# Technical Report
# Sakai Project

## Sakai Java Framework

## October 24, 2004

**Version 1.0**

**Craig Counterman**
**Glenn Golden**
**Rachel Gollub**
**Mark Norton**
**Charles Severance**
**Lance Speelmon**

**www.sakaiproject.org**

# 1 Introduction

This document describes the Sakai Java Framework that is one possible implementation of the Sakai Abstract Architecture. This document describes particular technologies chosen for the layers identified in the Sakai Architecture that leads to the Sakai Tool Portability Profile. The Tool Portability Profile (TPP) is more detailed guidance with respect to the development of tools that will operate within the Sakai Java Framework.

*It is important to note that this document is only intended to reflect the best thinking and information for a particular release of Sakai. The version of this document (see title page) is tied to a particular Sakai release. These documents should be expected to change in some possibly significant ways during the early Sakai releases. Because the early phases of the Sakai project are using legacy capabilities coming from the CHEF environment, development and tool deployment will be a mix of legacy and new Sakai capabilities throughout through Sakai 3.0 (late 2005). In each successive release, the focus will increasingly be on the non-legacy aspects of Sakai.*

*Until version 2.0 the Sakai Java Framework is in a state of flux. Releases prior to 2.0 are intended to provide functionality that can be installed and used to provide a collaborative and learning environment. However, the official 1.0 and 1.5 releases should not be used as the basis for significant development. Not all the elements of the framework will be present in the official releases because they may not be production ready. Those wanting to work with the framework and build new tools should use the CVS archive rather than the source code provided by the releases so as to work the most up-to-date code supporting new framework capabilities.*

*Users adopting early releases of Sakai should be prepared to track Sakai evolution through the public mailing lists and the Sakai Educational Partners Program described at www.sakaiproject.org.*

## 1.1 Purpose of this Document

This document describes a design based on the Abstract Sakai Architecture for a Java-based framework that allows tools and services to leverage the powerful support provided by existing web technologies. In particular, it describes a suite of technologies that allow Sakai Services and Tools to be created in a manner that promotes interoperability and code portability.

## 1.2 Definition of Terms and Acronyms

OKI, Open Knowledge Initiative
OSID, Open Service Interface Definition
OBA, Out of Band Agreement
JSF, JavaServer Faces
JSP, JavaServer Pages
Servlets
Portlets
Web Application
JISC, Joint Information Service Committees
IMS, The IMS Global Learning Consortium
IEEE, The Institute of Electrical and Electronic Engineers
OSPI, Open Source Portfolio Initiative

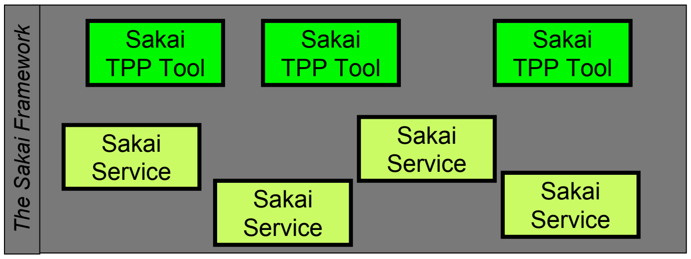
## 1.3 Intended Audience

The Sakai Tool Portability Profile is intended for people interested in understanding how Sakai is implemented and deployed in a Java environment.  It refines the Sakai architecture by providing an example of how that architecture be realized using existing, standard Java technologies.

# 2 The Sakai Java Framework

The Sakai Java Framework is a particular implementation of the Abstract Sakai Environment focused on support for Sakai TPP tools written in Java operating in a web-browser environment. Many other frameworks can be built to support Sakai TPP applications and different design decisions might be made as those frameworks are developed.

The ultimate goals of the Sakai Tool Portability Profile and the Sakai Java Framework is to provide an environment where tools and the services to support those tools can be dropped in as "units of expansion" or "building blocks" as to allow an organization to assemble the componentized units of functionality together to solve their particular application problem.



The Sakai Java Framework can be thought of as a suitable "container" for Sakai TPP tools and associated services.

In addition to supporting Sakai TPP tools, the framework also supports a number of other "real-world" capabilities such as portal integration, non-TPP application integration, and others. Both the Sakai TPP environment and these real-world considerations are described in this document.

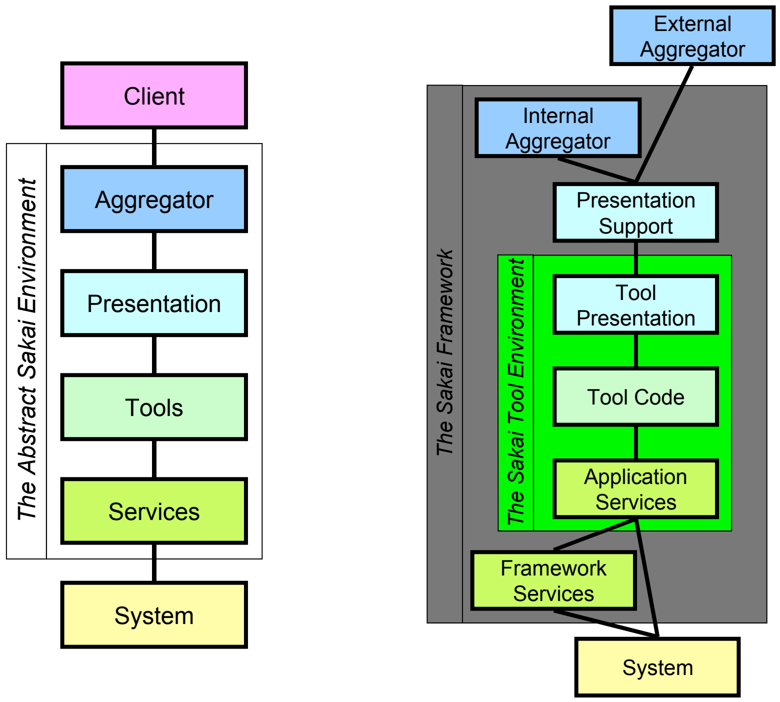
There are a number of different approaches to integrating application functionality into the Sakai Java Framework.

- The Sakai TPP Tools and Services are an approach to building units of extension for Sakai that are well coordinated with the other Sakai Tools. The Sakai TTP environment provides a specific set of JSF widgets to insure consistency between these tools.
- For existing tools written in Java, or tools that must operate both within Sakai and outside of Sakai, there is a simpler form of Servlet integration where an application can access the Sakai APIS without completely converting to be a Sakai TPP tool.
- The framework provides a "wrapper" to allow CHEF tools to be placed into the environment with only minor modifications. The Sakai environment provides services to these tools as a complete replacement to the Jetspeed portal that was used by CHEF.
- While the 1.0 release does not include provisions for integration using web services, into non-java languages such as PHP, the general direction in this area is described.

The rest of this section describes these approaches to integrating application functionality into the Sakai framework.

