Open Journal Systems

Version 2.1

Technical Reference

Revision 3

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/2.0/ca/ or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Table of Contents

Introduction

About the Public Knowledge Project

The Public Knowledge Project (http://pkp.sfu.ca) is dedicated to exploring whether and how new technologies can be used to improve the professional and public value of scholarly research. Bringing together scholars, in a number of fields, as well as research librarians, it is investigating the social, economic, and technical issues entailed in the use of online infrastructure and knowledge management strategies to improve both the scholarly quality and public accessibility and coherence of this body of knowledge in a sustainable and globally accessible form. The project seeks to integrate emerging standards for digital library access and document preservation, such as Open Archives and InterPARES, as well as for such areas as topical maps and doctoral dissertations.

About Open Journal Systems

Open Journal Systems (OJS) is a journal management and publishing system that has been developed by the Public Knowledge Project through its federally funded efforts to expand and improve access to research. OJS assists with every stage of the refereed publishing process, from submissions through to online publication and indexing. Through its management systems, its finely grained indexing of research, and the context it provides for research, OJS seeks to improve both the scholarly and public quality of referred research. OJS is open source software made freely available to journals worldwide for the purpose of making open access publishing a viable option for more journals, as open access can increase a journal's readership as well as its contribution to the public good on a global scale.

Version 2.x represents a complete rebuild and rewrite of Open Journal Systems 1.x, based on two years of working with the editors of the 250 journals using OJS in whole or in part around the world. With the launch of OJS v2.0, the Public Knowledge Project is moving its open source software development (including Open Conference Systems and PKP Harvester) to Simon Fraser University Library, in a partnership that also includes the Canadian Center for Studies in Publishing at SFU.

User documentation for OJS 2.x can be found on the Internet at

http://pkp.sfu.ca/ojs/demo/present/index.php/index/help; a demonstration site is available at http://pkp.sfu.ca/demo/present.

About This Document

Conventions

- Code samples, filenames, URLs, and class names are presented in a `courier` typeface;
- Square braces are used in code samples, filenames, URLs, and class names to indicate a sample value: for example, `[anything]Handler.inc.php` can be interpreted as any file name ending in `Handler.inc.php`
- The URL http://www.mylibrary.com used in many examples is intended as a fictional illustration only.

Technologies

Open Journal Systems 2.x is written in object-oriented PHP (http://www.php.net) using the Smarty template system for user interface abstraction (http://smarty.php.net). Data is stored in a SQL database, with database calls abstracted via the ADODB Database Abstraction library (http://adodb.sourceforge.net).

Recommended server configurations:

- **PHP** support (4.2.x or later)
- **MySQL** (3.23.23 or later) or **PostgreSQL** (7.1 or later)
- **Apache** (1.3.2x or later) or **Apache 2** (2.0.4x or later) or **Microsoft IIS** 6 (PHP 5.x required)
- **Linux**, **BSD**, **Solaris**, **Mac OS X, Windows** operating systems

Other versions or platforms may work but are not supported and may not have been tested. We welcome feedback from users who have successfully run OJS on platforms not listed above.

Design Overview

Conventions

General

- Directories are named using the lowerCamelCase capitalization convention;
- Because OJS 2.x will be translated into multiple languages, no assumptions should be made about word orderings. Any language-specific strings should be defined in the appropriate locale files, making use of variable replacement as necessary.

User Interface

- Layout should be separated from content using Cascading Style Sheets (CSS);
- Smarty templates should be valid XHTML 1.0 Transitional (see http://validator.w3.org/).

PHP Code

- Wherever possible, global variables and functions outside of classes should be avoided;
- Symbolic constants, mapped to integers using the PHP `define` function, are preferred to numeric or string constants;
- Filenames should match class names; for example, the `SectionEditorAction` class is in the file `SectionEditorAction.inc.php`;
- Class names and variables should be capitalized as follows: Class names use CamelCase, and instances use lowerCamelCase. For example, instances of a class `MyClass` could be called `$myClass`;
- Whenever possible and logical, the variable name should match the class name: For example, `$myClass` is preferred to an arbitrary name like `$x`;
- Class names and source code filenames should be descriptive and unique;
- Output should be restricted as much as possible to Smarty templates. A valid situation in which PHP code should output a response is when HTTP headers are necessary;
- To increase performance and decrease server load, `import(...)` calls

should be kept as localized as possible;

- References should be used with care, particularly as they do not behave consistently across different releases of PHP. For increased performance, constructors should be generally called by reference, and references should be used whenever possible when passing objects.

