

# Sakai Tools

Glenn R. Golden  
Sakai Framework Architect  
April 6, 2005

*Tools* are units of software that generate a user interface. In our web environment this is usually html to be sent to the browser, or some arbitrary mime-type byte stream that becomes a download through the browser to play in a plug-in or store as a local file.

Tools are used by Sakai's various navigation systems, such as our portal. Tools are also used as *helpers* to other tools, providing content to satisfy the request for user interface for some subset of a client tool's user interface.

Tools are rarely used in the normal model of a web server and a Servlet, where the end user enters a URL that is mapped to the tool's web application context and to the tool's Servlet or JSP or JSF. In this tools are similar to Portlets. Sakai tools are always behind the scenes players for Sakai users who are primarily interacting with a Sakai navigation system such as a portal or hierarchy browser.

A tool is invoked by an internal request dispatcher from navigation or portal code living another web application context, and need not be mapped to the web application URL space at all. But the Tool is mostly unaware of this. The full Servlet API level (Servlet, JSP, JSF) is in effect. The tool still takes requests and produces responses. The details that let a Servlet work as usual but really be a behind-the-scenes Sakai Tool are hidden by Sakai's request processing and tool invocation code.

At a higher abstraction level, from within the world of Sakai tools that are based on JSF, the tool is even less aware of anything unusual going on. From this level, the tool simply has a user interface component tree to establish, populate with information, and respond to changes in. The request / response cycle is even abstracted away from tools at this level.

One thing that is visible to Tools, and therefore to Tool designers, and is also visible to our end users and system administrators is the URL scheme used to navigate to and within tools, and to submit data to tools. Sakai defines a standard for these URLs, and provides support to Tool developers to follow the standard.

Special support for JSF based Tools is provided in Sakai.

## **What can be a tool?**

A tool can be anything covered in the Servlet API; Servlets or JSP or JSF. A tool can even be static content packaged in a web application. A tool could also be a Portlet (but that is not planned for Sakai 2). A tool can be a Sakai enhanced JSF.

Some tools know nothing, as in a static content. Some tools know nothing about Sakai, for example an unmodified Servlet or JSP. Tools can know a little about Sakai, such as when a Servlet uses the Sakai-wide Session and tool placement. Tools can know a great deal about and be dependent on Sakai, as when it is designed as a Sakai enhanced JSF backing bean and component tree.

## Tool Placement

Each tool can be placed into many locations within the Sakai system. Each of these placements acts as a different tool – it has a custom tool configuration, and keeps a separate interaction session with each end user.

Tools can detect their tool placement by looking for the placement id in the request attributes. They will also know about their placement from the placement configuration, also found in the request attributes.

The normal way to package a tool in Sakai is to let it have an `HttpSession` managed by Sakai, and one that is scoped to the tool placement. This means the tool can use the `HttpSession` to store interaction state with each end user, and be assured that Sakai keeps the sessions separate between different tool placements. Tools can also use the `ToolSession` object placed into the request attributes, or ask the `SessionManager` for the current `ToolSession` to get at the same set of information.

Placement id detection is the responsibility of the Sakai kernel (see the request processing document). Placement assignment and placement configuration management is the responsibility of the various Sakai navigation / portal components.

## Tool Context

Each Tool Placement has associated with it a context string. This is a value the tool can use when choosing what set of resources to associated with this placement. For example, the announcement tool uses context to select an announcement channel to use; the resources tool uses context to pick which root folder to show; these are different depending on the tool placement, and depending on the placement context value.

Context is found from the `Tool` interface with `getPlacementContext()`. Context, like placement configuration, is set for the tool by the navigation technology invoking the tool.

While the context is associated with a tool placement, it is not a one-to-one mapping with the tool placement (otherwise we could just use the placement id). In some cases, the context will be unique for each tool placement, and may even match the placement id. But usually there will be multiple tools placed at the same “place” in Sakai which all share the same context.