## 2.1 Extending Sakai Using the Sakai Tool Portability Profile

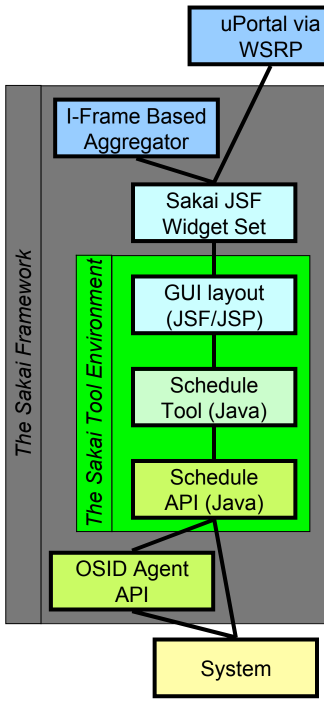
The goal of the Sakai Tool Portability Profile (TPP) is to define the interaction between a Sakai TPP Tool and the Sakai Framework. A Sakai TPP Tool is a well-formed unit of functionality (similar to a plug-in), which can be dropped into a Sakai Framework along with many other Sakai Tools to provide the overall application functionality with a consistent look and feel between tools.  Sakai TPP Tools are by their nature more restrictive than general Java Applications described in the next section - particularly because presentation aspects are constrained in a Sakai TPP tool to use the presentation support provided by the framework.



A key element of the Sakai TPP contract is that it completely specifies all of the interactions that a tool will make including the interfaces to a presentation layer, Application Services, and framework services.  The Sakai TPP forms a complete "perimeter" around a tool so that it truly can be "dropped" into any environment that complies with the Sakai TPP.  It is important to note that the definition of the Sakai TPP is evolving throughout the Sakai project and that we expect it to be quite solid by the Sakai 2.0 release.

Within the Sakai TPP tool environment, tools are broken into three basic layers: (1) Presentation logic, (2) Tool logic, and (3) Application Services.  The framework provides a set of services that can be used either by the tools or Application Services to interact with the framework as necessary.  The presentation of the tool is broken into two layers.  Within the tool, there is an "abstract" expression of what the GUI should look like.    The rendering of the GUI is up to the framework.  This paves the way for support for many presentation elements without the need to change the presentation layout in the tool.

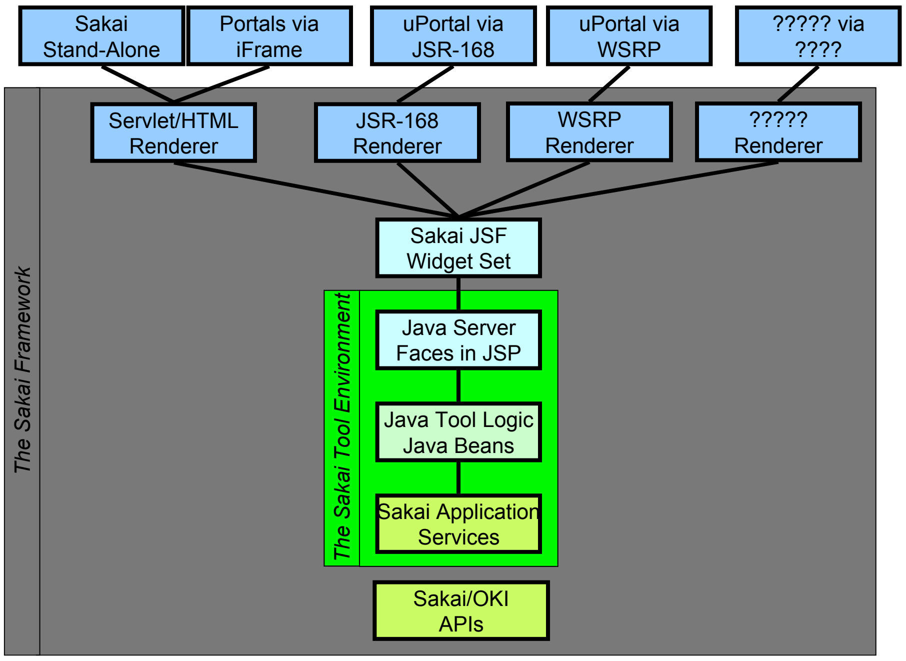
The following identifies some sample technologies for each of the abstract elements of the tool and framework environments in the Sakai Java Framework.



Within the tool environment, tool code is written in Java performing the necessary tool logic, and interacting with the presentation layer using JavaBeans. The presentation is expressed in JavaServer Faces that expresses the presentation elements using the Sakai GUI Components in the Sakai Widget Set.

Each tool will typically have one or more APIs that provide direct support for the tool when interacting with the system and framework APIs. These tool-facing APIs are designed to provide tool writers a convenient and easy-to-use interface. Often these APIs evolve as tool needs and requirements evolve. It is an important design pattern to move as much functionality as possible from the tool logic to the Application API to maximize the opportunity for the reuse of that functionality. It is also important to note that while many of the tool-facing APIs will be used primarily by one tool, there are many examples where the APIs will be used by many tools. A good example of this would be where a grade book tool would be the primary user of the grade book API but an assessment engine that needed to record grades from assessments would also use the grade book API.

As mentioned above, by separating the presentation into two layers where one layer is within the tool and the other layer is in the framework, Sakai tools can be repurposed to support multiple presentation approaches as shown below.



The goal of having the presentation express its GUI in a relatively restricted subset of JavaServer Faces is to allow for a multitude of ultimate presentation capabilities all handled transparently within the framework.

The above diagram can be thought of as a rough roadmap for Sakai presentation efforts. The initial versions of the framework will support the stand-alone aggregation and I-Frame integration within portals. Work is ongoing to evaluate and integrate WSRP as a rendering option and once WSRP has been evaluated, JSR-168 will be worked on as well. Beyond WSRP and JSR-168 there has been talk of a Swing-based renderer or even a Flash based renderer.

The essence of the Sakai TPP is to define how to build and deploy tools. It is naturally a constraining environment that is designed to maximize portability of the Sakai tools. Some may feel that it is too constraining and may choose to develop and integrate applications into Sakai that use mechanisms other than TPP compliance.

## 2.2 Integrating Applications into Sakai

There are many cases where the Sakai TPP is not an appropriate approach to integrating functionality into Sakai including:

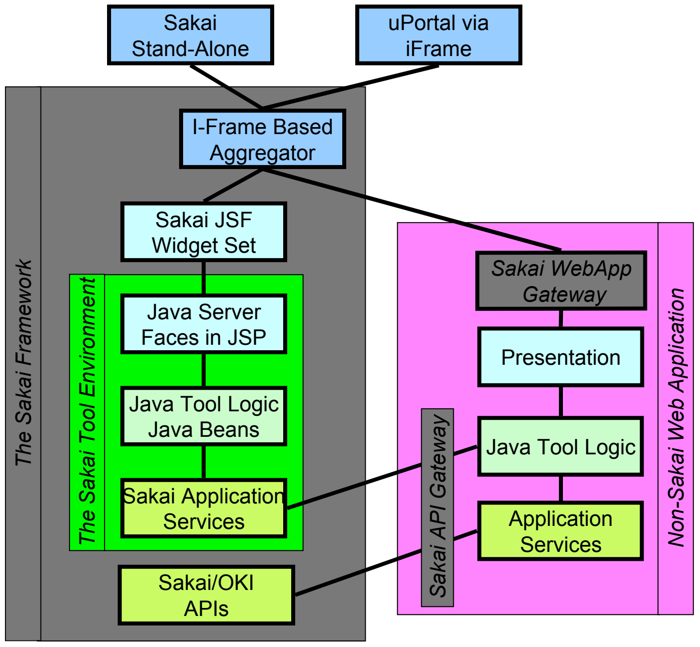
- An application that needs to operate both within Sakai and independent of Sakai
- A large application using presentation technology other than JSF, or is using JSF in ways which are not supported in the Sakai TPP.

