used to get an implementation of a Sakai service. An implementation is found by the Component Manager given an identifier string. By convention, this identifier string is a fully qualified name of a service interface. For the SessionManager, the current implementation is bound to the identifier org.sakaiproject.api.kernel.session.SessionManager.

Methods are provided in the Component Manager to get the current implementation of this service by providing the identifier.

The preferred service access mechanism is service injection. An application can define a property in its tool object that is initialized at startup time by the Sakai Component Manager using the Spring Framework. See Service Injection in the Recommended Practices section below.

Bundled Resources

In addition to code, resources can also be bundled and shared between applications. This is purely a convenience mechanism, for example to make it easier to find graphics that may need to be modified for a new skin, and text strings that need to be translated and localized.

2 Application Programming Interface

All of the interfaces, implementations, and objects described here can be found in the *kernel* module in the *component* module-part.

2.1 The ComponentManager API

The Component Manager provides methods that provide access to a singleton object that corresponds to a component (service, resources, etc.).

Get a Component given a class name

Object get(Class iface);	Get the singleton associated with the interface class provided. This singleton is an instance of the object that implements the indicated interface.

Get a Component given an interface name

Object get(String ifaceName);	Get the singleton associated with the interface name provided. This singleton is an instance of the object that implements the indicated interface. An example of an interface name is org.sakaiproject.api.kernel.session.SessionManager

Component Exists Given Class Name

boolean contains(Class iface);	Returns true if the singleton associated with the interface class provided is registered with the Component Manager and exists.

Component Exists Given Interface Name

boolean contains(String ifaceName);	Returns true if the singleton associated with the interface name provided is registered with the Component Manager and exists. An example of an interface name is org.sakaiproject.api.kernel.tool.ToolManager

Get the Set of Registered Components

Set getRegisteredInterfaces();	Returns the set of registered components currently loaded and available as singleton objects. This is a set of interface names. This set can be used to discover the implementation of a known interface.

Load an existing Component given its class

void loadComponent(Class iface, Object component);	In some cases, an instance of a service implementation may be created by methods other than the Component Loader. This method can be used to register this instance as a singleton and associate it with the interface class given.

Load an existing Component given its class name

void loadComponent(String ifaceName, Object component);	In some cases, an instance of a service implementation may be created by methods other than the Component Loader. This method can be used to register this instance as a singleton and associate it with the interface name given.

Close the Component Manager

void close();	This method is called to close the Component Manager. It, in turn, closes the Spring Application Context.

Get Configuration Properties

Properties getConfig();	This method can be used to access many of the configuration properties supported by Sakai. A complete list of configuration properties is included in Appendix 1 below.

2.2 The ComponentsLoader API

The Component Loader defines a method to load all registered components and create singleton objects.

Load Components

void load(ComponentManager mgr, String componentsRoot);	Takes an instance of the component manager and the file system canonical path to the directory where component packages are found. The components root path is saved as a system property using the key `sakai.components.root`: `System.getProperty("sakai.components.root")` System properties do not seem to be fully implemented at this time. The list of system properties is essentially the same as the configuration properties, but all values are null.

3 Default Sakai Implementation

The Sakai Component Manager leverages the Spring Framework Application Context object and the bean definition language. Two levels of context objects are used: a shared context for all of Sakai, and a local context for each web application. Since each local context has a reference to the shared context, shared components can be found.

At startup time, Tomcat processes the web.xml file for each web application (Sakai tool, etc.). If a reference to the SakaiLoaderListener is included in this file, it is invoked by Tomcat, allowing components to be referenced, injected, etc.

In addition, a special component definition file is consulted and all components defined are loaded.

Overall, the process looks like this:

1. Components are defined and registered.
2. APIs are deployed to shared/lib.
3. Implementations are deployed to web applications or to a shared component location.
4. The Component Loader loads all registered components at startup.
5. As tools (web applications) are started, injected services are initialized.
6. Additional runtime access to components is provided by the Component Manager and static covers.

SpringCompMgr is the default implementation of the ComponentManager interface. When it is instantiated, the shared application context is created. As web applications are started up by Tomcat, local application contexts are created and added.

ContextLoader extends org.springframework.web.context.ContextLoader. More information about application context and threading will be provided in [4].

org.sakaiproject.util.ComponentsLoader – this lives in the kernel/component-loader module-part. It isn't clear why this is a utility object and not a component, since it implements the ComponentLoader interface.