Traditionally, tool placement context was associated with the *site* in which the tool was embedded. This allowed the chat tool to pick the “main” channel for that site when it was not otherwise configured, the resources tool to show as the “root” folder the site’s

resources area, etc. While Sakai 2 navigation technology might deliver a site id as a tool placement's context, it might also deliver something else, such as a node id in the superstructure. Don't associate any additional meaning to context when using it in a tool.

## Tool Registry

Tools are *registered* automatically when the web application that hosts the tool starts up. This is done by including in the web application Sakai context listener, and one or more tool registration .xml files. These files identify the "well known" tool id used to refer to tools, the tool's default configuration, and other meta-data such as categories and keywords related to the tool.

This registry has two interfaces. One, the `Tool API`, which is not dependent on the Servlet API, is used to discover what tools are available and information about these tools. The other, the `ActiveTool API`, which is dependent on the Servlet API, is used to invoke tools to process requests and produce user interfaces.

## Use of Tools

There are three classes of Tool users in Sakai:

- Tool *aggregators*, which combine the user interface *fragment* produces by one or more tools, possibly with some decoration, to produce a user interface document.
- Tool *dispatchers*, which direct the request to a specific tool, asking it to produce the user interface document.
- Tool helper clients, which delegate portions of their user interface creating responsibility to another *helper* tool.

Aggregators use the request URL to pick a collection of tool placements to invoke and combine into a single display. For each tool placement, the tool is found from the registry, and invoked to produce a fragment of the response. The aggregator combines the responses, possibly adding other elements, to form a single response document. A portal that combines multiple tool placements into a single response acts as an aggregator.

Dispatchers use the request URL to pick a single tool placement to invoke for the entire response. A portal that uses iframes for tool placement display acts as a dispatcher, dispatching to multiple tools for the responses in each iframe. A hierarchy browser that supports drill down into a particular tool placement or data item also acts as a dispatcher. In each case, the dispatcher places some meaning on the request URL to pick a particular tool placement (or set of placements) to invoke.

Helper clients act like dispatchers (or even aggregators). They recognize parts of their request space (certain modes, for instance) that can be handled by some helper tool. The helper tool is found from the registry and invoked to handle the response while in this

mode. Information is shared between the two tools in the tool placement scoped `ToolSession`.

## Tool Manager

The `Tool` API is a kernel API that holds registrations of `Tool` objects and provides tool discovery for navigation and helper-tool clients. Tools are identified by a well-known id. This interface has no dependencies on the Servlet API. It is good for getting information about available tools. Use the `ActiveTool` interface to actually invoke tools.

The Tool API is provided by the `tool` project of the kernel module.

## CurrentTool

The `ToolManager` provides the “current” tool, i.e. the tool involved in the current request processing, i.e. a tool’s meta data, configuration and placement information. A tool can call the `ToolManager.getCurrentTool()` method. This is useful to get at the configuration parameters.

`Tool.getRegisteredConfig()` provides those parameters set in the tool registration. `Tool.getPlacementConfig()` provides those parameters that are specific to this tool placement. This is mutable; the tool may change these Properties, expecting that the changes are permanent for this placement (other aspects of `Tool` are immutable).

`Tool.getConfig()` gives another (read-only) view of the entire set of configuration properties for the tool, combining the registration and placement parameters. In this combination, the placement values override the registration value.

The current `Tool` is also available from the request object in the “`sakai.tool`” attribute (`Tool.TOOL`).

## Active Tools

The `ActiveTool` API is a kernel API that extends the `Tool` API to deliver `ActiveTool` objects (extensions of the `Tool` object) which can be invoked.

The `ActiveTool` API is in the `active-tool` package of the kernel module.

The `ActiveToolManager` is in fact a `ToolManager`. The default Sakai kernel registers the standard `ActiveTool` component as the implementation of both `ToolManager` and `ActiveToolManager`.