Sakai provides a method to integrate these applications into Sakai without requiring the rewrite of the presentation aspects of the tools. Tools that are integrated in this way have full access to the Sakai Framework and Application APIs.

With the proper code structure, it can be quite natural to maintain both a Sakai and stand-alone version of a tool.

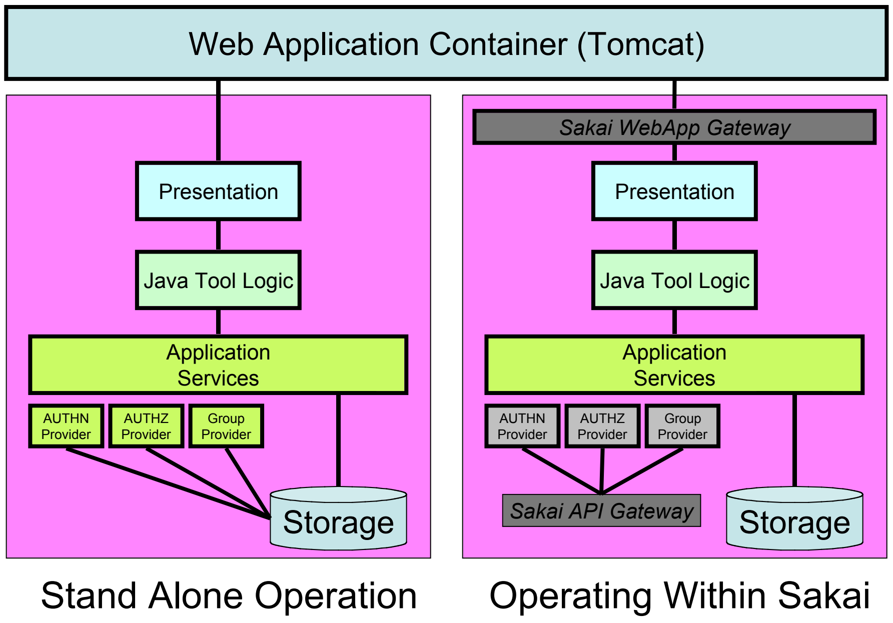
The primary problem which must be solved when bring a Servlet into Sakai is the necessary setup to insure that the Sakai APIs have access to information stored in the Servlet thread. Once the Servlet thread is properly set-up, the Java application can simply make calls to the Sakai APIs using the service location or cover patterns to locate the proper implementations for the Sakai APIs (see later section describing service location).

This necessary initialization has been placed in a Servlet filter that does the necessary thread local work to initialize the Sakai APIs within the web application for each incoming request. Because this is done with a Servlet filter, there is no need to modify the application - the change is to add a **filter** entry to the **web.xml** for the Servlet.



Given that the presentation elements are "within" the tool, the application is completely responsible for its look and feel and any compliance to the Sakai Style Guide, conformance with accessibility rules, and other presentation elements which a Sakai TPP tool delegates to the framework.

Often the application needs to maintain both a "stand-alone" version and a version that works within Sakai. A useful design pattern to solve this problem is for the application to decompose its internal functionality into a set of "providers" which can be plugged into their application services.



When the application is operating as a stand-alone web application, it uses one set of providers for those services that simply stores the appropriate data in the application's local storage. When the application is moved into Sakai it is configured to use a different set of providers for those internal interfaces which make calls to the Sakai services as appropriate.

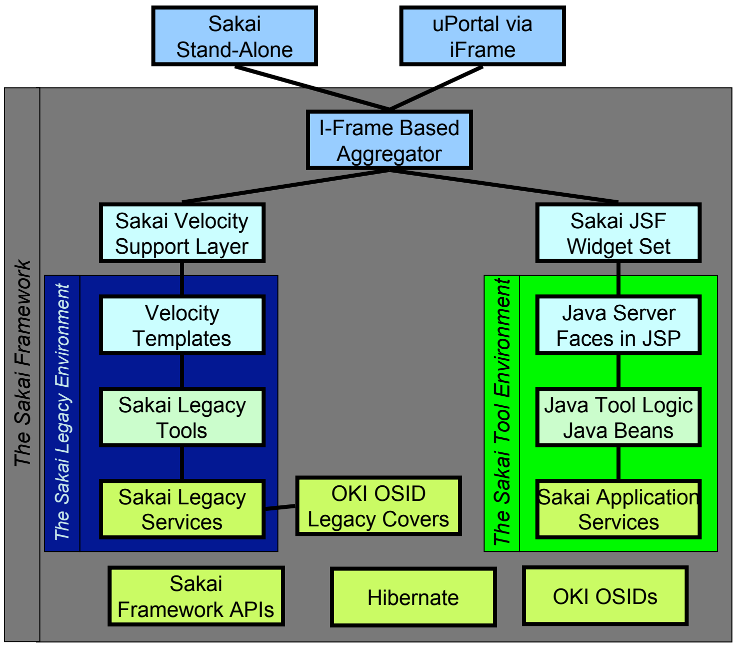
This way any Sakai-specific code is isolated into these providers rather than being sprinkled throughout the application services and tool logic.

## 2.3 Sakai Legacy Tools

The baseline functionality of the Sakai Collaborative and Learning Environment (CLE) in the Sakai 1.0 release is provided by a number of legacy tools that are evolved from the CHEF tools in the 1.12 release of CHEF. These tools were originally written to operate in the Jetspeed portal and used the Velocity template engine. These tools were brought into Sakai by producing a layer that provided basic implementations of needed Velocity and Jetspeed APIs but which interacted with the Sakai framework rather than Jetspeed.

To ease the porting of these tools, a legacy framework and set of services was developed and is supported within Sakai in addition to the TPP compliant tools. These two frameworks operate together in the same environment. The aggregator can display Sakai legacy tools and Sakai TPP tools at the same time.

In addition, the framework and application services are usable across both the legacy and non-legacy environments. This allows an evolutionary approach to tool and service development. The legacy services are quite complete in version 1.0 while the new-generation of Sakai APIs and OKI OSIDs will be developed and used in parallel with the legacy capabilities in Sakai version 2.0 and beyond.



The Sakai legacy environment does not include the Jetspeed portal - the Sakai legacy environment was created by producing a set of proxy implementations for much of the Jetspeed APIs needed by the legacy tools. These Jetspeed APIs are implemented so as to talk to the Sakai Framework.

