

Technical Report Sakai Project

The Sakai Technology Service Portability Profile for Java

March 29, 2004

**Craig Counterman
Glenn Golden
Rachel Golub
Mark Norton
Charles Severance
Lance Speelman**

**Comments to: csev@umich.edu
www.sakaiproject.org**



Introduction

The Sakai project is producing an extensible open-source learning management system that provides and uses a complete implementation of the OKI OSID standard interfaces for LMS portability. In addition Sakai will deploy the components of its learning management system using the uPortal enterprise portal technology.

This document describes the Sakai Technology Portability Profile for Java and how to write Sakai compliant tools and services in Java. Other profiles may be developed for other languages.

The elements of the TSPP are selected to ensure the long-term value, portability, and interoperability of the tools and services that are developed using the TSPP.

The Tool and Service Portability Profile includes:

- Everything needed to write Sakai code in Java
- Sakai Tool Development Guidelines
- Sakai Java Service Descriptions
- Sakai Java Service Development Guidelines
- Description of GUI Elements

One of the goals is to keep the TSPP relatively simple, and to limit the explicit dependencies of the tools and services to as few as possible with the idea that every additional explicit dependency is a possible constraint on portability.

Compliance with the Sakai TSPP

See previous comments in [TPP_v6.doc](#).

It is very important when writing a specification to define what it means to comply with the specification. It is not particularly meaningful for a service to be "compliant" with the TSPP because tools will use other specifications beyond the TSPP approved specifications (like Sockets or web-services). The TSPP is a contract between the framework and the tools and services. If the framework does not provide a described feature or capability or does not live up to the contract then the Framework can be declared as "not compliant". In a way, a tool or service *can* be "Sakai compliant" in the way it uses the Sakai interfaces. However because the tools and services use elements outside of the scope of Sakai, saying a tool or service is "Sakai compliant" essentially says nothing about the tool.

That said, the purpose of the Sakai TSPP is to provide a set of interfaces and rules that enable the portability of tools and services that use those interfaces

and capabilities. So, to the extent that a tool can use the Sakai interfaces and other highly portable elements of Java, the tool can depend on the Sakai framework to provide its interfaces across multiple Sakai compliant frameworks.

If other frameworks are developed to support Sakai tools and services, then there will be a need to certify those frameworks as Sakai compliant. Certification is not a simple process - there are significant legal and technical challenges in producing a specification and test suite suitable to truly certify framework as compliant.

Often the effort to produce a set of certification documents and the development of a rich set of tests to insure that that certification is actually meaningful is as large of an effort of producing the original specifications and first reference implementation. Sakai does have the advantage that in addition to producing a reference implementation, the Sakai project is producing a large number of tools and services which can serve as a de-facto certification mechanism until a more rich and focused certification mechanism can be developed.

When the phrase "for a tool to be compliant with the TSPP it must do X" is used, it means that the tool is properly using the TSPP or is compliant with that particular aspect of the TSPP - but nothing can be said as to whether the tool complies with the TSPP.

Specifications

One of the essential elements of a profile is to select a set of specifications, and add guidance, select options and define best practices around those services sufficient to insure that portable and interoperable code can be developed using the profile.

The specifications that form the foundation of the Sakai Technology Portability Profile include:

JavaServer Faces - In the early releases of Sakai, these will be stored in JSP files. It is important to note that the specification for a Sakai view is the XML representation of the JSF layout with no other JSP in the file - no HTML, no Java. In the future, Sakai may use actual XML files, instead of JSP files, to declare the views. To be TSPP-**Java** compliant, the JSF document must only use the approved JSF HTML, JSF core, and Sakai tag libraries. It is understood that some tools will require richer interfaces than those that are supported by the three approved tag libraries. The best approach to solve this problem is to work with the Sakai project to extend the approved Sakai tag library so that all applications can make use of the new capabilities. Another alternative is to build a tool-specific tag library. In this situation the developer of the tag library will be responsible for seeing that the tags are supported when the tools are moved from one framework to another.

OKI OSIDs - These APIs provide an integration layer that ensures portability of the tools and services across any environment that provides implementations for the OKI OSIDs. In addition, the OKI OSIDs provide interfaces where local implementations of the OSIDs can be developed to integrate OKI compliant tools and services into the local environment.

The Sakai Project will define its own internal specifications that are part of the Portability Profile - these specifications will build upon and add detail to the OKI and JSF Specifications to define their use within the Sakai Framework.

Sakai GUI Elements - Sakai will define additional JSF tags based on the Sakai Tool Development Style Guide. By using these tags, tool developers will automatically comply with the Sakai Tool Style Guide. These tags also insure a uniform look and feel across Sakai tools developed by different developers. It is also important to note that the Sakai mechanisms are in place to ensure that tools have a consistent look and feel that is under the control of the deploying institutions.