4 Utility Objects

4.1 The ContextLoaderListener Object

Sakai's extension to the Spring ContextLoaderListener - use our ContextLoader, and increment / decrement the child count of the ComponentManager on init / destroy.

Create a Context Loader

public createContextLoader()

Initialize root web application Context

public void contextInitialized(ServletContextEvent event)

Close root web application Context

public void contextDestroyed(ServletContextEvent event)

4.2 The ComponentMap Object

ComponentMap exposes the registered components as a map – the component ID is mapped to the component implementation. This utility object is currently only a partial implementation of a Map object.

Contains Componet Map key

public boolean
containsKey(Object arg0)

Get Component Map object

public Object get(Object arg0)

The following methods are **nonfunctional and should not be used**:

public int size()
public boolean isEmpty()
public boolean containsValue(Object arg0)
public Object put(Object arg0, Object arg1)
public Object remove(Object arg0)
public void putAll(Map arg0)
public void clear()
public Set keySet()
public Collection values()
public Set entrySet()

4.3 The PropertyOverrideConfigurer Object

Sakai's extension to the Spring PropertyOverrideConfigurer – allow our dotted bean IDs, use @ as a separator between the bean ID and the property name.

This could be an extension, just defining a new processKey(), but for the *private* members that the extension does not have access to...

Get value

public String getValue(String name)	Access the value of the entry with this name key.

Ignore invalid keys

public void setIgnoreInvalidKeys(boolean ignoreInvalidKeys)	Default is false. If you ignore invalid keys, keys that do not follow the beanName.property format will be logged as warning. This allows you to have arbitrary other [??] keys in a properties file.

Has property overrides for

public boolean hasPropertyOverridesFor(String beanName)	Were there overrides for this bean? Only valid after processing has occurred at least once.

5 Recommended Practices

5.1 Use the Component Manager

Although it is possible to code directly to the Spring Framework, it is recommended that all Sakai applications access components using declared injection. If direct access to components is needed, use the Component Manager via its cover or an injected reference.

5.2 Packaging and Registering Components

Components are implementations of an interface (API). Each API (typically a service) is packaged as JAR and deployed to the shared/lib directory in Tomcat. The following sections show how APIs, components, and resource bundles are setup and deployed.

5.2.1 API Packaging

The Session Manager provides a good example of how an API is defined for general use in Sakai. The session manager module-part can be found in "kernel/session" and has a directory structure that looks like this:

```
session
        src/java/org/sakaiproject/api/kernel/session
                Session.java
                SessionManager.java
                [etc]
        project.xml
```

The critical action for packaging an API happens in the project.xml file, which is used by Maven to build and deploy this code. This example has been edited a bit to show only the essential parts.

```
<project>
        <pomVersion>3</pomVersion>
        <name>Sakai Session API</name>
        <groupId>sakaiproject</groupId>
        <id>sakai-session</id>
        <currentVersion>2.0.0</currentVersion>

        <properties>
                <deploy.type>jar</deploy.type>
                <deploy.target>shared</deploy.target>
        </properties>

        <dependencies>
                <dependency>
                        <groupId>sakaiproject</groupId>
                        <artifactId>sakai-component</artifactId>
```

```
                        <version>${pom.currentVersion}</version>
                </dependency>
        </dependencies>

        <build>
                <sourceDirectory>src/java</sourceDirectory>
        </build>
</project>
```

Let's examine the important aspects of this file via its XML tags.

The <name> tag defines the human readable name of this service.
The <groupId> tag defines what group this package will belong to. In particular, this defines the directory in the local maven repository where it will reside.
The <id> tag is the identifier for this package. This names the directory under the group directory in the maven repository.
The <version> tag is package version. Sakai tries to keep package identifiers in synch with releases. This example is taken from the 2.0.0 release of Sakai, therefore. This version number is very important when creating dependencies as well.

If you concatenate the "id", "-", and "version" together, you get the name of the JAR kept in the maven repository (in the directory defined by id, under groupId). It's good to know this when trying to figure out why you are getting errors about being unable to find an API. This usually comes up as an undefined symbol during the maven build phase.