Database

- SQL tables are named in the plural (e.g. `users`, `journals`) and table names are lower case;
- SQL database feature requirements should be kept minimal to promote broad compatibility. For example, since databases handle date arithmetic incompatibly, it is performed in the PHP code rather than at the database level.
- All SQL schema information should be maintained in `dbscripts/xml/ojs_schema.xml` (except plugin schema, described later).

Security

- The validity of user requests is checked both at the User Interface level and in the associated Page class. For example, if a user is not allowed to click on a particular button, it will be disabled in HTML by the Smarty template. If the user attempts to circumvent this and submits the button click anyway, the Page class receiving the form or request will ensure that it is ignored.
- Wherever possible, use the Smarty template engine's string escape features to ensure that HTML exploits and bugs are avoided and special characters are displayed properly. Only the Journal Manager and Site Manager should be able to input unchecked HTML, and only in certain fields (such as the multiline fields in Journal Settings). For example, when displaying a username, always use the following: `{$user->getUsername()|escape}`
- Limited HTML support can be provided using the Smarty `strip_unsafe_html modifier`, e.g. `{$myVariable|strip_unsafe_html}`

Introduction

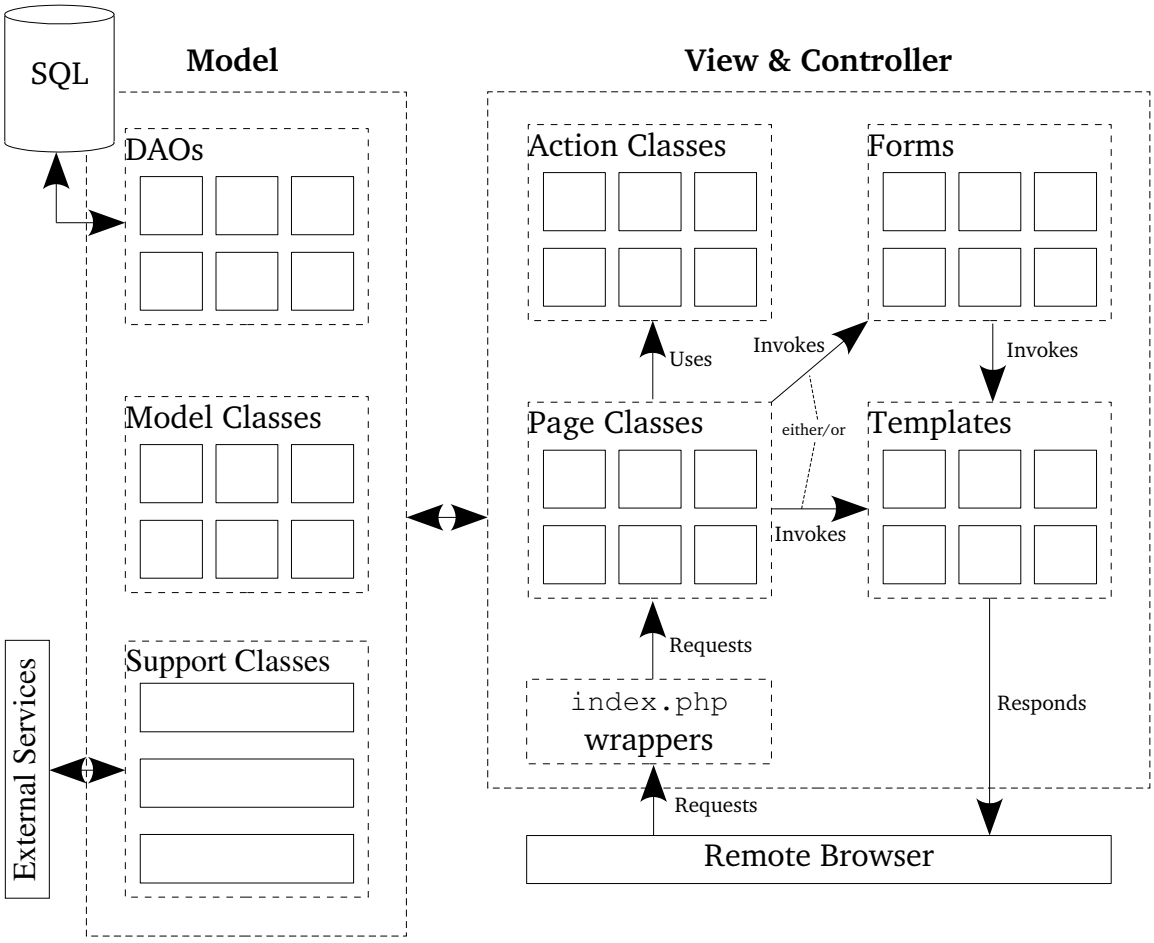
The design of Open Journal Systems 2.x is heavily structured for maintainability, flexibility and robustness. For this reason it may seem complex when first approached. Those familiar with Sun's Enterprise Java Beans technology or the Model-View-Controller (MVC) pattern will note many similarities.