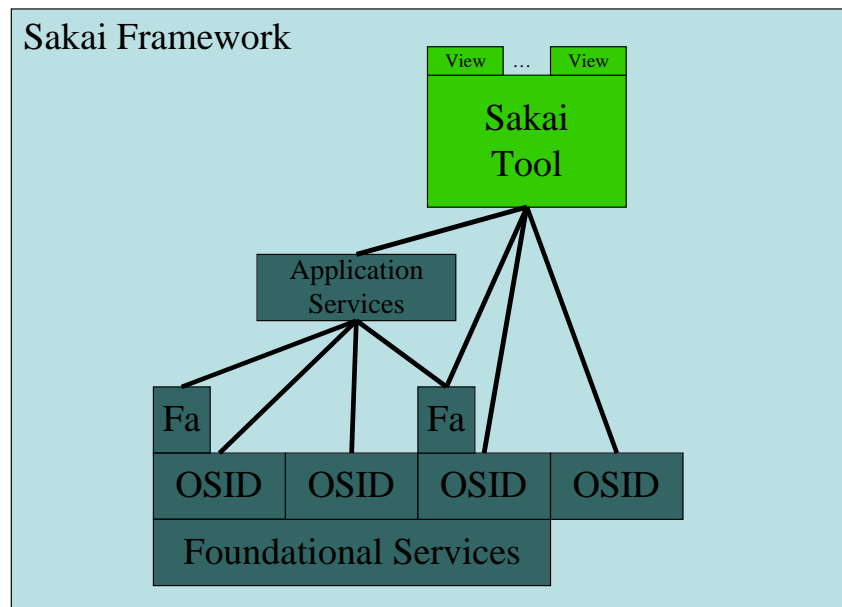
Sakai/OKI Façade Services - These will be new interfaces and implementations developed by the Sakai Project primarily to add a semantic layer on top of the OKI OSID interfaces and add schema-specific semantics to the OKI OSIDs which are designed to support a wide range of schemas. The purpose of the façade services is to provide a mechanism to insure interoperability in terms of the data used by Sakai tools using the OKI OSIDs to store and retrieve data. These Façade services will act as a layer on top of the OKI OSIDs - in order to move the Sakai tools to a different OKI compliant environment, one must also bring the façade services as well. The intent for the façade services is to be relatively "thin" - their purpose is not to capture business logic but instead to add a schema and semantic model and explicit typing to abstract data objects stored using the OKI OSIDs. The application services described below are the proper place to capture business logic.

In addition, another set of services will be developed which fall outside the Portability Profile per-se as they are more related to the Sakai produced tools and services which will be developed for deployment within the framework rather than being part of the framework:

Sakai Application Services - These will be new interfaces developed by the Sakai Project as part of the development of the tools that make up the Sakai Collaborative Learning Toolkit - The application classes will evolve as the needs of tools expand. The goal is to limit the amount of code that is place in the tools with respect to interacting with the storage services. The application Services are a convenient way to implement functionality once and use it across multiple tools.

Sakai Foundational Services - This a set of interfaces developed for the specific purpose of supporting the storage needs of the OSID implementations. The goal is to provide an abstract general-purpose storage layer which handles object-to-relational mapping, multi-system caching, scaling, and high-performance access. Not all OSID implementations will use the foundational services - sites can develop their own OSID implementations that replace the Sakai OSID implementations for particular services. These locally developed services may completely ignore the Sakai foundation services. The foundational services are not formally part of the TSPP but are included here to show the complete architectural picture.

As the figure below shows, while there is a logical layering in terms of the general purpose of each of the types of interfaces, much of the layering is neither required nor strict:



A tool can talk to any of the interfaces from the application directly to the OKI OSIDs. As a matter of fact, it is quite reasonable for some tools not to use application or façade classes at all and to instead go directly to the OKI OSIDs. The Sakai framework must support this direct access to OSIDs to accommodate tools that are developed to comply with the OSIDs but not initially developed in the Sakai environment.

Similarly an application service can use OSIDs directly or work through the façade classes.

Façade classes only communicate with a single OSID and have a one-to-one mapping with their underlying OSIDs.

Foundational services should only be used by the high-performance enterprise implementations of the Sakai OSIDs. Tools and application services should use the OSIDs for their storage needs rather than calling the foundation services. If developers begin to develop or modify an enterprise OSID implementation, then they will interact with the foundation services.

Tool Development Guidelines

This section describes how to write a Sakai tool. It looks at the interaction from the perspective of a tool writer. In other documents we will look at how the Sakai web-based framework is implemented to support the tool in terms of lifecycle, services and user interaction.

Many of the design choices are designed to move any non-portable aspects of the tool into the framework. Furthermore the tool will have very little control over the final look and feel of the user interface - the flip side of this is that the framework has almost complete control of the look and feel of the tools.

If we imagine a time when literally hundreds of tools have been independently written by many different organizations, allowing tool developers to specify the "chrome aspects" of their tools will result in a completely inconsistent set of tools which cannot be logically composed to make a system which meets the needs of its ultimate customer.

In short, if the tools are not forced to adhere to a standard, then each will be an expression of the design tendencies of the team developing the tool. In Sakai we intend to allow the same creativity of designers, but the effect is that their changes affect all tools allowing complete sites with very different looks and feels but within the site all the tools are completely consistent.

It is important to note that this Sakai framework relies on CSS for basic graphical elements - the Sakai framework uses CSS where appropriate. However the Sakai framework allows far more detailed control - for example, a site administrator can dramatically change the look and feel of every menu of every Sakai tool by changing a single file. The site can choose to use buttons, simple href-style text, walking menus in JavaScript, or even an ActiveX control of JAVA Applet for menus.