Clients of the `ActiveToolManager` get `ActiveTool` instances. `ActiveTools` can be invoked with the `include()`, `forward()` or `help()` methods. The invocation uses the `ActiveTool`’s request dispatcher, and prepares the request object for the invocation. The particular context and Servlet path to show to the `Tool`, and the path information, are set as part of

the tool invocation. This means that even a Servlet without Sakai awareness will likely form proper URLs if they base their URLs on the request information.

## Tool Registration

Tools are packaged in web applications; the packaging includes a tool registration file for Sakai. Web applications that have tools to register must include the `ToolListener`, which finds the tool registration files in the web application and registers tools with the `ToolManager`. This is registered as a Listener in the web application's `web.xml` file.

The `ToolListener` is provided by the `tool-registration` project of the `kernel` module. This is usually loaded into `shared` so it's available to all web applications to use.

The tool registration process enables `ActiveTools`, and grabs each tool's Servlet API `RequestDispatcher` at registration time. We require that the Servlet name in the `web.xml` match the Tool well-known id, so that we can get the request dispatcher by Servlet name (i.e. tool id).

JSF based tools are registered by including a servlet based on the Sakai `JsfTool` for each JSF tool. This Servlet is named with tool's well know id, and is configured for each specific JSF tool being registered.

## Helper Tools

Helper tools are tools that are designed to handle some smaller and common part of user interface that is found in many other tools. Examples include attachment choosers, options editors, and permissions editors.

Tools that want a helper tool to handle a particular request find the helper tool from the `ActiveTool` registry. They place information for the helper into the `ToolSession`, which is shared between the client and the helper. Then they invoke the helper tool with the `ActiveTool`'s `help()` method.

A Tool can map the helper to a part of the tool's URL space. While requests come in to that space, the Tool can invoke the helper to handle the requests.

When a tool invokes a helper, the current tool `PlacementId` and `ToolSession` remain the same, that of the client tool. The client tool and the helper tool use the `ToolSession` to communicate. Before invoking the helper, the client places certain attributes into the `ToolSession`. During and after processing of the helper tool, the client can read other attributes from the `ToolSession` to get information back from the helper. The attributes understood by each helper are determined by each helper and documented with the tool code.

## Special Tool Features

Sakai aware tools can find certain special features in the http request.

## Fragment

If the “`sakai.fragment`” attribute is set in the request (`Tool.FRAGMENT`), the tool should produce a partial document, without the Html `<head>` information, to be aggregated into the already established body of the response. Otherwise the tool should create a complete document.

## ToolSession

The `ToolSession` object can be found in the “`sakai.tool.session`” (`Tool.TOOL_SESSION`) attribute. This is based on the request’s placement id, if present. This is also available from the `SessionManager`’s `getCurrentToolSession()` method.

## PlacementId

The `PlacementId` for the current request can be found in the “`sakai.tool.placement`” (`Tool.PLACEMENT`) attribute of the request. This is the basis of distinguishing one instance of a tool from another, and the basis for selecting the proper `ToolSession` for this request.

## Sakai Session

The Sakai-wide session is made available by the `RequestFilter` in the “`sakai.session`” (`RequestFilter.ATTR_SESSION`) attribute of the request. This is also available from the `SessionManager`’s `getCurrentSession()` method.

## Tool URLs

When an `ActiveTool` is invoked, Sakai sets the context path, Servlet path and path info in the `Request` object that the tool sees. This is set to match the URL scheme of the invoker – that of the hierarchy browser or portal or navigator. It will not match any idea of the tool’s native context id or Servlet path (the tool usually has no Servlet path since it is usually not directly mapped to the URL space of the web application). It is set so that if the tool uses the normal means to produce a URL “back” to the tool, based on information from the request object, it will likely be as the navigation technology wishes it to be.

The path info is of particular interest to the tool. Tools can map their various modes and functions to different URLs within this path part of the overall URL. The tool can use the path info it gets to figure out what to do.