As in a MVC structure, data storage and representation, user interface presentation, and control are separated into different layers. The major categories, roughly ordered from “front-end” to “back-end,” follow:

- **Smarty templates**, which are responsible for assembling HTML pages to display to users;
- **Page classes**, which receive requests from users' web browsers, delegate any required processing to various other classes, and call up the appropriate Smarty template to generate a response;
- **Action classes**, which are used by the Page classes to perform non-trivial processing of user requests;
- **Model classes**, which implement PHP objects representing the system's various entities, such as Users, Articles, and Journals;
- **Data Access Objects** (DAOs), which generally provide (amongst others) update, create, and delete functions for their associated Model classes, are responsible for all database interaction;
- **Support classes**, which provide core functionalities, miscellaneous common classes and functions, etc.

As the system makes use of inheritance and has consistent class naming conventions, it is generally easy to tell what category a particular class falls into. For example, a Data Access Object class always inherits from the `DAO` class, has a class name of the form `[Something]DAO`, and has a filename of the form `[Something]DAO.inc.php`.

The following diagram illustrates the various components and their interactions.



File Structure

The following files are in the root directory of a typical OJS 2.x installation:

File/Directory	*Description*
`cache`	Directory containing cached information
`classes`	Directory containing most of the OJS 2.x PHP code: Model classes, Data Access Objects (DAOs), core classes, etc

File/Directory	*Description*
`config.TEMPLATE.inc.php`	Sample configuration file
`config.inc.php`	System-wide configuration file
`dbscripts`	Directory containing XML database schemas and data such as email templates
`docs`	Directory containing system documentation
`help`	Directory containing system help XML documents
`includes`	Directory containing system bootstrapping PHP code: class loading, miscellaneous global functions
`index.php`	All requests are processed through this PHP script, whose task it is to invoke the appropriate code elsewhere in the system
`js`	Directory containing client-side javascript files
`lib`	Directory containing ADODB (database abstraction) and Smarty (template system) classes
`locale`	Directory containing locale data and caches
`pages`	Directory containing Page classes
`plugins`	Directory containing additional plugins
`public`	Directory containing files to be made available to remote browsers; for example, journal logos are placed here by the system
`registry`	Directory containing various XML data required by the system: scheduled tasks, available locale names, default journal settings, words to avoid when indexing content.
`rt`	Directory containing XML data used by the Reading Tools
`styles`	Directory containing CSS stylesheets used by the system
`templates`	Directory containing all Smarty templates
`tools`	Directory containing tools to help maintain the system: unused locale key finder, scheduled task wrapper, SQL generator, etc.

Request Handling

The way the system handles a request from a remote browser is somewhat confusing if the code is examined directly, because of the use of stub files whose sole purpose is to call on the correct PHP class. For example, although the standard `index.php` file appears in many locations, it almost never performs any actual work on its own.

Instead, work is delegated to the appropriate Page classes, each of which is a subclass of the `Handler` class and resides in the `pages` directory of the source tree.

A Note on URLs

Generally, URLs into OJS make use of the `PATH_INFO` variable. For example, examine the following (fictional) URL:

http://www.mylibrary.com/ojs2/index.php/myjournal/user/profile

The PHP script invoked to handle this request, `index.php`, appears halfway through the URL. The portion of the URL appearing after this is passed to `index.php` via a CGI variable called `PATH_INFO`.

Some server configurations do not properly handle requests like this, which most often results in a 404 error when processing this sort of URL. If the server cannot be re-configured to properly handle these requests, OJS can be configured to use an alternate method of generating URLs. See the `disable_path_info` option in `config.inc.php`. When this method is used, OJS will generate URLs unlike those used as examples in this document. For example, the URL above would appear as:

http://www.mylibrary.com/ojs2/index.php?journal=myjournal&page=user&op=profile

Request Handling Example

Predictably, delegation of request handling occurs based on the request URL. A typical URL for a journal is:

http://www.mylibrary.com/ojs2/index.php/myjournal/user/profile

The following paragraphs describe in a basic fashion how the system handles a request for the above URL. It may be useful to follow the source code at each step for a more comprehensive understanding of the process.

