CKEditor Integration

CKEditor Support / Generic Editor Binding

Summary

Support for CKEditor as an upgrade to FCKeditor has been desired for some time within the Sakai community. Perhaps most importantly, it addresses aaa a long standing lack of keyboard accessibility. The benefits also include performance, functionality, and ongoing maintenance of the software.

There has been work to provide this support and, at the same time, reduce the effort to integrate further editors or UI toolkits. This document provides a technical overview of the methodology used, status of the work, and implementation advice for those interested in working with other UI technologies or editors.

Methodology Overview

To achieve the goals of adding support for CKEditor and simplifying additional implementations and maintenance, some modifications have been made and some new components have been added. Namely, the portal (`SkinnableCharonPortal`) now renders script tags for the active editor, and the determination of which editor is active is centralized in the `PortalService` and `EditorRegistry`, rather than distributed across UI binding layers.

Status of the Work

The initial integration / refactoring is complete and documented. It is tracked in JIRA as SAK-17880. It is implemented in trunk (2.9) across the various modules and also included within 2.8.x. There are certain to be some bugs found in testing and they should be reported in JIRA and linked to SAK-17880 as related issues. Issues and concerns should be recorded on the Outstanding Issues page during 2.8 QA to have a single point of reference and to avoid duplication of work.

UI Technology Implementation

Support for rich text editors within a UI toolkit has been simplified significantly. The traditional model required that each toolkit determine which editor should be used as well as implement editor-specific bindings for each. The new approach moves this determination to the portal and the binding to a generic JavaScript function call.

Tools can assume that the active editor's script will be loaded in the document head and that the correct launch code will be available as the `sakai.editor.launch()` function. This reduces the implementation requirement for a UI toolkit to the ability to accept the header and footer material from the portal and to render an inline script tag that calls the launch function with the appropriate textarea ID and any appropriate configuration settings. If the UI toolkit should produce the entire document, the URLs and inline script to configure the correct editor for a tool placement are available via utility to be rendered as needed.

There is also a convenience mechanism that allows static script URLs to be embedded, where the request is redirected to the active editor. This could, for example, be used to place a static HTML form somewhere and still have access to the active editor under the `sakai.editor.launch()` function, without being rendered as a traditional tool or even being in the JVM.

It is important to note that the existing UI support within Sakai has been upgraded to use this mechanism. This means that existing tools need only be rebuilt (with updated dependency versions) to take advantage of this. There need be no code changes and all Velocity macros, JSF tags, and so on will continue to function.

Editor Binding Implementation

Each editor is now responsible for supplying its own launch function (rather than being implemented in the UI technology binding). This function will be called consistently as `sakai.editor.launch(textareaId, config)`. There is also a value supplied for supporting file browsing, where `sakai.editor.collectionId` contains the path to the collection in Resources (Content Hosting) for the active site. If the site is unknown, this will be the path to the user's My Workspace Resources.

In addition to providing a script that implements the launch function, the editor must be registered with the system. This is done by implementing the `Editor` Java interface and registering with the `EditorRegistry`. The `Editor` interface has a set of simple methods that return Strings to identify the editor, the paths to the main editor code and launch script, and any inline bootstrapping script that should be included before the editor is loaded. The `EditorRegistry` is available as a bean (`org.sakaiproject.portal.api.EditorRegistry`), so it can be injected or retrieved via the `ComponentManager`.

Modules Affected

Because the usage and implementation of the editor is broad, a number of modules are affected. These changes are:

- jsf – Binding for JSF tools such as Samigo and Message Center (Forums and Messages).
- metaobj – Binding for OSP tools and Forms, including updates to `formCreate.xslt` and `formFieldTemplate.xslt`.

- osp – Updates to JSP templates to support specific OSP tools (Glossary, Matrices, etc.).
- portal – Addition of the `EditorRegistry`, enhancements to the `PortalService` interface and implementation, and exposure of head matter in the default portal, `SkinnableCharonPortal`, so all tool UI technology can access the simpler binding mechanisms.
- reference – Addition of the CKEditor itself and the launch scripts for the editors.
- velocity – Binding for Velocity tools such as Assignments and Site Info.

This work has an umbrella JIRA issue: SAK-17880.

Configuration Options

The new implementation consults the same system-wide option as before, `wysiwyg.editor`. It can be set to `ckeditor` to select CKEditor for the entire instance. The default for the 2.8 release line will remain set to FCKeditor. Site properties are also checked so that a site with a `wysiwyg.editor` property can override the default for early adopters, testing, or special site needs.

Adding new Plugins

See this separate page for more information