Separation of Graphic Layout from Tool Logic

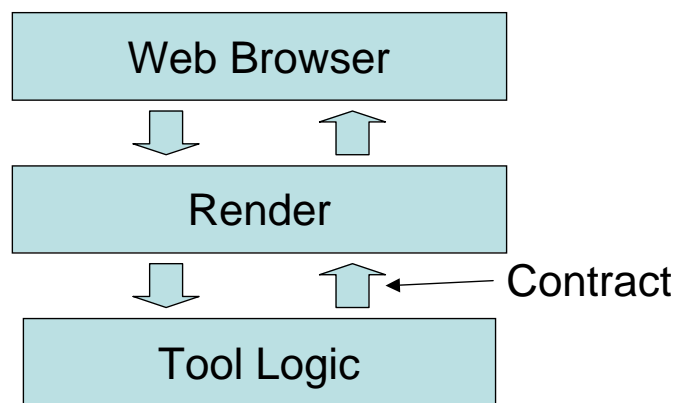
There is a distinct departure in Sakai from the recent approaches to the separation of the Graphic layout from the tool logic. The typical approach to accomplish this decomposition is to define two roles:

- User Interface Designer

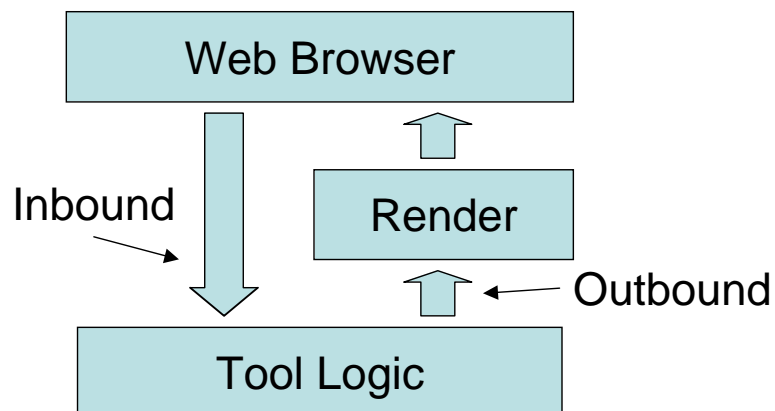
- Tool Developer

User Interface Designers would often use tools such as Dreamweaver, JSP, XLST, Velocity or another technology to define the HTML aspects of the layout. The tool developer would develop in a language such as Java, JSP, PHP, etc performing the database access, and maintaining the state of the interaction between the tool and the user

To facilitate the separation of concerns and to allow independent development of the GUI and tool logic, it is necessary to establish a contract in the form of an XML document or set of JAVA objects which the tool hands to the rendering component.



However, typically the contract is not symmetric because in a web-environment the "incoming" variables are provided as part of the HTTPRequest object and the naming of those parameters is a further part of the contract. In many of these arrangements the request object is passed directly to the tool logic as shown below:



There are several problems that appear with this approach:

- The inbound and outbound contracts are expressed using different technologies. The outbound contract may be in XML while the inbound contract is usually expressed in the names of HTML form elements. The GUI developer must implement the inbound contract and distribute the elements of the inbound contract throughout the render for the particular tool.
- As a tool is moved from a servlet to a portlet environment there may be constraints on some aspects of the HTML requiring some of the contract to be altered as a tool moves from one environment to another.
- The outbound contract in XML is usually flexible and easy to extend while the inbound contract is usually more brittle and requires more coordination between the tool and GUI developers during changes.

Ultimately, the flaws all come down to a basic lack of symmetry between the outbound and inbound contract.

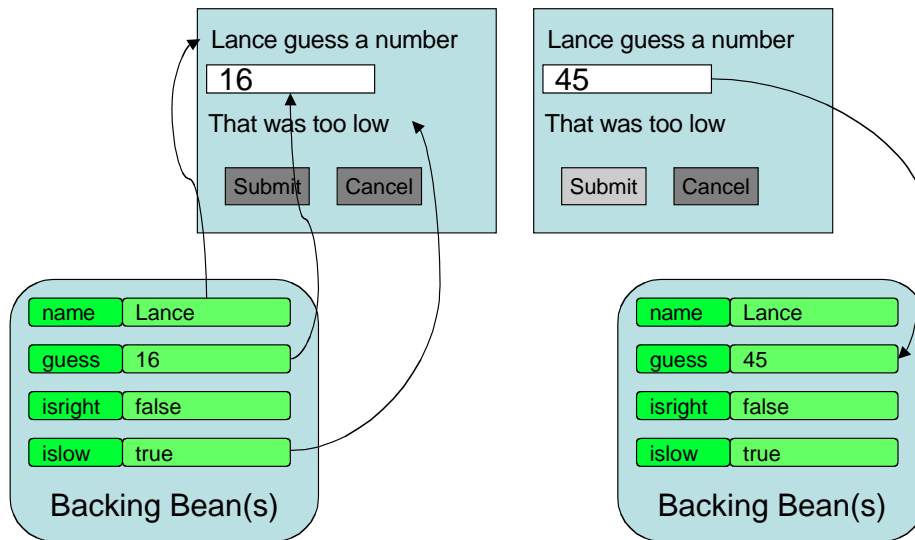
A number of new and popular technologies such as Struts have improved the situation by allowing the inbound contract to be expressed in the form of bean setter methods rather than direct dependency on the request object values.