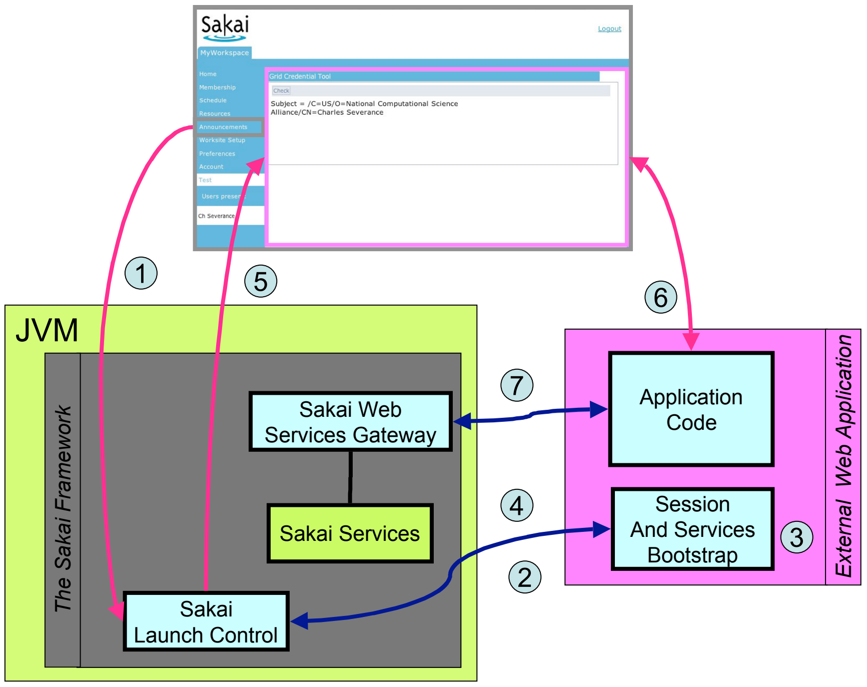
During development of new tools and capabilities there may be a need to use a combination of Sakai legacy framework capabilities and the new Sakai APIs to accomplish the needed tasks. One approach to bridge this gap is to develop a set of OKI OSID interface implementations that cover the legacy APIs - this allows newly developed tools to use the OKI OSIDs to access and interoperate with the legacy services.

## 2.4 Integrating Non-Java Applications into Sakai

While this document is describing the Sakai Java Framework, it is appropriate to discuss how applications which are written in a language such as PHP, Perl, or C# will be integrated into Sakai. Since the Sakai APIs are only available to applications in Java, a web-services gateway must be provided to allow these applications to make use of the Sakai APIs.

In addition there must be a way for the output of non-Java applications to be aggregated together with the output of the Sakai tools. There are two approaches under consideration to solving this problem of aggregating across Java and non-Java applications: (1) I-Frame integration or (2) WSRP integration. In both situations, there will be a need for an analogue to the Sakai Web Application gateway that passes critical set-up information regarding session context using either request parameters or in optional fields in the WSRP information. This information will then be used when making web-services calls back to Sakai to access information such as identity, authorization, or group membership.

An approach that is being discussed is to use a blend of i-Frame and Web-Services to integrate truly external tools that are running on their own web server and running in any language. The following diagram describes the interaction:



There are a number of steps in the process between the point when the user launches the tool and when the tool interaction is complete.

1. First the user selects the tool by pressing a button.
2. The Sakai launch control intercepts the incoming request and contacts the external application via web services. Launch control communicates the user identity, session information, contextual information, and a web-services handle to get back to the Sakai web service gateway.
3. The external web application sets up any session information for the use, records any necessary information about the user and the web services handle.
4. The external application returns a contact URL back to Sakai launch control that sends it to the user's browser in an iFrame.
5. The Sakai launch control launches the application's starting URL in an iFrame
6. The user then interacts with the application within the iFrame using http.
7. When the application needs access to information such as user roles, user information, or other information, Web-Services are used to retrieve this information from Sakai.

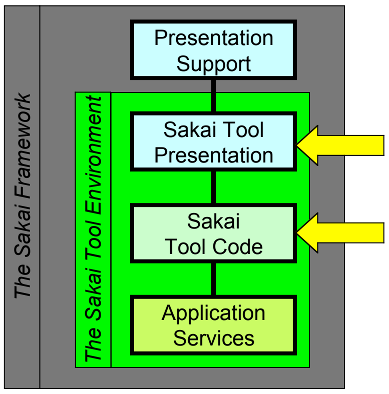
While this shows an outline and approach, there is still a great deal of engineering that must be invested to fully define this approach. We will need to develop specifications for the web-services (WSDL) for the interaction between the launch and the external application, and the interaction between the application and the Sakai web services gateway.

This is only one possible approach to this problem. As requirements are identified and prioritized this and/or other approaches may be designed, evaluated, scheduled and implemented.

# 3 Inside a Sakai TPP Tool

This section includes a very short summary of how the presentation works within a Sakai tool. For more detail you should look at the Sakai Tool Development Guide.

This section describes how the presentation is separated from the tool logic within the Sakai TPP Tool environment.



The GUI layout is described in a restricted variant of JavaServer Faces using a set of Sakai-provided UI Components. The interface is limited to the Sakai provided components to allow future flexibility in rendering for environments beyond HTML such as WSRP, JSR-168, Swing, or even Flash.

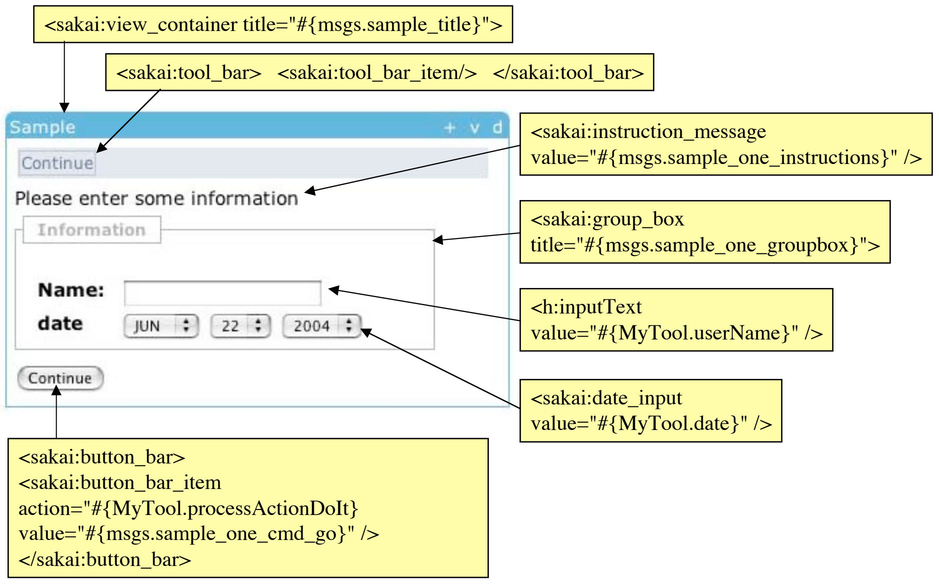
An additional, very important aspect of the Sakai UI Components is to provide for the ability to render the Sakai interface in a number of accessible formats. The Sakai Presentation Support will be informed by the user's accessibility profile and transparently render the Sakai UI components appropriately based on the user's profile. By delegating this to the Sakai Presentation Support we can insure that all Sakai TPP compliant tools are uniformly accessible without placing accessibility oriented code separately into every tool.