In this example, `http://www.mylibrary.com/ojs2/index.php` is the path to and filename of the root `index.php` file in the source tree. All requests pass through this PHP script, whose task is to ensure that the system is properly configured and to pass control to the appropriate place.

After `index.php`, there are several more components to the URL. The function of the first two (in this case, `myjournal` and `user`) is predefined; if others follow, they serve as parameters to the appropriate handler function.

An Open Journal Systems 2.x installation can host multiple journals; `myjournal` identifies the particular journal this request refers to. There are several situations in which no particular journal is being referred to, such as when a user is viewing the Site Administration pages. In this case, this field takes a value of `index`.

The next field in this example URL identifies the particular Page class that will be used to process this request. In this example, the system would handle a request for the above URL by attempting to load the file `pages/user/index.php`; a brief glance at that file indicates that it simply defines a constant identifying the Page class name (in this case, `UserHandler`) and loads the PHP file defining that class.

The last field, `profile` in this case, now comes into play. It identifies the particular function of the Page class that will be called to handle the request. In the above example, this is the `profile` method of the `User` class (defined in the `pages/user/UserHandler.inc.php` file).

Locating Request Handling Code

Once the framework responsible for dispatching requests is understood, it is fairly easy to locate the code responsible for performing a certain task in order to modify or extend it. The code that delegates control to the appropriate classes has been written with extensibility in mind; that is, it should rarely need modification.

In order to find the code that handles a specific request, follow these steps:

- Find the name of the Page class in the request URL. This is the second field after `index.php`; for example, in the following URL:

  `http://www.mylibrary.com/index.php/myjournal/user/profile`

  the name of the Page class is `UserHandler`. (Page classes always end with `Handler`. Also note the differences in capitalization; in the URL, lowerCamelCase is used; class names are always CamelCase.)
- Find the source code for this Page class in the `pages` directory of the source tree. In the above example, the source code is in `pages/user/UserHandler.inc.php`.
- Determine which function is being called by examining the URL. This is the third field after `index.php`, or, in this case, `profile`.
- Therefore, the handling code for this request is in the file `pages/user/UserHandler.inc.php`, in the function `profile`.

Database Design

The Open Journal Systems 2.x database design is flexible, comprehensive, and consistent; however, owing to the number of features and options the system offers, it is also fairly broad in its scope.

For further information, please see `dbscripts/xml/ojs_schema.xml`.

Table Name	*Primary Key*	*Description*
`access_keys`	`access_key_id`	Stores keys for one-click reviewer access
`article_authors`	`author_id`	Stores article authors on a per-article basis
`article_comments`	`comment_id`	Stores comments between members of the article editing process; note that this is *not* used for reader comments
`article_email_log`	`log_id`	Stores log entries describing emails that have been sent with regard to a specific article
`article_event_log`	`log_id`	Stores log entries describing events that have taken place with regard to a specific article
`article_files`	`file_id, revision`	Stores information regarding the various files (e.g. images, galleys, supplementary files) associated with a particular article
`article_galleys`	`galley_id`	Stores information about a particular layout (or "galley") associated with a particular article
`article_html_galley_images`	`galley_id, file_id`	Associates images with galleys stored in the `article_galleys` table
`article_notes`	`note_id`	Stores notes made for tracking purposes about a particular article by the editor(s)
`article_search_object_keywords`	`object_id, pos`	Provides an index associating keywords, by position, with search objects they appear in
`article_search_`	`object_id`	Lists search "objects", or entities that can be