Next, we define a couple of properties that will be used by maven to determine how to package this code. In this case, the <deploy.type> is set to "jar". Tools are deployed as WARs. The <deploy.target> is set to "shared", which means the "shared/lib" directory of Tomcat. By deploying it to this directory, we ensure that this API will be in the classpath of all web applications in this Tomcat environment.

Project dependencies come next. Here, a dependency on the Component Manager is defined. It is part of the "sakaiproject" group id, and the component id is "sakai-component". The version number is handled by a bit of wizardry in this case by defining above. This is the common practice in most Sakai API package definitions now. However, if you need to reference a non-Sakai JAR, you should enter its version number directly. Thus a dependency on the Apache Commons Loader, would have a group id of "commons-logging", an artififact id of "commons-logging" and a version of "1.0.4".

Finally, maven is directed to look for the java sources of this API in the "src/java" directory under the module-part, which is "kernel/session" in this example.

When maven encounters this module part (and the project.xml file above), it compiles the API code, packages it as a JAR, and deploys it to Tomcat/shared/lib where it can be accessed by all web applications in Sakai.

5.2.2 Component Packaging

The Session Manager in the Sakai kernel also serves as a good example of how an implementation is packaged as a component. The session manager component module-part can be found in "kernel/session-component" and has a directory structure that looks like this:

```
Session-component
        src/java/org/sakaiproject/api/kernel/session
                SessionComponent.java
        project.xml
```

Again the packaging operation for a component happens in the project.xml file, which is used by Maven to build and deploy this code. This example has been edited a bit to show only the essential parts.

```
<project>
        <pomVersion>3</pomVersion>
        <name>Sakai Session API Component</name>
        <groupId>sakaiproject</groupId>
        <id>sakai-session-component</id>
        <currentVersion>2.0.0</currentVersion>

        <properties>
                <deploy.type>jar</deploy.type>
        </properties>

        <dependencies>
                <dependency>
                        <groupId>sakaiproject</groupId>
                        <artifactId>sakai-session</artifactId>

        <version>${pom.currentVersion}</version>
                </dependency>

                Other dependencies include:
                        commons-logging
                        sakai-id
                        sakai-thread_local
                        sakai-util-java
                        concurrent
                        servletapi

                </dependency>
        </dependencies>

        <build>
```

```
                <sourceDirectory>src/java</sourceDirectory>
        </build>
</project>
```

In many ways, this is very similar to the way that API packages are built, with a few exceptions.

The <name> tag defines the human readable name of this service.
The <groupId> tag defines what group this package will belong to.
The <id> tag is the identifier for this package
The <version> tag is package version.

Next we start to see some differences in how packages are deployed. In the properties section of the project.xml file above, this component is to be built as a JAR. This could be built as a WAR, but in this case, all of the kernel components will be bundled later into a combined component package. Thus, this JAR creation step is intermediate to building the final component package. This assembly is further described in the next section.

Implementation dependencies come next. The first dependency (should fully for clarity) is on the Session Manager API (naturally). Several other dependences for the implementation are also included (declarations eliminated to save space, have a look at the code to see the full version).

Finally, maven is directed to look for the java sources of this API in the "src/java" directory under the module-part, which is "kernel/session-component" in this example.

When maven encounters this module part (and the project.xml file above), it compiles the API code and packages it as a JAR leaving it in the local target directory. This will subsequently be collected into a combined component package.

5.2.3 Combined Component Packaging

Certain aspects of a system like Sakai are best kept together for release and development purposes. Web applications (Sakai Tools) are the obvious example of this (see [5]), but collections of components are also useful. The combined component package provides support for this and the Sakai kernel is a good example of it.

The kernel-components directory is laid out like this:

```
kernel-components
        src/java/webapp/WEB-INF
                components.xml
        project.xml
```