JavaServer Faces - Symmetric View Technology

Many view JavaServer Faces as the follow-on or next-generation of the highly successful Struts approach. In addition to using a bean paradigm for the inbound contract, JavaServer Faces uses the bean paradigm as the outbound contract as well. JSF defines a new language using a JSP tag library which makes references to bean elements as part of its syntax.

During the outbound rendering phase, JSF pulls the variable information from the bean methods. During the inbound phase, new values are placed in the beans using the bean setter methods.

In the diagram below, the outbound values are placed in a bean that is called a "Backing Bean" as it is what "backs" the view. The JSF render process extracts the bean contents according to the layout and renders the proper HTML. The field names are not part of the layout at all. JSF generates the field names so that it is able to parse them when the user presses a button and submits a request.



When the JSF framework processes the incoming request, it uses the named fields to determine which bean values are to be set. JSF sets the beans and then processes the event.

By delegating the form field names to JSF and allowing JSF to receive the http request, neither the GUI designer nor the tool developer need any awareness of the (very HTML centric) names of the form elements. The interface is far less brittle than the asymmetric rendering approach described above.

For Struts programmers these concepts will be very familiar. JSF can be viewed as a richer, cleaner version of Struts with stronger focus on complete symmetry in the input and output contracts.

This also makes JSF a much cleaner layer in terms of allowing multiple implementations of the JSF render kit targeting different environments including non-HTML environments. When the Tool and GUI layout elements cannot even detect whether or not they are operating in an HTML environment, then there is some hope of portability between HTML and non-HTML environments. It also reduces the number of dependencies (import statements) that a tool must use. For example, no Sakai tool will need to import `javax.servlet.*` because all of that is handled by the JSF implementation and by the time the "request" arrives at the tool, all the data is simply sitting in a tool.

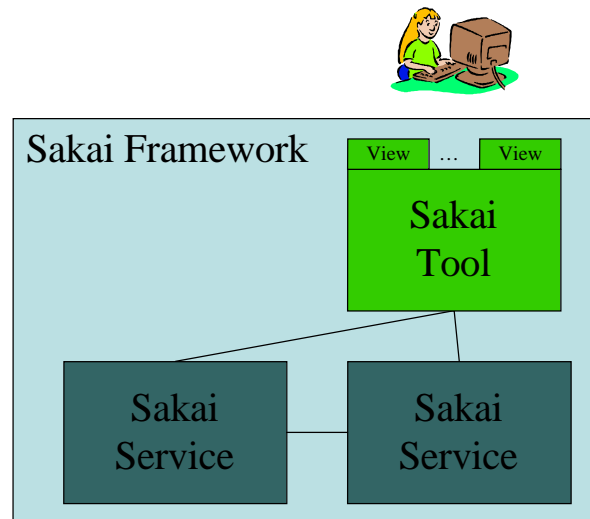
How do I use JSF in my tool?

How do I create a new Sakai GUI element using JSF?

Components of a Sakai Tool

This is when the term 'view' is introduced. What is a view?

A Sakai tool consists of tool logic written in Java, one or more views, and usually accesses some services to access persistent data or other information which or common processing logic which is separate from the tool.



Multiple views are typically used when the tools has several different modes depending on the actions being performed such as edit mode, list mode, or wizard step 4.

Overview of the View

Views are expressed in XML syntax as specified by the JSF and Modified by the Sakai GUI toolkit. The following is a small snippet:

```
<sakai:tool_bar>
  <sakai:tool_bar_item
    action="#{AnnouncementTool.processActionListNew}"
    value="#{msgs.annc_list_new}" />
  <sakai:tool_bar_item
    action="#{AnnouncementTool.processActionListDelete}"
    value="#{msgs.annc_list_delete}" />
...
<sakai:flat_list value="#{AnnouncementTool.announcements}" var="annc">
  <h:column>
    <f:facet name="header">
      <h:output_text value="" />
    </f:facet>
    <h:selectboolean_checkbox value="#{annc.selected}" />
  </h:column>
```

In the first section, the `tool_bar_item` is defining a method to be called when a particular action is pressed. In the `flat_list` tag, the actual bean (`AnnouncementTool`) returns a collection (via `getAnnouncements()`) which is then iterated and each element of the collection is given a row in the output and the

iteration variable (**annc**) is used to determine whether or not the particular announcement in the list is selected (via the **annc** object's **isSelected()**).

It is important to remember that this syntax is symmetric. When the view is rendered outbound, the collection is iterated, and the selected value determines the current value of the checkbox as displayed to the user. However, if the user changes the value of the checkbox and submits the screen, JSF goes back through the collection and *updates* the selected flag on the members of the collection using the bean setter method.

In the TP1 version of the Sakai framework these views are stored in JSP files and rendered using the Sun implementation of JSF which uses a taglib to define the custom JSF and Sakai tags. Given that these are in JSP files, it would be technically possible to mix Java syntax with the layout information. However, Sakai explicitly forbids the use of Java or HTML in the view files to insure portability and cross-tool consistency.