Table Name	*Primary Key*	*Description*
objects		searched.
article_search_ keyword_list	keyword_id	Stores all keywords appearing in items the system has indexed
article_ supplementary_files	supp_id	Stores information about supplementary files belonging to a particular article
articles	article_id	Stores information on every submission in the system
comments	comment_id	Stores reader comments about articles
copyed_assignments	copyed_id	Stores information about copy editor assignments
currencies	currency_id	Stores information about currencies available to the subscription subsystem
custom_ section_orders	issue_id, section_id	Stores information about issue-specific ordering of journal sections
edit_assignments	edit_id	Stores information on editing assignments
edit_decisions	edit_decision_id	Stores editor decisions with regard to a particular article
email_templates	email_id	Stores a list of email templates that have been modified by the journal manager
email_templates_ data	email_id, locale, journal_id	Stores locale-specific text for emails in email_templates that have been modified by the journal manager
email_templates_ default	email_id	Stores a list of default email templates shipped with this version of OJS 2.x
email_templates_ default_data	email_id, locale, journal_id	Stores locale-specific text for emails in email_templates_default that shipped with this version of OJS 2.x
group_memberships	user_id, group_id	Stores membership information for groups
groups	group_id	Stores information about groups (a.k.a. custom masthead)
issues	issue_id	Stores information about particular issues of

Table Name	*Primary Key*	*Description*
		hosted journals
journal_settings	journal_id, setting_name	Provides a means of storing arbitrary-type settings for each journal
journals	journal_id	Stores a list of hosted journals and a small amount of metadata. (Most metadata is stored in journal_settings)
layouted_assignments	layouted_id	Stores information about layout editor assignments
notification_status	journal_id, user_id	If a user wishes to be notified about a particular journal, they are associated with the journal ID in this table
oai_resumption_ tokens	token	Contains resumption tokens for the OAI protocol interface
plugin_settings	plugin_name, journal_id, setting_name	Stores settings for individual plugins
proof_assignments	proof_id	Stores information about proofreading assignments
published_articles	pub_id	When an article is published, an entry in this table is created to augment information in the articles table
review_assignments	review_id	Stores information about reviewer assignments
review_rounds	article_id, round	Associates an article ID with a review file revision for each round of review
roles	journal_id, role_id, user_id	Defines what roles (manager, editor, reviewer, ...) users have within particular journals
rt_contexts	context_id	Reading Tools contexts
rt_searches	search_id	Reading Tools searches
rt_settings	journal_id	Reading Tools settings for each journal
rt_versions	version_id	Reading Tool versions

Table Name	*Primary Key*	*Description*
scheduled_tasks	class_name	On systems supporting scheduled tasks, this table is used by the task execution script to store information about when tasks were last performed
section_editors	journal_id, section_id, user_id	Associates section editors with sections of journals that they edit
sections	section_id	Defines sections within which journals can publish articles
sessions	session_id	Stores session information for the users who are currently using the system
site	title	Stores site-wide configuration information
subscription_types	type_id	Defines types of subscriptions made available by the subscription subsystem
subscriptions	subscription_id	Describes subscriptions “owned” by the system's users
temporary_files	file_id	Used for situations in which a file must be temporarily stored on the server between user requests
users	user_id	Stores information about every user registered with the system
versions	major, minor, revision, build	Stores information about the current deployment of OJS 2.x