Note that this module-part doesn't contain any source files. The whole point is to take code created elsewhere and created a combined package. Let's have a look at the

project.xml file (once again edited for size) that maven uses to build this combined package:

```
<project>
	<pomVersion>3</pomVersion>
	<name>Sakai Kernel Components Package</name>
	<groupId>sakaiproject</groupId>
	<id>sakai-kernel-components</id>
	<currentVersion>2.0.0</currentVersion>

	<properties>
		<deploy.type>components</deploy.type>
	</properties>

	<dependencies>
		<dependency>
			<groupId>sakaiproject</groupId>
			<artifactId>sakai-session-component</artifactId>
			<version>${pom.currentVersion}</version>
			<properties>
				<war.bundle>true</war.bundle>
			</properties>
		</dependency>

		Other dependencies include:
			sakai-id-component
			sakai-configuration
			sakai-thread_local-component
			sakai-tool-component
			sakai-active-tool-component
			commons-id
			concurrent
			sakai-util-xml
			sakai-util-java
			kernel-config-component
	</dependencies>
</project>
```

If you've been reading along, this should start to look familiar to you. The layout of the maven project is the same starting with name, groupId, id, and version elements.
The <deploy.type> property is a something new, however. In this case, it has a value of component. This is a signal to maven (via the Sakai maven plugin) to create a combined component package. Rather than creating a java (JAR) or web (WAR) archive, it creates a directory in the Tomcat/compoments directory. Each directory here is essentially an expanded WAR in that it includes a manifest file (describing the pieces of this package) and a WEB-INF directory that contains a components.xml file and the various components.

As before, implementation dependencies come next. These dependencies are defines much like other maven dependencies with one exception. Each dependency element also contains a property (war.bundle) that indicates whether this dependence should be included in the assembled package (expanded WAR) or not. One dependency for sakai-session-component is included to illustrate how these are declared, followed by a list of other dependencies. One interesting thing to note is that this list of dependencies documents what is actually included in the collection of kernel components.

The first dependency (should fully for clarity) is on the Session Manager API (naturally). Several other dependences for the implementation are also included (declarations eliminated to save space, have a look at the code to see the full version).

Finally, maven is directed to look for the java sources of this API in the “src/java” directory under the module-part, which is “kernel/session-component” in this example.

When maven encounters this module part (and the project.xml file above), it combines the JARs identified by the dependencies and puts them into the combined component package, “Tomcat/compoments/sakai-kernel-compoments” in this case.

The story is not quite finished at this point, however. When the Component Loader loads these combined component packages at Tomcat startup time, it needs a way to register each component implementation to the corresponding API. This is accomplished by the components.xml file (edited for clarity):

```
<beans>
	<bean id="org.sakaiproject.api.kernel.config.ServerConfigurationManager"
	class="org.sakaiproject.component.kernel.config.KernelConfigurationService"
	init-method="init"
	destroy-method="destroy"
	singleton="true">
		<property name="registrationPath">
			<value>registration.xml</value>
		</property>
		<property name="toolOrderFile">
			<value>toolOrder.xml</value>
		</property>
	</bean>

	<bean id="org.sakaiproject.api.kernel.id.IdManager"
	class="org.sakaiproject.component.kernel.id.UuidV4IdComponent"
	init-method="init"
	destroy-method="destroy"
	singleton="true">
	</bean>

	<bean id="org.sakaiproject.api.kernel.thread_local.ThreadLocalManager"
	class="org.sakaiproject.component.kernel.thread_local.ThreadLocalComponent"
	init-method="init"
```

```
        destroy-method="destroy"
        singleton="true">
        </bean>

        <!-- register one component as both the ToolManager & ActiveToolManager -->
        <bean id="org.sakaiproject.api.kernel.tool.ActiveToolManager"
        name="org.sakaiproject.api.kernel.tool.ToolManager
        org.sakaiproject.api.kernel.tool.ActiveToolManager"
        class="org.sakaiproject.component.kernel.tool.ActiveToolComponent"
        init-method="init"
        destroy-method="destroy"
        singleton="true">
        <property name="threadLocalManager">
        <ref bean="org.sakaiproject.api.kernel.thread_local.ThreadLocalManager"/>
        </property>
        </bean>

        <bean id="org.sakaiproject.api.kernel.session.SessionManager"
        class="org.sakaiproject.component.kernel.session.SessionComponent"
        init-method="init"
        destroy-method="destroy"
        singleton="true">
        <property name="idManager">
                <ref bean="org.sakaiproject.api.kernel.id.IdManager"/>
                </property>
                <property name="threadLocalManager">
                <ref
                bean="org.sakaiproject.api.kernel.thread_local.ThreadLocalManager"/>
                </property>
                <property name="inactiveInterval"><value>1800</value></property>
                <property name="checkEvery"><value>60</value></property>
        </bean>

</beans>
```

While there are a lot of other things going on in this file, the main purpose is to register a component as a bean using a name that corresponds to the interface used to implement the component.