At the beginning of the Sakai project, JSF is an emerging technology. JSF is both a standard for expressing layout and an implementation from Sun that renders that layout. There are other efforts to develop JSF render kits implementations which support the JSF layout description but are not implemented using JSP. If Sakai tools add Java code to the JSP, it severely limits the ultimate portability of the tools and the ability to implement a wide variety of frameworks [future ref].

The best way to think of the views is that they are XML documents that happen to be stored in a JSP file. In future Sakai releases, views will likely be expressed using XML files instead of JSP files.

Actions are expressed as methods that are called when the particular button is pressed. In this example, **AnnouncementTool.processActionListNew()** is called when the "New" entry in the toolbar is pressed.

Specifically, how do I use this capability in my tool? Give an example.

Overview of the Tool Logic

The tool logic is written as a class in Java and includes both the bean information to back the view as well as the tool logic to react to the actions requested by the user.


```
public class AnnouncementTool
{
    public class Annc
    {
        private AnnouncementMessage announcement = null;
        private boolean selected = false;
        public boolean getSelected() { .. }
    }
}
```

```

    public void setSelected(boolean b) { .. }
    public void setAnnouncement(AnnouncementMessage message) { .. }
    public AnnouncementMessage getAnnouncement() { .. }
}
public List getAnnouncements() { ... }
public setAnnouncements(List annnc) { ... }
public String processActionListNew()
{
    ...
    return "edit";
}
}

```

The AnnouncementTool object, with its inner Annnc class, along with the getters and setters and actions, is the backing bean for the view. Notice that **getAnnouncements()** returns a List of **Annnc** elements. Annnc is a decorator class over the Announcement Service's **AnnouncementMessage** entity class. Each Annnc object adds extra information to that which is stored in the announcement variable: the **selected** boolean.

New...	Delete	Revise		Merge...	Options...	Permissions
	Subject	Site	From	Date 		
<input type="checkbox"/>	another	aSite	Some User	20040206022118969		
<input type="checkbox"/>	A Test	aSite	Some User	20040203030018064		
<input type="checkbox"/>	Welcome	aSite	Some User	20020918010411296		

The purpose of the decorator information is simply to interact with the view and to act as a backing store for a GUI element such as a checkbox associated with the item in the list. In a MVC sense the decorator(s) are added to the elements of the domain model (from the persistence) forming the model which is part of the View.

Expand the MVC acronym to Model-Value-Component.

Note that the action method **processActionListNew** takes no parameters. This is because any data from the request has already been placed in the backing bean before the action method has is called. The return value from the action method is called a "logical outcome"; an application defined term that indicates the outcome of the action. The primary purpose of the logical outcome is to select the view for the next rendering step.

The views are defined and selected in an XML configuration file. The file used by JSF is the faces-config.xml. This file defines several things:

- The location of any message bundles used for Internationalization
- The managed bean information including any service dependencies
- The list of views and the relationship between the outcomes and the views

How are dependencies on services described?

We will look at this file in more detail below but the selection of the view is defined as follows:

```
<navigation-rule>
  <from-view-id>/annc/list.jsp</from-view-id>

  <navigation-case>
    <description></description>
    <from-outcome>edit</from-outcome>
    <to-view-id>/annc/edit.jsp</to-view-id>
  </navigation-case>
...
```

The outcome selects the new view based on the current view. There are a number of approaches to this - a simple approach is to simply use the outcome to select the next view regardless of the current view. This is done by omitting the <from-view-id> tag from the navigation-rule causing the rule to apply to any from-view.

JavaServer Faces

Much of the GUI aspects of Sakai are based on the standard JavaServer Faces elements, but Sakai provides a complete set of elements to use. Sakai adds the rule of not allowing HTML or JAVA in the JSP declaration of the component tree.

JSF is a rich environment for development and processing of interface elements. It handles things from the start of the request from the user (after the user presses a button or link to a JSF view) by parsing the incoming request, applying the values to the components in the UI tree, validating and converting the strings used in the user interface, applying the values to the backing beans, and invoking the action methods.

JFS also handles the response, constructing the proper component tree for the selected view, populating the tree with values from the backing beans, and invoking the installed and selected RenderKit to produce the markup that is sent to the user.

Internationalization

It is important to develop tools from the beginning to support multiple languages. JSF supports multiple languages using the standard language bundle approach. First, the strings that make up the messages from the program are placed in a set of property files. The first is the default language Messages.properties:

```
annc_list_title=Announcement List
annc_list_new=New...
annc_list_edit=Revise
```

...