The full URL to a tool’s particular mode or function is a combination of the URL scheme of the navigation technology and that of the tool. For instance, a navigation component might define this URL to get to a particular tool placed for a particular part of the Sakai hierarchy:

```
http://sakaiserver.org/umich/eecs/cs/311/001/chat/main
```

The `sakai.chat` tool might define a URL scheme so that:

```
/presence
```

is interpreted to show the presence list, and

```
/messages
```

is interpreted to show the message list. The full URL to the part of the browser window showing a particular chat's messages would then be:

```
http://sakaiserver.org/umich/eecs/cs/311/001/chat/main/messages
```

The navigation technology would interpret the beginning of the URL. When it sees the “chat” part, it maps that to the `sakai.chat` tool. When it sees the “main” part, it maps it to a particular placement of the chat tool (for this class section's “main” chat).

The navigation code now can invoke the tool with the proper placement id, `ToolSession`, context and Servlet paths, and the path info.

The tool sees the path info:

```
/messages
```

when invoked, and has access to the Tool configuration, Placement configuration, and `ToolSession` data with any interaction state with this particular user at this instance of the tool.

## Integrating Servlet Technology Tools

A Servlet, JSP or JSF system developed outside of Sakai can be used in Sakai as a tool with no or minor modification.

At minimum, we must register the tool, using the tool registration `.xml` file and the `ToolListener` in the `web.xml`. And we must put the `RequestFilter` in front of all invocations of the Servlet, usually by mapping the filter to the Servlet name.

The `web.xml` section that invokes our `ToolListener` looks like this:

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

This comes after all Servlet mapping entries.

The request filter is registered like this:

```
<!-- Sakai request filter -->
<filter>
```

```
<filter-name>sakai.request</filter-name>
<filter-class>
    org.sakaiproject.util.RequestFilter</filter-class>
</filter>
```

And mapped for each tool Servlet defined, like this:

```
<filter-mapping>
    <filter-name>sakai.request</filter-name>
    <servlet-name>sakai.tool.well.known.id</servlet-name>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>INCLUDE</dispatcher>
</filter-mapping>
```

See the request filter document for configuration details.

A minimally integrated tool will be invocable for any non-fragment dispatch, and be tool session aware without knowing it by getting an HttpSession scoped to the tool placement.

The problem likely to be found in this minimal integration is that the tool does not know anything about the context of the request, which becomes important when it selects which data items to work with. Context is usually found from a tool placement's configuration, or the placement's relation to the navigation path as established by the Sakai navigation component that invoked the tool.

URLs generated by the tool will follow the navigation component's scheme extended with the tool's path, as described above, as long as the Servlet technology does not hard code the URLs, but uses the Request object fields (transport, server id, port, context path, Servlet path, path info) to form URL. The code must also properly call the Response object's encodeURL methods(), as called for in the Servlet spec.

Further integration, such as context awareness for data selection, fragment support, invocation of Sakai APIs, etc, can be added to the code when it recognizes it is running in Sakai.

## Integrating JSF Technology Tools

JSF tools have special treatment in Sakai. The preferred method for Sakai tool development is based on JSF, using the Sakai custom UIComponents, JSP Tags and Renders. Sakai also provides integration projects to use Sun's RI and MyFaces.

JSF tools that are not fully Sakai compliant and dependent can be used in Sakai, much like Servlets can, using the same methods outlined in the previous section. But there are some additional integration features needed for JSF.

To meet the needs of tool registration, where we have a Servlet name to associate with each Tool, we need a Servlet registration in the web.xml for each tool. Sakai provides a Servlet to use for each JSF tool, the JsfTool Servlet. This Servlet can be configured to

customize it to each JSF tool view file location specifics. JsfTool also helps in the dispatching of tool requests to JSF tools.

The JsfTool Servlet is in the `tool` project of the `jsf` module.