All of this happens in the <bean> tag parameters. Each bean element has an id, which will be used as the registration name (by the Component Manager later). This id should be the fully qualified interface name, “org.sakaiproject.api.kernel.config.ServerConfigurationManager” for example. This is followed by the name of the class that implements it, “org.sakaiproject.component.kernel.config.KernelConfigurationService”. An initialize method called “init” is defined and should be called at startup time. And a destroy-method is defined called “destroy” that will be called at shutdown time. Finally, there is a

property that indicates if this should be loaded as a singleton or not, which is true in virtually all Sakai service cases.

Properties can also be used to provide configuration and other information to the registered components. In the Configuration Manager, for example, a property is defined for a tool order file. The value defined here is injected into the ServerConfigurationService instance at startup time and can be subsequently accessed. Service references can also be injected. The ThreadLocalManager is injected into several components above.

5.2.4 Resource Packaging

Just as it's useful to package components together, it can also be useful to package file-based resources together as well. The Sakai Kernel Bundle includes an example of resource packaging in the LoginTool. Resources related to the login page are contained in the login/bundle module-part:

```
login/bundle
        src/bundle
                auth.properties
                auth_ko.properties
                auth_zh.properties
        project.xml
```

This is a bundle of three properties files that contribute to authentication.

The project.xml (edited) file tells maven how to create the package and deploy it:

```
<project>
        <pomVersion>3</pomVersion>
        <name>Sakai Legacy Bundle</name>
        <groupId>sakaiproject</groupId>
        <id>sakai-legacy-bundle</id>
        <currentVersion>2.0.0</currentVersion>

        <properties>
                <deploy.type>jar</deploy.type>
                <deploy.target>shared</deploy.target>
        </properties>

        <dependencies>
        </dependencies>

        <build>
                <sourceDirectory>src/java</sourceDirectory>

                <!-- other resources for the jar - properties and xml files-->
```

```
                <resources>
                        <resource>
                                <directory>${basedir}/src/bundle</directory>
                                <includes>
                                        <include>**/*.properties</include>
                                </includes>
                        </resource>
                </resources>
        </build>

</project>
```

The package is defined to be part of the sakaiproject group and has an id of "sakai-legacy-bundle" indicating that it was originally drawn from the Sakai legacy architecture. A version number is given.

A property is defined to signal to maven that this package should be built as a JAR and deployed to "Tomcat/shared/lib". Deploying it here puts it into the classpath of all Sakai web applications, including the LoginTool.

Since these are simple files, they have no dependencies.

Finally, the file indicates to maven that the there are resources and they can be found in the base directory under "src/bundle" and that all property files should be included.

5.3 Getting a Component

While static service covers are available for most of the kernel services, injection is the recommended method to provide access. This documents the preferred implementation, but also allows it to be changed without modifying the code.

If service injection is used appropriately, there will be little or no need to directly access the Component Manager. If needed, however, the Component Manager itself can be injected [Verify that this is true.] into a tool or service or access via its static cover.

5.4 Coding Services as Singletons – Thread Safety

Implementations of any shared API will be made available to applications via the Component Manager as threaded singletons. Having a single instance of the code greatly reduces memory requirements, but does require that it be implemented in a thread safe manner. Service implementations should give careful consideration to critical regions of code, shared resources, blocking, and locking. More information will be available in "SKB Manual – Context and Thread Safety", [4].

5.5 Service Injection

Ideally, objects that need to access Sakai services should be initialized with a reference to that service when the object is created or instantiated. Sakai uses the Spring Framework to allow these references to be injected before the object is accessed.

5.5.1 Service Injection in a Tool

Each tool has a web.xml file that defines it as a Tomcat web application. In this web.xml file, you can create a context parameter:

```
….
<context-param>
        <param-name>contextSharedLocation</param-name>
        <param-value>
                /WEB-INF/components.xml
        </param-value>
</context-param>
….
```

The components.xml file is typically used to define beans and properties to be initialized. The compoments.xml file in turn might look like this:

```
<beans>

        <bean class="org.sakaiproject.tool.mytool.MyTool"
        init-method="init"
        destroy-method="destroy"
        singleton="true">
        <property name="compMgr">
                <ref bean="org.sakaiproject.api.kernel.component.ComponentManager"/>
        </property>
        </bean>

</beans>
```

MyTool is written according to JavaBean guidelines. In particular, object properties have getters and setters with names that correspond to the property name. Here, a property named “compMgr” is initialized with the Component Manager.