The following Sakai GUI Elements are initially defined:

- ButtonBar
- ButtonBarItem
- Comment
- DateInput
- DateOutput
- DocProperties
- DocSection
- DocSectionTitle
- FlatList
- GroupBox
- InstructionMessage
- PanelEdit
- Toolbar
- ToolbarItem
- ToolbarMessage
- ToolbarSpacer
- ViewContainer
- ViewContent

As tools are developed and additional needs are identified the UI Component set will be expanded.

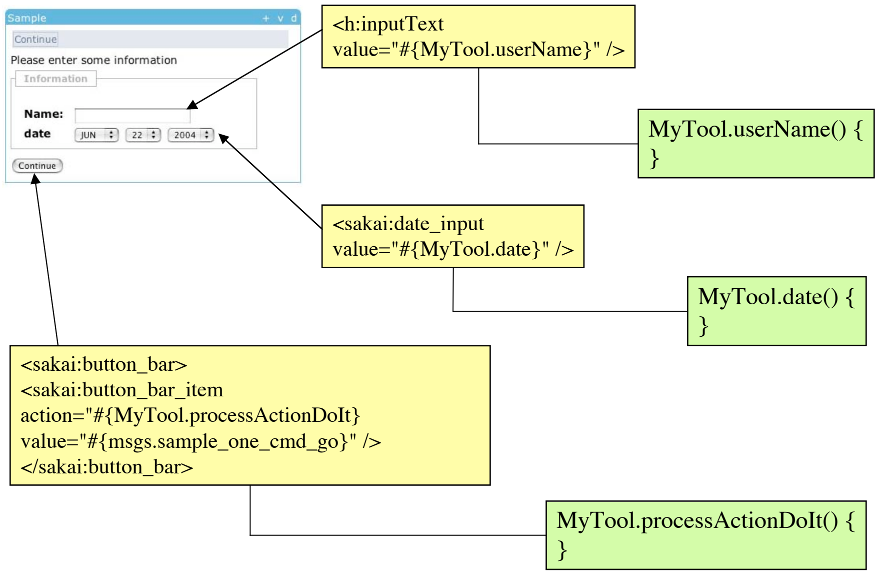
These UI components are assembled together in a JSP description of the interface layout.



```
<sakai:view_container title="#{msgs.sample_title}">
<sakai:tool_bar>   <sakai:tool_bar_item/>   </sakai:tool_bar>
<sakai:instruction_message
value="#{msgs.sample_one_instructions}" />
<sakai:group_box
title="#{msgs.sample_one_groupbox}">
<h:inputText
value="#{MyTool.userName}" />
<sakai:date_input
value="#{MyTool.date}" />
<sakai:button_bar>
<sakai:button_bar_item
action="#{MyTool.processActionDoIt}
value="#{msgs.sample_one_cmd_go}" />
</sakai:button_bar>
```

The values that are used to populate the UI Components come from JavaBeans that are associated with the tool session.

For areas of the user interface where there is some action to be taken, the GUI layout specifies methods to be called in the Tool Java code when a particular action is indicated.  In the example below the **processActionDoIt** is a JavaBean method which is called when the "Continue" button is pressed.



```
<h:inputText
value="#{MyTool.userName}" />
MyTool.userName() {
}
<sakai:date_input
value="#{MyTool.date}" />
MyTool.date() {
}
<sakai:button_bar>
<sakai:button_bar_item
action="#{MyTool.processActionDoIt}
value="#{msgs.sample_one_cmd_go}" />
</sakai:button_bar>
MyTool.processActionDoIt() {
}
```

In the 1.0 release of Sakai, the Sakai Widget set is known to be incomplete and will need to be evolved as part of the later Sakai releases.  The pattern of using the widgets to present GUI elements requires new widgets to be developed to support new capabilities.   While many developers are used to doing this more directly in the tools, taking the approach of extending the widgets insures consistency between tools and insures that accessibility can be addressed within the widgets rather than being spread out into the tools.

Part of the engineering process of Sakai between versions 1.0 and 2.0 is to improve and extend the Sakai GUI widget set.

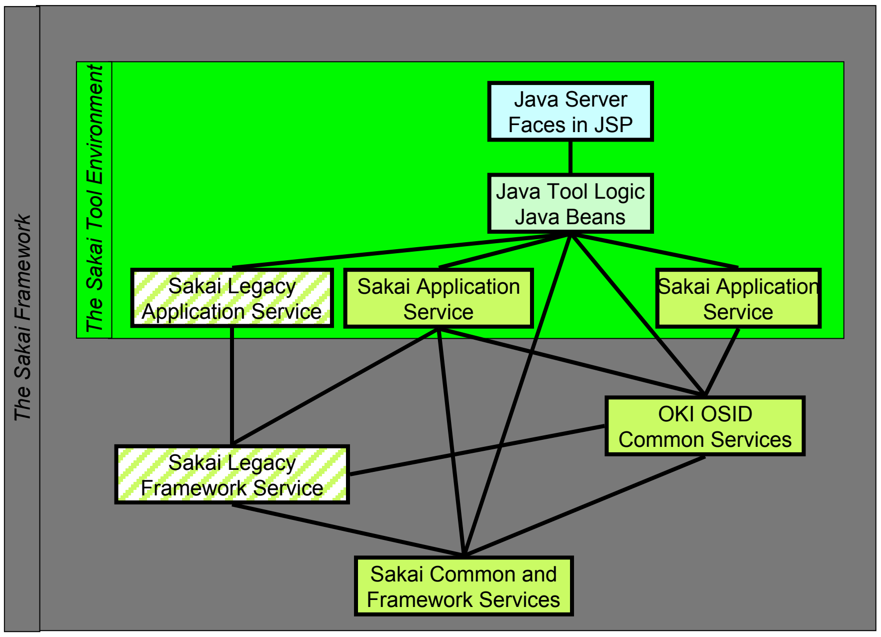
# 4 Understanding the Sakai APIs

Application Programming Interfaces (APIs) and the services that implement those interfaces are a critical part of the Sakai TPP. Often the term "services" is used to describe many different kinds of API implementations. In this section we look at several different categories of APIs that are used within Sakai to help guide you as you look through the Sakai Source code.

The first critical set of APIs are the "Application Services" - these are APIs which often initially created as part of the development of a tool. As an example, the calendar tool will likely use a calendar API heavily and the calendar tool will likely drive the feature set of the calendar API. However it is important to note that other tools may need to use the Application APIs. As an example, an assignment tool may need to access the calendar API when it needs to place an assignment due-date on the calendar.

These Application Services will likely have an interface that is specifically designed for maximal tool convenience. Generally the trend is to move functionality out of tools and into Application Services to maximize the possibility of reuse of the code. As such the application service APIs will generally expand over time as new tool requirements are identified and need support in the application service API.

These Application Services are best thought of as being part of the tool environment and as such, should be written with portability in mind and depend on the framework to help maintain that portability.