A JsfTool based tool declaration in the `web.xml` for a JSF tool might look like this:

```
<!-- Sakai request filter, mapped to anything
      that goes to the faces servlet -->
<filter>
  <filter-name>sakai.request</filter-name>
  <filter-class>org.sakaiproject.util.RequestFilter
    </filter-class>
</filter>

<filter-mapping>
  <filter-name>sakai.request</filter-name>
  <servlet-name>Faces Servlet</servlet-name>
  <dispatcher>REQUEST</dispatcher>
  <dispatcher>FORWARD</dispatcher>
  <dispatcher>INCLUDE</dispatcher>
</filter-mapping>

<!-- Sakai JSF Tool Servlet, for some tool -->
<servlet>
  <servlet-name>sakai.well.known.tool.id</servlet-name>
  <servlet-class>org.sakaiproject.jsf.util.JsfTool
    </servlet-class>
  <init-param>
    <param-name>default</param-name>
    <param-value>main</param-value>
  </init-param>
  <init-param>
    <param-name>path</param-name>
    <param-value>/test</param-value>
  </init-param>
  <load-on-startup> 1 </load-on-startup>
</servlet>

<!-- Faces Servlet -->
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  <load-on-startup> 2 </load-on-startup>
</servlet>

<!-- Faces Servlet Mapping -->
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>*.jsf</url-pattern>
</servlet-mapping>
```

The FacesServlet and its mapping are registered in the normal way, using the `*.jsf` mapping form.

In addition to the JsfToolServlet, Sakai provides customization of the stock JSF implementations in the `app` project of the `jsf` module. This collection of *application* plugins, and a `faces-config.xml` to register them, work with our RequestFilter, ActiveTool dispatching, and JsfTool Servlet to enhance various aspects of JSF processing to support Sakai's use of JSF (mostly in URL handling, and redirects after POST that don't lose FacesMessages). To use these plugins, a set of `.jar` files from Sakai need to be bundled, along with the JSF api and implementation files, into each tool hosting webapp. For Sun's RI, the webapp producing project's `project.xml` would look like this:

```
<dependency>
  <groupId>sakaiproject</groupId>
  <artifactId>sakai2-jsf-tool</artifactId>
  <version>sakai.2.0.0</version>
  <properties>
    <war.bundle>>true</war.bundle>
  </properties>
</dependency>

<dependency>
  <groupId>sakaiproject</groupId>
  <artifactId>sakai2-jsf-app</artifactId>
  <version>sakai.2.0.0</version>
  <properties>
    <war.bundle>>true</war.bundle>
  </properties>
</dependency>

<dependency>
  <groupId>sakaiproject</groupId>
  <artifactId>sakai2-util-web</artifactId>
  <version>sakai.2.0.0</version>
  <properties>
    <war.bundle>>true</war.bundle>
  </properties>
</dependency>

<!-- Sun JSF RI jars (jsf-impl, jsf-api) and
dependencies (commons-digester, commons-collections,
commons-digester, commons-beanutils) -->
<dependency>
  <groupId>jsf</groupId>
  <artifactId>jsf-impl</artifactId>
  <version>1.1.01</version>
  <properties>
    <war.bundle>>true</war.bundle>
  </properties>
</dependency>

<dependency>
  <groupId>jsf</groupId>
  <artifactId>jsf-api</artifactId>
  <version>1.1.01</version>
  <properties>
```

```

        <war.bundle>true</war.bundle>
    </properties>
</dependency>

<dependency>
    <groupId>commons-digester</groupId>
    <artifactId>commons-digester</artifactId>
    <version>1.6</version>
    <properties>
        <war.bundle>true</war.bundle>
    </properties>
</dependency>

<dependency>
    <groupId>commons-collections</groupId>
    <artifactId>commons-collections</artifactId>
    <version>3.1</version>
    <properties>
        <war.bundle>true</war.bundle>
    </properties>
</dependency>

<dependency>
    <groupId>commons-beanutils</groupId>
    <artifactId>commons-beanutils</artifactId>
    <version>1.7.0</version>
    <properties>
        <war.bundle>true</war.bundle>
    </properties>
</dependency>

<dependency>
    <groupId>taglibs</groupId>
    <artifactId>standard</artifactId>
    <version>1.0.4</version>
    <properties>
        <war.bundle>true</war.bundle>
    </properties>
</dependency>

<dependency>
    <groupId>jstl</groupId>
    <artifactId>jstl</artifactId>
    <version>1.0.2</version>
    <properties>
        <war.bundle>true</war.bundle>
    </properties>
</dependency>