This example needs to be verified with code.

5.5.2 Service Injection in a Service

See Combined Component Packaging above for an example of declared injection into a service.

Write a simpler example of service inject and describe it here.

5.6 Configuration Properties

Some are saved via the Sakai configuration system. Others are saved in the system properties, retrieved via “System.getProperty()”.

Most of the configuration properties are replicated into the System properties, however, values are not transferred at this time.

6 References

[1] Sakai Kernel Bundle – Overview, Mark J. Norton, Sept. 2005 [URL Here]

[2] Sakai Kernel Bundle Requirements, Mark J. Norton, Sept. 2005 [URL Here]

[3] Sakai's Component Manager API, Component Packaging, and the Underlying Spring Implementation, Glenn R. Golden, March 23, 2005 [URL here]

[4] SKB Manual - Context and Thread Safety, [to be written]

[5] SKB Manual – Tools, [to be written]

7 Document History

This document was created by Mark Norton. It represents research into the Component Manager (and related) software developed by Glenn Golden of the University of Michigan and includes many of the concepts initially described him in [1].

Date	Version	Who	Work
Sept. 29, 2005	1	Mark Norton	Initial version of the document. Overview and introduction. Interface definitions. Implementation description. Utility objects described. System configuration properties. Initial recommended practices.
11/9/05	2	Brigid Cassidy	Terminology changes; edits.

Appendix 1: Configuration Properties

The properties below can be accessed via the ComponentManager.getProperties() method. This is an aggregated set of properties collected from all installed services. Many of these properties have no use in the Sakai Kernel Bundle. Others merely serve as examples on how to include properties (menu collections, in particular).

Copyright Properties

Property Name	Typical Value	Notes
bottom.copyrighttext	(c) 2003, 2004, 2005 sakaiproject.org. All rights reserved.	Notice that appears at the bottom of the Sakai portal.
copyrighttype.1	Material is in public domain.	Copyright menu item 1
copyrighttype.2	I hold copyright.	Copyright menu item 2
copyrighttype.3	Material is subject to fair use exception.	Copyright menu item 3
copyrighttype.4	I have obtained permission to use this material.	Copyright menu item 4
copyrighttype.5	Copyright status is not yet determined.	Copyright menu item 5
copyrighttype.6	Use copyright below.	Copyright menu item 6
copyrighttype.count	6	Copyright menu item count.
copyrighttype.new	Use copyright below.	
copyrighttype.own	I hold copyright.	
default.copyright.alert	true	Display copyright alert
default.copyright	I hold copyright.	
newcopyrightinput	true	

URL Properties

Property Name	Typical Value	Notes
fairuse.url	http://fairuse.stanford.edu	Link to the Stanford fair use description.
myworkspace.info.url	/library/content/myworkspace_info.html	My workspace info URL.
news.feedURL	http://www.sakaiproject.org/cms/index2.php?option=com_rss&feed=RSS2.0&no_html=1	News feed (RSS) URL.
accessPath	/access	
helpPath	/help	
loggedOutUrl	/portal	
portalPath	/portal	

powered.url.1	http://sakaiproject.org	Powered by Sakai item 1
powered.url.count	1	Powered by Sakai item count.
server.info.url	/library/content/server_info.html	Server information URL (relative link).
serverId	*Name of this server*	This is set to the Id of the current server.
serverName	localhost	This is set to the name of the current server.
serverUrl	http://localhost:8080	This is the URL to the root of the current server.
webcontent.instructions.url	/library/content/webcontent_instructions.html	Web content instructions URL (relative link).

Miscellaneous Properties

activeInactiveUser	true	
auto.ddl	true	
container.login	false	
content.upload.max	20	Upload size limit in megabytes. ???
hibernate.dialect	net.sf.hibernate.dialect.HSQLDialect	
java.beep	false	
jdbc.defaultTransactionIsolation	java.sql.Connection.TRANSACTION_READ_UNCOMMITTED	
sitesearch.noshow.sitetype:		
smtp.enabled	false	
subjectsize	8	
version.sakai	2.0.0[506121]	Current version of Sakai.
version.service	2.0.0	
gatewaySiteId	!gateway	The gateway site identifier (legacy site service).