The Sakai Framework
The Sakai Tool Environment
Java Server Faces in JSP
Java Tool Logic Java Beans
Sakai Legacy Application Service
Sakai Application Service
Sakai Application Service
OKI OSID Common Services
Sakai Legacy Framework Service
Sakai Common and Framework Services

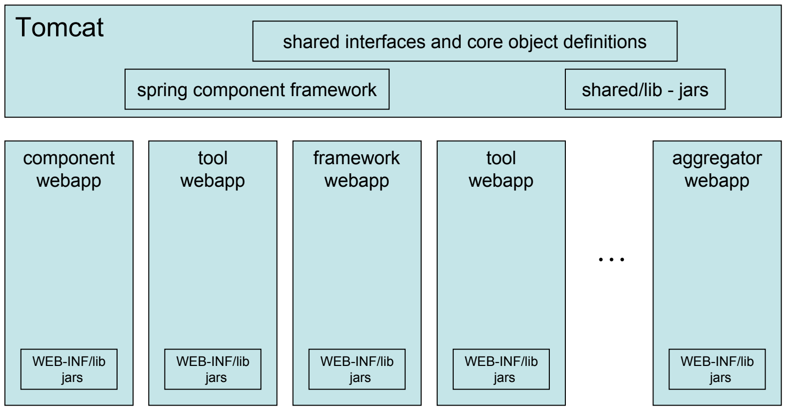
There are different ways to group services within the framework. The Sakai Common and Framework services provide APIs to interact with the entire framework and to provide portable ways to access critical framework information.

There is a set of Sakai Legacy services, both at the framework and application service level. Because much of the end-user functionality in Sakai 1.0 is delivered using legacy tools, the Legacy services must work with the new and evolving OKI OSIDs and the new Sakai Common and Framework services. This compatibility will be maintained using a number of techniques including: (1) implementing a new interface as a cover of an old interface, (2) implementing the old interface as a cover of a new interface, or (3) rewriting legacy tool code to use the newer interfaces.

These approaches will be used to provide a seamless transition for developers from the 1.0 release to the 2.0 release. The legacy capabilities will be maintained as interoperable with the new capabilities well beyond the 2.0 releases.

# 5 Founding Principles of the Sakai Java Framework

The Sakai Java Framework operates in a Tomcat web server environment. However Sakai takes over the entire Tomcat instance and uses the Servlet web application capabilities as a unit of extension within the Sakai framework. This is a departure from the classic web application paradigm.



This is a significant departure from the classic use of a Servlet container and is done for several very important reasons:

- In a JSR-168 portal environment such as Pluto (from Jakarta) each Portlet is also a web application (Servlet) so that Portlets can take advantage of the Servlet API capabilities.

- As multiple development efforts begin to be integrated into a Sakai environment, having a single location for jars becomes an intractable problem. By isolating major units of functionality into separate web applications the needed jars can be kept separate from the other components and tools. Earlier CHEF/Jetspeed experience with integration lead to the conclusion that after about four major applications being integrated, that the WEB-INF/lib was so polluted that it was difficult to keep the system reliable. The biggest problem is that different projects operate at different paces and the task of merging the jars of a two-year-old project with those of a very up-to-date project become nearly impossible and causes a reliability nightmare.

Having tool, implementation, and portal functionality placed into separate web applications allows a far superior model of extension and fits well with the JSR-168 and Portlet needs. However, there are several technical challenges that come from separating the overall application into separate web applications:

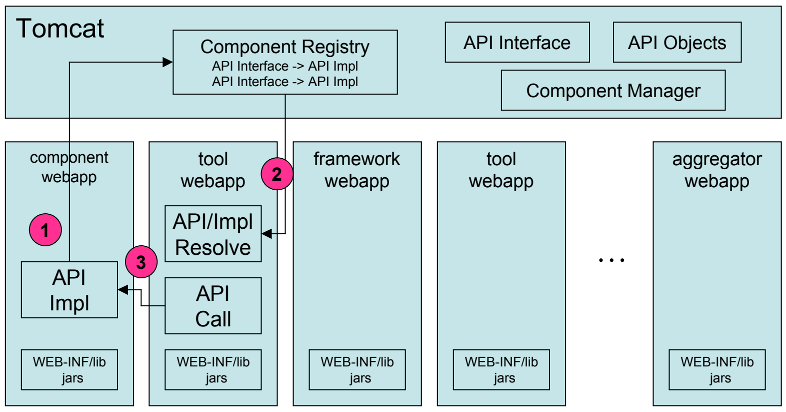
- When tool or service in one web application can call a service implementation where the code is operating in another web application. This requires that the service interfaces and the objects returned by the service APIs are placed in Tomcat's shared/lib area. There must be a way to "find" the implementation using service location or service injection (described below). Sakai has a cross-web-application component manager that is based on Spring that facilitates these cross-web-application lookups.

- When multiple web applications are operating in the normal way, each has a separate Tomcat session. Tomcat tracks these sessions and expires the sessions independently for each web app. In a portal environment where some web applications are not "visible" (i.e. not active), their sessions will expire. There are two solutions to this problem (1) share a session across web applications or (2) have a heartbeat which causes each web applications to "touch" their session to keep it from expiring. Sakai is using the shared session across web applications initially using a tunneling approach pioneered by Jakarta Pluto and is investigating using the heartbeat approach in the future. Sakai is working with Tomcat, Pluto, uPortal, and others to produce a standard mechanism to solve this problem.

The Sakai Java framework solves both of these problems so that the multiple web applications can operate together as an application.

## 5.1 Sakai Component Manager

The Sakai Component manager is a set of software that uses Spring managed-bean capabilities to create and maintain a component registry which maps API interfaces to the actual API implementations. This can be thought of as a "domain name resolver" for API interfaces. This is similar to the OKI OSID loader that provides for the lookup of implementations for a particular interface.



The tasks that must be performed for the Component Manager to operate include the following:

1. When a web application with components (implementations for a particular interface) starts up, a special Servlet in the web application called the ComponentsServlet reads a configuration file and registers the components contained within the Servlet.

2. When the tools (or other consumers of this API) are loaded, the ComponentManager locates the implementation for their needed interfaces. The Component Manager uses the registry to find the pre-registered implementations. There are several methods to perform this lookup including Service Injection and Service Locator (described below).

3. The tool then makes calls to the API implementation across the web application boundary. Because the API interface and objects returned by the API are defined in the shared space, the cross-web-application call works fine. Because the tool and the API implementation operate in different web applications, they can have different jar configurations. An example of this is where a tool needs one version of a utility such as Xerxces while an API needs a different version of the Xerxces utilities.

While deploying application elements across several web applications causes some additional effort, the ability for different elements to have different jar sets is highly valuable and helps to maintain the reliability of the overall system.