Then translated versions of the strings are placed in a language-specific file such as `Messages_fr.properties`:

```
annc_list_title=Announcement List
annc_list_new=New...
annc_list_edit=Revise
...
```

These messages can be referenced from the view files by loading them into a bean that can be referenced throughout the View. To associate the message bundle with the tool, the following is added to the `faces-config.xml` file:

```
<faces-config>
  <application>
    <message-
bundle>org.sakaiproject.tool.annc.bundle.Messages</message-bundle>
    <locale-config>
      <default-locale>en</default-locale>
      <supported-locale>de</supported-locale>
      <supported-locale>fr</supported-locale>
      <supported-locale>es</supported-locale>
    </locale-config>
  </application>
  ...
```

In the View file, the message bundle is loaded into a bean and then referenced throughout the view using the bean:

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://sakaiproject.org/jsf/sakai" prefix="sakai" %>

<f:loadBundle
  basename="org.sakaiproject.tool.annc.bundle.Messages" var="msgs"/>

<f:view>
<sakai:view_container title="#{msgs.annc_list_title}">
...

```

The `loadBundle` tag knows the requested localization and loads the appropriate bundle and associates it with a local bean (**msgs**). This bean can be accessed throughout the view to retrieve the messages in the proper language.

In the Java code, the messages can be extracted from the bundle using the following sequence:

```
import com.sun.faces.util.MessageFactory;

String msg =
  MessageFactory.getMessage( FacesContext.getCurrentInstance(),
```

```
"annc_list_title", null);
```

The Sun message factory is used to query JSF (driven by the faces-config.xml file) to find the appropriate message within the bundle given the current localization settings for this particular user.

Does this handle UTF-16? How will ideographic languages be displayed? Do I need to make special provisions for this kind of support? What about other I18N issues like dates, money, decimal notation, etc.?

Service Injection and Using Services

Services are a fundamental concept of the Sakai architecture - services handle all persistence, data modeling, and business logic to allow the tool code to focus on the management of the presentation to the user. It is necessary for a tool to be able to find the appropriate service implementation at run-time that satisfies the appropriate Interface needed for the service.

This configuration information is stored in the faces-config.html and is part of the Managed bean support. The classes that are the appropriate implementations for this installation are injected as follows:

```
<managed-bean>
  <description>Tool Bean for announcements</description>
  <managed-bean-name>AnnouncementTool</managed-bean-name>
  <managed-bean-class>
    org.sakaiproject.tool.annc.AnnouncementTool</managed-bean-class>
  ...
  <managed-property>
    <description>Service Dependency: announcement service</description>
    <property-name>announcementService</property-name>
    <value>#{ComponentManager.components
      ["org.sakaiproject.osid.announcement.AnnouncementService"]}
    </value>
  </managed-property>
</managed-bean>
```

Within the code, there is a setter and getter method for **announcementService**:

```
import org.sakaiproject.osid.announcement.AnnouncementService;

public class AnnouncementTool
{
  /** Dependency: The announcement service. */
  protected AnnouncementService announcementService = null;

  public void setAnnouncementService
    (AnnouncementService announcementService) { .. }
  protected AnnouncementService getAnnouncementService() { ... }
}
```

When JSF is creating the **AnnouncementTool** object, it creates it as a managed-bean and calls the setters as appropriate. Within the faces-config.xml file, there is a call to another managed-bean called **ComponentManager**, part of the Sakai framework component manager, which has a list of all of the correctly registered service implementations indexed by the name of the desired interface. As a result the implementing interface is injected as a side effect of the managed-bean creation process.

Within the application, the **getAnnouncementService()** getter method is used to access all the methods in the interface.