Class Reference

Class Hierarchy

All classes and subclasses of the major OJS 2.x objects are listed below. Indentation indicates inheritance; for example, `AuthorAction` inherits from `Action`.

```
AccessKeyManager
Action
        AuthorAction
        CopyeditorAction
        LayoutEditorAction
        ProofreaderAction
        ReviewerAction
        SectionEditorAction
                EditorAction
ArticleLog
ArticleSearch
ArticleSearchIndex
CacheManager
CommandLineTool
        dbXMLtoSQL
        importExport
        installTool
        migrate
        rebuildSearchIndex
        runScheduledTasks
        upgradeTool
Config
ConfigParser
Core
DAO
        AccessKeyDAO
        ArticleCommentDAO
        ArticleDAO
        ArticleEmailLogDAO
        ArticleEventLogDAO
        ArticleFileDAO
        ArticleGalleyDAO
        ArticleNoteDAO
        ArticleSearchDAO
        AuthSourceDAO
        AuthorDAO
        AuthorSubmissionDAO
        CommentDAO
        CopyAssignmentDAO
        CopyeditorSubmissionDAO
        CountryDAO
        CurrencyDAO
        EditAssignmentDAO
        EditorSubmissionDAO
        EmailTemplateDAO
        GroupDAO
        GroupMembershipDAO
```

```
        IssueDAO
        JournalDAO
        JournalSettingsDAO
        JournalStatisticsDAO
        LayoutAssignmentDAO
        LayoutEditorSubmissionDAO
        NotificationStatusDAO
        OAIDAO
        PluginSettingsDAO
        ProofAssignmentDAO
        ProofreaderSubmissionDAO
        PublishedArticleDAO
        RTDAO
        ReviewAssignmentDAO
        ReviewerSubmissionDAO
        RoleDAO
        ScheduledTaskDAO
        SectionDAO
        SectionEditorSubmissionDAO
        SectionEditorsDAO
        SessionDAO
        SiteDAO
        SubscriptionDAO
        SubscriptionTypeDAO
        SuppFileDAO
        TemporaryFileDAO
        UserDAO
        VersionDAO
DAORegistry
DBConnection
DBDataXMLParser
DBResultRange
DataObject
        AccessKey
        Article
                AuthorSubmission
                CopyeditorSubmission
                LayoutEditorSubmission
                ProofreaderSubmission
                PublishedArticle
                ReviewerSubmission
                SectionEditorSubmission
                        EditorSubmission
        ArticleComment
        ArticleEmailLogEntry
        ArticleEventLogEntry
        ArticleFile
                ArticleGalley
                        ArticleHTMLGalley
                ArticleNote
                SuppFile
        AuthSource
        Author
        BaseEmailTemplate
                EmailTemplate
                LocaleEmailTemplate
        Comment
        CopyAssignment
        Currency
        EditAssignment
        Group
        GroupMembership
        HelpToc
        HelpTopic
```

```
        HelpTopicSection
        Issue
        Journal
        LayoutAssignment
        Mail
                MailTemplate
                        ArticleMailTemplate
        ProofAssignment
        ReviewAssignment
        Role
        Section
        Site
        Subscription
        SubscriptionType
        TemporaryFile
        User
                ImportedUser
        Version
EruditExportDom
FileManager
        ArticleFileManager
        PublicFileManager
        TemporaryFileManager
FileWrapper
        FTPFileWrapper
        HTTPFileWrapper
Form
        ArticleGalleyForm
        AuthSourceSettingsForm
        AuthorSubmitForm
                AuthorSubmitStep1Form
                AuthorSubmitStep2Form
                AuthorSubmitStep3Form
                AuthorSubmitStep4Form
                AuthorSubmitStep5Form
        AuthorSubmitSuppFileForm
        ChangePasswordForm
        CommentForm
                CopyeditCommentForm
                EditorDecisionCommentForm
                LayoutCommentForm
                PeerReviewCommentForm
                ProofreadCommentForm
        CommentForm
                CopyeditCommentForm
                EditorDecisionCommentForm
                LayoutCommentForm
                PeerReviewCommentForm
                ProofreadCommentForm
        ContextForm
        CreateReviewerForm
        EditCommentForm
        EmailTemplateForm
        GoogleScholarSettingsForm
        GroupForm
        ImportOJS1Form
        InstallForm
        IssueForm
        JournalSetupForm
                JournalSetupStep1Form
                JournalSetupStep2Form
                JournalSetupStep3Form
                JournalSetupStep4Form
                JournalSetupStep5Form
```

```
        JournalSiteSettingsForm
        LanguageSettingsForm
        MetadataForm
        ProfileForm
        RegistrationForm
        SearchForm
        SectionForm
        SiteSettingsForm
        SubscriptionForm
        SubscriptionTypeForm
        SuppFileForm
        UpgradeForm
        UserManagementForm
        VersionForm
FormError
FormValidator
        FormValidatorArray
        FormValidatorCustom
        FormValidatorInSet
        FormValidatorLength
        FormValidatorRegExp
                FormValidatorAlphaNum
                FormValidatorEmail
GenericCache
        FileCache
        MemcacheCache
Handler
        AboutHandler
        AdminHandler
                AdminFunctionsHandler
                AdminJournalHandler
                AdminLanguagesHandler
                AdminSettingsHandler
                AuthSourcesHandler
        ArticleHandler
                RTHandler
        AuthorHandler
                SubmissionCommentsHandler
                SubmitHandler
                TrackSubmissionHandler
        CommentHandler
        CopyeditorHandler
                SubmissionCommentsHandler
                SubmissionCopyeditHandler
        GatewayHandler
        HelpHandler
        IndexHandler
        InformationHandler
        InstallHandler
        IssueHandler
        LayoutEditorHandler
                SubmissionCommentsHandler
                SubmissionLayoutHandler
        LoginHandler
        ManagerHandler
                EmailHandler
                FilesHandler
                GroupHandler
                ImportExportHandler
                JournalLanguagesHandler
                PeopleHandler
                PluginHandler
                SectionHandler
                SetupHandler
```

```
                StatisticsHandler
                SubscriptionHandler
        OAIHandler
        ProofreaderHandler
                SubmissionCommentsHandler
                SubmissionProofreadHandler
        RTAdminHandler
                RTContextHandler
                RTSearchHandler
                RTSetupHandler
                RTVersionHandler
        ReviewerHandler
                SubmissionCommentsHandler
                SubmissionReviewHandler
        SearchHandler
        SectionEditorHandler
                EditorHandler
                        IssueManagementHandler
                SubmissionCommentsHandler
                SubmissionEditHandler
        SubscriptionManagerHandler
        UserHandler
                EmailHandler
                ProfileHandler
                RegistrationHandler
Help
HookRegistry
ImportOJS1
Installer
        Install
        Upgrade
IssueAction
ItemIterator
        ArrayItemIterator
        DAOResultFactory
        DBRowIterator
                JournalReportIterator
        VirtualArrayIterator
Locale
NativeExportDom
NativeImportDom
OAI
        JournalOAI
OAIConfig
OAIIdentifier
        OAIRecord
OAIMetadataFormat
        OAIMetadataFormat_DC
        OAIMetadataFormat_MARC
        OAIMetadataFormat_MARC21
        OAIMetadataFormat_RFC1807
OAIRepository
OAIResumptionToken
OAISet
Plugin
        AuthPlugin
                LDAPAuthPlugin
        GatewayPlugin
                GoogleScholarPlugin
        GenericPlugin
        ImportExportPlugin
                EruditExportPlugin
                NativeImportExportPlugin
                SampleImportExportPlugin
```

```
                UserImportExportPlugin
PluginRegistry
RT
        JournalRT
RTAdmin
        JournalRTAdmin
RTContext
RTSearch
RTVersion
RTXMLParser
Registry
Request
SMTPMailer
SQLParser
ScheduledTask
        ReviewReminder
SearchFileParser
        SearchHTMLParser
        SearchHelperParser
SessionManager
String
TemplateManager
Transcoder
UserExportDom
UserXMLParser
Validation
VersionCheck
XMLCustomWriter
XMLDAO
        HelpTocDAO
        HelpTopicDAO
XMLNode
XMLParser
XMLParserHandler
        XMLParserDOMHandler
```