```

JSF applications can choose to store the View between response and request on the server or in the client HTML. If stored on the server, it will be properly scoped to the tool placement because of Sakai's HttpSession handling. Beans scoped for "session" which need instead to be scoped for tool placement are also properly handled by our scoped HttpSession.

Support for fragment responses is provided, if desired, by using the Sakai View UIComponent (<sakai:view>) to wrap the entire response, instead of hard coding html header elements. This is recommended practice for Sakai tools, and is easily retrofitted to tools developed outside of Sakai.

Support for direct access tool modes can be added using TBD.

## Tool Integration Chart

Various features of Servlet or JSF tool integration are described in this chart on the left. The technologies that make this possible are listed on the right.

<p>Placeable</p> <p>The ability for the tool to exist in multiple places within Sakai, each with its own session and configuration</p>	<ul style="list-style-type: none"> <li>• Dispatch from a Sakai Navigator to get a placement id and configuration.</li> <li>• Include the RequestFilter with the default tool placement detection and tool scoped HttpSession</li> <li>• Access (if needed) the placement id and configuration objects in the Request.</li> <li>• Derive context from placement id or configuration.</li> </ul>
<p>Tool</p> <p>The ability for the tool to be invoked and configured (default configuration)</p>	<ul style="list-style-type: none"> <li>• Include the ToolListener and a tool.xml file for each tool hosted by the webapp.</li> <li>• Include the RequestFilter for session handling, etc.</li> <li>• Dispatch from a Sakai Navigator for default configuration.</li> </ul> <p>For JSF:</p> <ul style="list-style-type: none"> <li>• Include a JsfTool Servlet configuration for each tool hosted in the webapp, named to match the tool.xml registration.</li> <li>• Configure the JstTool with the path and default view for the tool.</li> </ul>
<p>Redirect After Post</p> <p>Part of Sakai standard URL handling, keeping the browser history clean.</p>	<p>For Servlet:</p> <ul style="list-style-type: none"> <li>• Respond to each POST with a redirect to the direct URL to the next tool mode.</li> </ul> <p>For JSF:</p> <ul style="list-style-type: none"> <li>• Use the &lt;redirect /&gt; in the JSF navigation rules.</li> <li>• Bundle the jsf-app project to get Sakai application plugins that help preserve FacesContext Messages between the redirect and the return call.</li> </ul>
<p>Fragment</p> <p>Produce a fragment of HTML that lives in</p>	<p>For Servlet:</p> <ul style="list-style-type: none"> <li>• Recognize the sakai.fragment attribute is set in the Request, and</li> </ul>

<p>a &lt;body&gt; section, without header or footer.</p>	<p>produce the fragment response.  For JSF:</p> <ul style="list-style-type: none"> <li>• Use the Sakai View UIComponent instead of html header and footer code</li> <li>• Bundle the Sakai widgets in the webapp.</li> </ul>
<p>Navigator URLs</p> <p>Produce URLs back to the tool that matches the URL that the navigator recognized as leading to the tool. Part of Sakai standard URL handling.</p>	<p>For Servlet:</p> <ul style="list-style-type: none"> <li>• Dynamically produce “return” URLs based on Request parameters (the Sakai Web util object can be useful).</li> </ul> <p>For JSF:</p> <ul style="list-style-type: none"> <li>• bundle the jsf-app project to get Sakai applicaton plugins to control the URLs generated by Faces.</li> </ul>
<p>Direct URLs</p> <p>Produce URLs back to the tool, and recognize URLs to the tool that include complete information for view (mode) selection as well as “selector” information. Part of Sakai standard URL handling</p>	<p>TBD</p>