```
AnnouncementChannel channel =
    getAnnouncementService().getAnnouncementChannel(getChannel());
```

The combination of these mechanisms allows a tool to access its services without any explicit dependencies on the framework. The only dependency is on the service interface - which is the way it should be.

Sakai Service Descriptions

OKI OSIDs - These APIs provide an integration layer that ensures portability of the tools and services across any environment that provides implementations for the OKI OSIDs. In addition, the OKI OSIDs provide interfaces where local implementations of the OSIDs can be developed to integrate OKI compliant tools and services into the local environment.

Exactly, what OSIDs will be included with Sakai?

The Sakai Project will define its own internal specifications that are part of the Portability Profile - these specifications will build upon and add detail to the OKI and JSF Specifications to define their use within the Sakai Framework.

Sakai/OKI Façade Services - These will be new interfaces and implementations developed by the Sakai Project primarily to add a semantic layer on top of the OKI OSID interfaces and add schema-specific semantics to the OKI OSIDs which are designed to support a wide range of schemas. The purpose of the façade services is to provide a mechanism to insure interoperability in terms of the data used by Sakai tools using the OKI OSIDs to store and retrieve data. These Façade services will act as a layer on top of the OKI OSIDs - in order to move the Sakai tools to a different OKI compliant environment, one must also bring the façade services as well. The intent for the façade services is to be relatively "thin" - their purpose is not to capture business logic but instead to add a schema and semantic model and explicit typing to abstract data objects stored using the OKI OSIDs. The application services described below are the proper place to capture business logic.

What façade classes are being considered?

In addition, another set of services will be developed which fall outside the Portability Profile per-se as they are more related to the Sakai produced tools and services which will be developed for deployment within the framework rather than being part of the framework:

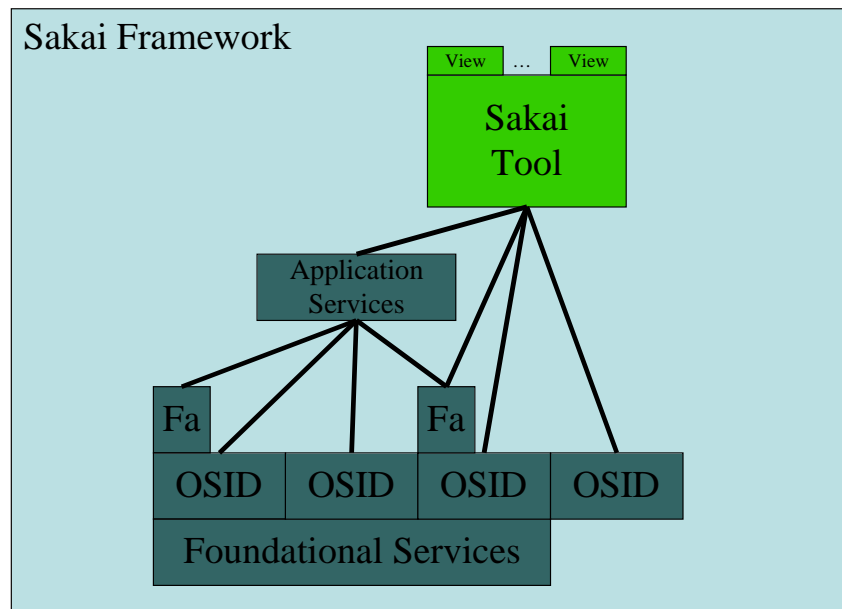
Sakai Application Services - These will be new interfaces developed by the Sakai Project as part of the development of the tools that make up the Sakai Collaborative Learning Toolkit - The application classes will evolve as the needs of tools expand. The goal is to limit the amount of reusable business logic that is placed in the tools. The Application Services are a convenient way to implement functionality once and use it across multiple tools.

Exactly, what are the Sakai application services?

Sakai Foundational Services - This a set of interfaces developed for the specific purpose of supporting the storage needs of the OSID implementations. The goal is to provide an abstract general-purpose storage layer which handles object-to-relational mapping, multi-system caching, scaling, and high-performance access. Not all OSID implementations will use the foundational services - sites can develop their own OSID implementations that replace the Sakai OSID implementations for particular services. These locally developed services may completely ignore the Sakai foundation services.

Exactly, what are the Sakai foundational services?

As the figure below shows, while there is a logical layering in terms of the general purpose of each of the types of interfaces, much of the layering is neither required nor strict:



A tool can talk to any of the interfaces from the application directly to the OKI OSIDs. As a matter of fact, it is quite reasonable for some tools not to use application or façade classes at all and to instead go directly to the OKI OSIDs. The Sakai framework must support this direct access to OSIDs to accommodate tools that are developed to comply with the OSIDs but not initially developed in the Sakai environment.

Similarly an application service can use OSIDs directly or work through the façade classes.

Façade classes only communicate with a single OSID and have a one-to-one mapping with their underlying OSIDs.

Foundational services should only be used by the high-performance enterprise implementations of the Sakai OSIDs. Tools and application services should use the OSIDs for their storage needs rather than calling the foundation services. If developers begin to develop or modify an enterprise OSID implementation, then they will interact with the foundation services.

Service Framework

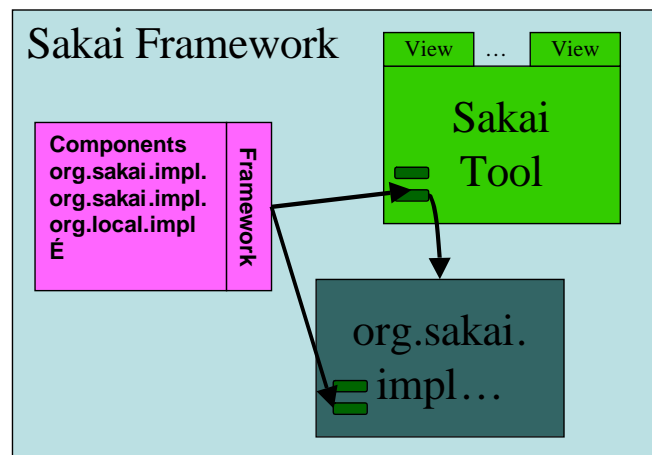
Usually in a system where there is a dynamic mapping of implementation classes to interfaces it is necessary to specify a service framework that can be used so that tools can locate services, and so that services can locate other services. Traditionally, the pattern used to solve this problem is for the service framework to provide an API that can be used to "look-up" the framework configured implementation for a particular interface.

This is called the "service locator" pattern and there are a number of examples of the use of this pattern:

- Jakarta Turbine
- Jakarta Avalon
- OKI OSID Loader
- IBM WebSphere

The problem with this pattern is that it introduces a dependency on the particular chosen framework because each framework invents its own method syntax, types, and parameters even though the underlying functionality is nearly always the same. As a result the tool or service must import packages specific to a particular framework. This explicit dependency on the framework is a few lines of code that is used in nearly every tool and service.

A new generation of frameworks typified by Spring and Pico removes this explicit dependency by allowing the tool or service express its dependencies either using a Java Bean setter method or constructor. When the tool or service object is being created, the framework examines it for the external dependencies and "injects" the implementation classes in via the constructor or bean-setter methods, making the necessary linkups. As the object begins execution, all of its dependencies are already satisfied and stored in local variables.