Page Classes

Introduction

Pages classes receive requests from users' web browsers, delegate any required processing to various other classes, and call up the appropriate Smarty template to generate a response (if necessary). All page classes are located in the `pages` directory, and each of them must extend the `Handler` class (`see classes/core/Handler.inc.php`).

Additionally, page classes are responsible for ensuring that user requests are valid and any authentication requirements are met. As much as possible, user-submitted form parameters and URL parameters should be handled in Page classes and not elsewhere, unless a Form class is being used to handle parameters.

An easy way to become acquainted with the tasks a Page class must fulfill is to

examine a typical one. The file `pages/about/AboutHandler.inc.php` contains the code implementing the class `AboutHandler`, which handles requests such as http://www.mylibrary.com/ojs2/myjournal/about/siteMap. This is a fairly simple Page class responsible for fetching and displaying various metadata about the journal and site being viewed.

Each Page class implements a number of functions that can be called by the user by addressing the appropriate Page class and function in the request URL. (See the section titled "Request Handling" for more information on the mapping between URLs and page classes.)

Often, Page classes handle requests based on the role the user is playing. For example, there is a Page class called `AuthorHandler` (in the directory `pages/author/AuthorHandler.inc.php`) that delegates processing of the various tasks an author might perform. Similarly, there are classes called `LayoutEditorHandler`, `ManagerHandler`, and so forth.

The number of tasks a Page handler must perform can frequently be considerable. For example, if all requests for Section Editor functions were handled directly by the `SectionEditorHandler` class, it would be extremely large and difficult to maintain. Instead, functions are further subdivided into several other classes (such as `SubmissionEditHandler` and `SubmissionCommentsHandler`), with `SectionEditorHandler` itself remaining just to invoke the specific subclass.

Action Classes

Action Classes are used by the Page classes to perform non-trivial processing of user requests. For example, the `SectionEditorAction` class is invoked by the `SectionEditorHandler` class or its subclasses (see Page Classes) to perform as much of the work as can be offloaded easily. This leaves the Page class to do its job – validation of user requests, authentication, and template setup – and keeps the actual processing separate.

The Action classes can be found in `classes/submission/[actionName]/[ActionName]Action.inc.php`; for example, the Section Editor action class is `classes/submission/sectionEditor/SectionEditorAction.inc.php`.