In the 1.0 version of Sakai, you can dump out the Component Registry using the Component Servlet. Sample output is shown below:

```
Sakai Component Manager
```

```
Components
org.sakaiproject.service.framework.log.Logger -->
org.sakaiproject.component.framework.log.Jdk14Logger@1dae99
org.sakaiproject.service.legacy.calendar.CalendarService -->
org.sakaiproject.component.legacy.calendar.XmlFileCalendarService@8e54e8
org.sakaiproject.service.legacy.time.TimeService -->
org.sakaiproject.component.legacy.time.BasicTimeService@4e1c6f
org.sakaiproject.service.legacy.assignment.AssignmentService -->
org.sakaiproject.component.legacy.assignment.XmlFileAssignmentService@c5822c
org.sakaiproject.service.framework.portal.PortalService -->
org.sakaiproject.component.framework.portal.BasicPortalService@d8a1bf
org.sakaiproject.service.framework.current.CurrentService -->
…
```

## 5.2 Finding Sakai Service Implementations

There are two basic approaches to locating the implementation for a particular interface: Service Locator and Service Injection. The Sakai Component Manager supports either service injection or service location as a mechanism to resolve references to API implementations. The service injection pattern is preferred as it is more portable.

### 5.2.1 Service Locator Pattern

The "Service Locator" pattern is where the tool makes an explicit call to retrieve the implementation for the interface.

```
import org.sakaiproject.service.framework.component.cover.ComponentManager;

Logger logger = (Logger) ComponentManager.get(Logger);
```

This causes the code to actually depend on a particular component manager due to the explicit import statement required and the method signature.

A common pattern when using the service locator pattern is to "hide" the service locator call using a "Cover Pattern". The "cover" simply has a method for every method in the underlying interface - within each method, the service locator is called, and then the method is called in the implementation returned by the service locator. In Sakai code we use the string "cover" in the package name to indicate when this pattern is in use. This naming convention helps keep static covers separate from interfaces, objects and interface implementations.

### 5.2.2 Service Injection Pattern

A more portable approach is to use a JavaBeans setter method to "inject" the implementing class. This is called the "Service Injection" pattern, because the setter is used to "inject" the actual implementation of the interface after the tool bean is constructed.

This way there is no dependency in the Java on the ComponentManager method signature or Java package name.

```
	public void setLogger(Logger logger) {
		this.logger = logger;
	}
```

In the Sakai Java Framework version 1.0, there are two ways to indicate that service injection is desired. When a service needs to be injected with another service, the service injection is a side-effect of the creation of the service bean. An example **components.xml** file is as follows:

```
<beans>
  <bean id="org.sakaiproject.service.common.id.IdManager"
           class="org.sakaiproject.component.common.id.SakaiIdManager"
            init-method="init"
            destroy-method="destroy"
            singleton="true">
     <property name="logger">
        <ref bean="org.sakaiproject.service.framework.log.Logger"/>
     </property>
  </bean>
</beans>
```

In this example, we are both creating and registering the IdManager bean and at the same time, injecting that bean with the Logger implementation.

For a TPP-Compliant tool, the service injection is described in the **faces-config.xml** file:

```
<faces-config>
   <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: logging service</description>
          <property-name>logger</property-name>
          <value>
             #{Components["org.sakaiproject.service.framework.log.Logger"]}
          </value>
       </managed-property>
  </managed-bean>
</faces-config>
```

Service injection allows the framework to have complete control over the implementations used at run-time by the consuming Java code. In addition, using service injection simplifies the building of unit tests as the unit tests can provide the code with a surrogate or stubbed implementation as part of the test operation.

The Sakai architecture team is investigating other possible mechanisms such as the Java Naming and Directory Interface (JNDI) as a way to revolve API references across web application contexts.

# 6 Aspects of the Sakai Java Framework

## 6.1 Hardware/Software Requirements

The Sakai software is intended to work on a wide range of hardware and operating systems that support Java. This section will describe the typical environments used within the Sakai project for both the production and developer environments.

Sakai developers typically use one of the following environments

Operating System: Windows-XP, Macintosh OS/X, Solaris, or Linux
Processor: PowerPC 800Mhz or higher, Pentium 2Ghz or higher
Memory: 512 MB or higher (1GB recommended)
Software: JAVA 1.4 or later and Jakarta Maven build environment
Optional software: Eclipse

The typical production environment:

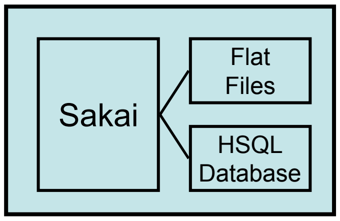
Operating System: Linux or Solaris
Processor: Intel-Based
Memory: >= 4GB
Software: Java 1.4

Users are welcome to use other environments, but by sticking with the common environment used by the Sakai developers and the production Sakai systems, it allows users to take advantage of the testing and QA that is done by the developers and core institutions.

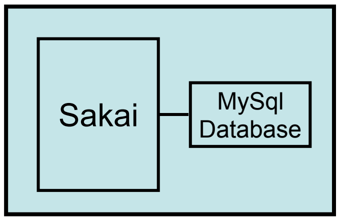
## 6.2 Sakai Target Environments

Sakai is intended to operate in a number of different environments ranging from a developer's desktop through a scalable production environment.

The developer environment is designed to work without a network connection and is configured to store all data either in flat files (for legacy services) or Hypersonic SQL (for newer capabilities). Hypersonic SQL will be included in the release distribution to simplify startup.



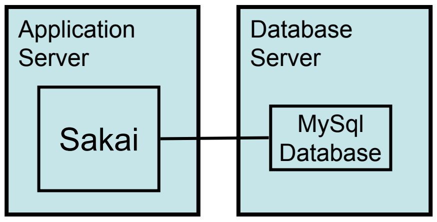
Developer Environment



Small Production Environment

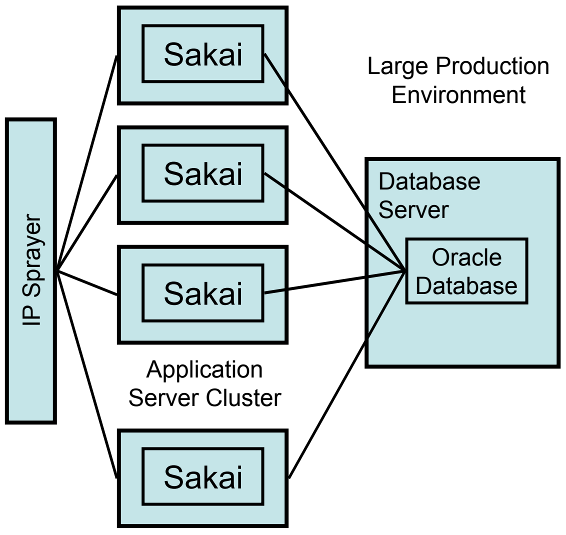
It is a short step to move from the developer environment to a small production environment. The small production environment is reconfigured so as to use a MySql database. This environment is intended for a relatively small population of users (<200) but allows a small server to be set up using a single piece of hardware - perhaps a commodity single-CPU system with 1GB+ of RAM. While backup and security are important issues, it is conceivable that a clever instructor could set up their own Sakai system to support a single class or few research groups.

For a medium sized institution with < 2000 users, a configuration with a single multi-processor application server and separate MySql (or Oracle) database server is the expected configuration. This would likely be run in a professionally managed central computing facility as an enterprise service.