User Interface Properties
iconNames.1: humanities
iconNames.2: engineering
iconNames.3: pig
iconNames.count: 3
iconSkins.1:
iconSkins.2:
iconSkins.3: examp-u
iconSkins.count: 3
iconUrls.1: /library/icon/humanities.gif

iconUrls.2: /library/icon/engineering.gif
iconUrls.3: /library/icon/pig.gif
iconUrls.count: 3
powered.alt.1: Powered by Sakai
powered.alt.count: 1
powered.img.1: /library/image/sakai_powered.gif
powered.img.count: 1
skin.default: default
skin.repo: /library/skin
display.users.present: false
editViewRosterSiteType.1: project
editViewRosterSiteType.count: 1
emailInIdAccountInstru: To log in, you must first get a guest account. A guest account is a special kind of account that is used to give non-Sakai University members access to the general Sakai University web environment. The Sakai University web environment includes many tools and services, one of which is Sakai.
emailInIdAccountLabel: Guest(s) Email Address (external participants, e.g. jdoe@yahoo.com)
emailInIdAccountName: guest
news.title: Sakai News
noEmailInIdAccountLabel: Username(s)
noEmailInIdAccountName: username
notifyNewUserEmail: true
titleEditableSiteType.1: project
titleEditableSiteType.count: 1
top.login: true
ui.institution: Sakai Using Institution
ui.service: Sakai Based Service

Course Management Properties

termendtime.10: 20050801000000000
termendtime.11: 20051201000000000
termendtime.1: 20031201000000000
termendtime.2: 20040501000000000
termendtime.3: 20040801000000000
termendtime.4: 20040801000000000
termendtime.5: 20040801000000000
termendtime.6: 20041201000000000
termendtime.7: 20050501000000000
termendtime.8: 20050801000000000
termendtime.9: 20050801000000000
termendtime.count: 11
termiscurrent.10: false
termiscurrent.11: false
termiscurrent.1: false
termiscurrent.2: false
termiscurrent.3: false

termiscurrent.4: false
termiscurrent.5: false
termiscurrent.6: false
termiscurrent.7: false
termiscurrent.8: true
termiscurrent.9: false
termiscurrent.count: 11
termlistabbr.10: Su05
termlistabbr.11: F05
termlistabbr.1: F03
termlistabbr.2: W04
termlistabbr.3: Sp04
termlistabbr.4: SpSu04
termlistabbr.5: Su04
termlistabbr.6: F04
termlistabbr.7: W05
termlistabbr.8: Sp05
termlistabbr.9: SpSu05
termlistabbr.count: 11
termstarttime.10: 20050801000000000
termstarttime.11: 20050901000000000
termstarttime.1: 20030901000000000
termstarttime.2: 20040101000000000
termstarttime.3: 20040501000000000
termstarttime.4: 20040515000000000
termstarttime.5: 20040801000000000
termstarttime.6: 20040901000000000
termstarttime.7: 20050101000000000
termstarttime.8: 20050501000000000
termstarttime.9: 20050515000000000
termstarttime.count: 11
termterm.10: SUMMER
termterm.11: FALL
termterm.1: FALL
termterm.2: WINTER
termterm.3: SPRING
termterm.4: SPRING_SUMMER
termterm.5: SUMMER
termterm.6: FALL
termterm.7: WINTER
termterm.8: SPRING
termterm.9: SPRING_SUMMER
termterm.count: 11
termyear.10: 2005
termyear.11: 2005
termyear.1: 2003
termyear.2: 2004
termyear.3: 2004

termyear.4: 2004
termyear.5: 2004
termyear.6: 2004
termyear.7: 2005
termyear.8: 2005
termyear.9: 2005
termyear.count: 11
sitebrowser.termsearch.property: term
sitebrowser.termsearch.type: course
roster.available.weeks.before.term.start: 4
sectionsize: 3
courseSiteType: course
coursesize: 3

Some of the properties included here should be moved to resource bundles where they can be properly localized.