This approach makes the tool/service dependency resolution an implicit dependency rather than an explicit dependency. So while the bean-setter pattern was initially made popular using the Spring framework, it could be performed by some completely different framework such as a web-services based framework.

The Sakai Portability Profile includes bean-setting dependency as its best practice approach to dependency resolution. The bean-setting pattern is far more flexible when using containers (such as JSR-168 or JSF) that perform the actual construction of the object. This is in contrast to implementing constructor-

style injection in a JSR-168 or JSF environment where one must revise JSF or JSR-168 so that it properly performs the service injection at constructor time. This is not practical in situations where you either do not have source to a component (such as JSF) or do not want to make modifications to a component (such as JSR-168) so as not to end up with a forked-source tree.

The bean-setter pattern allows dependencies to be injected after the constructor has been completed but before the object has been activated. Using this approach allows the opportunity to make clever use of existing features (such as the fact that every JSF tool is also a managed bean) to satisfy the necessary dependencies using standard mechanisms.

The Sakai framework easily supports constructor style and service locator mechanisms in addition to the bean-setter method, because there are situations where these are preferred, but the bean-setter method is the recommended best practice.

Sakai Service Development Guidelines

This is how to write services. Look at the tool section, and come up with representative code snippets.

How do I write a service? I could describe the process I'm using with the Presentation Service.

Sakai GUI Element Descriptions

(Consider putting this into a separate document)

JavaServer Faces - In the early releases of Sakai, these will be stored in JSP files. It is important to note that the specification for a Sakai view is the XML representation of the JSF layout with no other JSP in the file - no HTML, no Java. In the future, Sakai may use actual XML files, instead of JSP files, to declare the views. To be TSPP compliant, the JSF document must only use the approved JSF HTML, JSF core, and Sakai tag libraries. It is understood that some tools will require richer interfaces than those that are supported by the three approved tag libraries. The best approach to solve this problem is to work with the Sakai project to extend the approved Sakai tag library so that all applications can make use of the new capabilities. Another alternative is to build a tool-specific tag library. In this situation the developer of the tag library will be responsible for seeing that the tags are supported when the tools are moved from one framework to another.

How do I make a new one?

Sakai GUI Elements - Sakai will define additional JSF tags based on the Sakai Tool Development Style Guide. By using these tags, tool developers will automatically comply with the Sakai Tool Style Guide. These tags also insure a uniform look and feel across Sakai tools developed by different developers. It is also important to note that the Sakai mechanisms are in place to ensure that tools have a consistent look and feel that is under the control of the deploying institutions.

Some of the Elements being considered are:

Input Widget

- wysiwyg text input box
- audio record
- upload widget
- color picker
- add/subtract from list - side by side lists to/from
- calendar date entry widget
- resource picker for sakai files/content search

Messages

- warning/alert/error messages
- immediate in your face alert - push technology
- confirmation messages
- how alert that fields are improperly filled in
- waiting - something is happening widget
- tool tip text - what do you show where

Tables/Lists

- sorting of lists
- tables
 - header, jump indices, navigation, highlighting
- rows
- hierarchal tables w/ dropdown gadgets (open / close)
- folder hierarchy view
- date/event list display

threaded list (e.g. discussion)
matrix widget (e.g., for permissions)
collapsing lists
search results
image gallery

Navigation

nav bar
paging
breadcrumb
hierarchy navigation
wizard navigation

Page Layout

footer
chunk and chunk headings (like
navigo)
page headers (and subheads)
tabs
toolbar - top of tool / drop down
menus
page indicator - nav bar highlights
inline info
text layout for various elements
rollovers - change colors
how required fields are indicated
scrolling text box for presentation
related links
fields in a form, read vs write mode

Times/Dates

day/week/month at a long
glance

Interaction

action buttons

view changing / student preview
return/cancel/exit/done conventions
selecting - single and multiple /
checkboxes
url grabber

Status

what have I already read
indicator
what is new indicator
presence widget

Miscellaneous

xslt widget
video/audio players
slide show

Summary

Re-write the summary based on the Sakai TSPP for Java material.

Acknowledgements

This document is the result of the Sakai Architecture Research Group including: Craig Counterman, MIT, Rachel Gollub, Stanford, Mark Norton, Sakai Educational Partnership Program, Charles Severance, University of Michigan, Lance Speelman, Indiana University, Curt Seifert, Indiana University, Bryan McGough, Indiana University, Glenn Golden, University of Michigan, Ken Weiner, uPortal, and many others from the core Sakai institutions.

In addition, a number of groups and individuals helped significantly in early review of this material including teams from the NMI Grid Portals project team, Cambridge University and Berkeley University as well as productive discussions with Benjamin Maillistat who is the project architect of the eXo Portal project and Jason Novotny who is the architect of the GridSphere project.

References

Tomcat Servlet Container (jakarta.apache.org)

JavaServer Pages and standard tags library (java.sun.org/jsp)

JavaBeans (java.sun.com/products/ejb)

Spring Framework (www.springframework.org)

OKI Open Service Interface Definitions (www.sourceforge.net/okiproject)

JavaServer Faces (java.sun.com/j2ee/javaserverface)