Medium Sized Production Environment

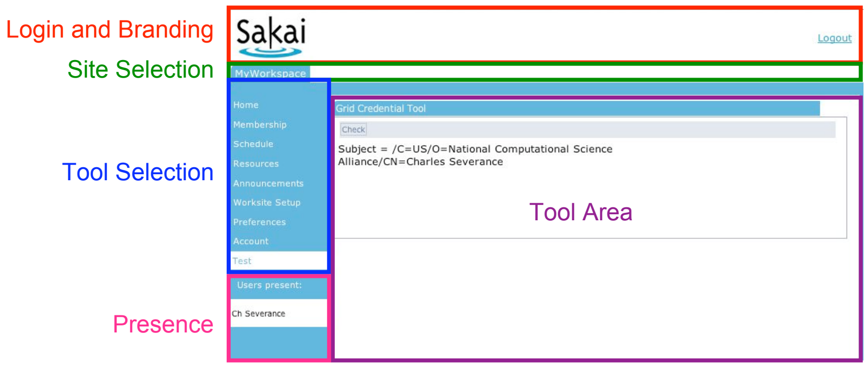
For a large production environment, it becomes necessary to cluster multiple application servers to provide the necessary performance. The clustered version of Sakai depends on the Oracle database. It may be possible to cluster using MySql or other database, but the all cluster testing and deployment is done with an Oracle configuration.



To use the clustered environment, the servers must be accessed through some type of IP sprayer that maintains "sticky" sessions. When a user first associates with a particular application server their incoming requests must always be routed back to the same service so that session information is maintained. Several options exist to solve this problem ranging from dedicated hardware sprayers to specifically configured Apache servers.

## 6.3 Sakai Internal Aggregator

When Sakai is operating as a stand-alone web application, it uses an internal aggregator to render the user view of their pages. The internal aggregator is simply a template that assembles the different elements together for each user request.



Subject = /C=US/O=National Computational Science Alliance/CN=Charles Severance

The internal aggregator template makes use of the following pieces of information:

- Whether or not the user is logged in - this is used to render the login and branding area.
- The list of subscribed sites for the user - this is used to populate the site selection area.
- Which site is currently selected - this is used to highlight the site in the site selection area and determines the tools that are listed in the tool selection area.
- Which tool is selected - highlights the selected and determines what to place in the tool area.
- Presence - indicates which users are in the site at the same time - this is an optional feature which can be enabled, depending on the desires and needs of the site.

The tool area is actually a panel that can contain a number of different tools. In the example shown above, there is a single tool. In the "Home" tool, you will often see a number of tools (perhaps even two columns of tools) displayed in synoptic views.

The template that renders the portal can be edited locally to change look and feel. In successive releases (post 1.0) the rendering for the internal aggregator may change in significant ways. Users who customize this aspect of Sakai should expect that any look-and-feel changes would require some porting effort as new releases come out.

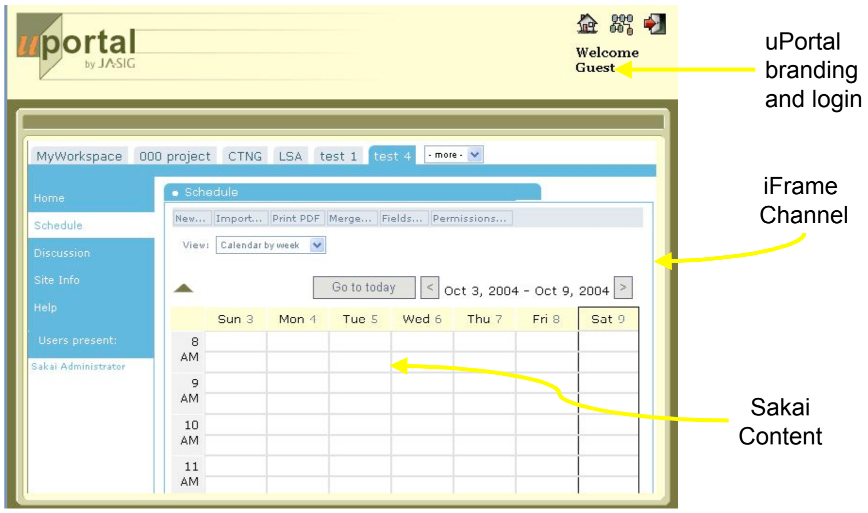
## 6.4 Sakai and uPortal

There are a number of approaches in progress with respect to integrating Sakai into a variety of portals including uPortal. WSRP and JSR-168 are two possible ways for this integration.

For 1.0 Sakai provides a simple mechanism to integrate into a portal:

- The Sakai internal aggregator will produce a reduced portal page that does not include the top frame (branding, login, and logout).
- Both Sakai and the portal will use the same external single-on mechanism such as CAS (http://www.yale.edu/tp/auth/cas20.html)
- The portal administrator adds a reference to the portal using an iFrame channel to Sakai that points at the reduced portal page.

This allows a seamless integration between a portal and Sakai.  The top navigation, branding, log in and log out are all handled by the portal.



In future releases of Sakai, there will be additional capabilities for portal integration that allow a finer grain of access to sites.

The ultimate goal is to use the portal to provide end users the ability to assemble their own federated view across many different Sakai servers.

## 7 Conclusion

The Sakai Java framework provides capabilities to deploy tools and services in a collaborative learning environment. There are a number of different levels of integration between a tool and the Sakai Java Framework. The Sakai Tool Portability Profile provides a Sakai-specific unit of expansion that constrains developers but produces tools with a very uniform look and feel and flexibility in rendering technologies. Sakai also provides a mechanism to integrate tools that already exist without major re-write. In this integration, a tool adds a Servlet filter and can then use the Sakai APIs to access Sakai information such as Agents, Authorizations, or other information.

By allowing multiple approaches developers can choose how to integrate their particular application into Sakai. So far the pattern has been to build new small tools as TPP compliant tools (Sakai Syllabus Tool or Sakai Profile Tool) while larger separately developed applications (The SAMigo assessment engine, Open Source Portfolio, and the Berkeley Gradebook) have chosen to integrate as Java applications.

There is a clear understanding that the TPP capabilities present in the 1.0 release will need to evolve as the tools are developed and new requirements are identified.

The Sakai Java Framework provides both a production environment and developer environment and is a blend of old technologies with new and evolving technologies. Between releases 1.0, 2.0 and 3.0 there will be a shift in emphasis from the legacy capabilities to the newer elements of the framework. The framework has been designed to allow smooth transitions for a wide variety of applications into Sakai.

# 8 List of Contributors

The following individuals contributed to the development of this document:

| | |
|---|---|
| Craig Counterman | Massachusetts Institute of Technology |
| Glenn Golden | University of Michigan |
| Rachel Golub | Stanford University |
| Mark Norton | Sakai |
| Charles Severance | University of Michigan |
| Lance Speelmon | Indiana University |

## 9 Revision History

| Version No. | Release Date | Comments |
|---|---|---|
| 1.0 | October 14, 2004 | Developed from earlier documents for the Sakai 1.0 release. |
| 1.0 | October 24, 2004 | Added diagram to the "Non-Java" section. Typos. |
| | | |