The most common sorts of tasks an Action class will perform are sending emails, modifying database records (via the Model and DAO classes), and handling uploaded files (once again via the appropriate classes). Returning to the Model/View/Controller (MVC) architecture, Action classes perform the more interface-agnostic functions of the Controller component.

Each of the more complex roles, such as Author, Section Editor, and Proofreader, has its own Action class. Another way to consider the function of an Action class is to look at it from a role-based perspective, ignoring the user interface: any major processing that an Author should be able to perform should be implemented in the `AuthorAction` class. The user interface then calls these functions as necessary.

Model Classes

The Model classes are PHP classes responsible only for representing database entities in memory. For example, the `articles` table stores article information in the database; there is a corresponding Model class called `Article` (see `classes/article/Article.inc.php`) and DAO class called `ArticleDAO` (see the section called Data Access Objects [DAOs]).

Methods provided by Model classes are almost exclusively get/set methods to retrieve and store information, such as the `getTitle()` and `setTitle($title)` methods of the `Article` class. Model classes are not responsible for database storage or updates; this is accomplished by the associated DAO class.

All Model classes extend the `DataObject` class.

Data Access Objects (DAOs)

Data Access Objects are used to retrieve data from the database in the form of Model classes, to update the database given a modified Model class, or to delete rows from the database.

Each Model class has an associated Data Access Object. For example, the `Article`

class (`classes/article/Article.inc.php`) has an associated DAO called `ArticleDAO` (`classes/article/ArticleDAO.inc.php`) that is responsible for implementing interactions between the Model class and its database entries.

All DAOs extend the `DAO` class (`classes/db/DAO.inc.php`). All communication between PHP and the database back-end is implemented in DAO classes. As much as is logical and efficient, a given DAO should limit its interaction to the table or tables with which it is primarily concerned.

DAOs, when used, are never instantiated directly. Instead, they are retrieved by name using the `DAORegistry` class, which maintains instances of the system's DAOs. For example, to retrieve an article DAO:

```
$articleDao = &DAORegistry::getDAO('ArticleDAO');
```

Then, to use it to retrieve an article with the ID stored in `$articleId`:

```
$article = &$articleDao->getArticle($articleId);
```

Note that many of the DAO methods that fetch a set of results will return subclasses of the `ItemIterator` class rather than the usual PHP array. This facilitates paging of lists containing many items, and can be more efficient than preloading all results into an array. See the discussion of Paging Classes in the Support Classes section.

Support Classes

Sending Email Messages

```
classes/mail/Mail.inc.php
classes/mail/MailTemplate.inc.php
classes/mail/ArticleMailTemplate.inc.php
```

These classes, along with the `EmailTemplate` and `MailTemplate` model classes and `EmailTemplateDAO` DAO class, provide all email functionality used in the system.

`Mail.inc.php` provides the basic functionality for composing, addressing, and sending an email message. It is extended by the class `MailTemplate` to add

support for template-based messages. In turn, `ArticleMailTemplate` adds features that are useful for messages pertaining to a specific article, such as message logging that can be viewed on a per-article basis.

For a sample of typical usage and invocation code, see the various Action classes, such as `SectionEditorAction`'s `notifyReviewer` method. Note that since nearly all emails composed by the system must be displayed to the user, who then must be able to modify it over several browser request-response cycles, some complexity is necessary to maintain the system's state between requests.

Internationalization

System internationalization is a critical feature for OJS 2.x; it has been designed without making assumptions about the language it will be presented in.

There is a primary XML document for each language of display, located in the `locale` directory in a subdirectory named after the locale; for example, the `en_US` locale information is located in the `locale/en_US/locale.xml` file.

This file contains a number of locale strings used by the User Interface (nearly all directly from the Smarty templates, although some strings are coded in the Page classes, for example).

These are invoked by Smarty templates with the `{translate key="[keyName]"}` directive (see the section titled User Interface for more information). Variable replacement is supported.

The system's locales are configured, installed and managed on the Languages page, available from Site Settings. The available locales list is assembled from the registry file `registry/locales.xml`.

In addition to the language-dependent `locale.xml` file, locale-specific data can be found in subdirectories of the `dbscripts/xml/data/locale` and `registry/locale` directories, once again named after the locale. For example, the XML file `dbscripts/xml/data/locale/en_US/email_templates_data.xml` contains all email template text for the `en_US` (United States English) locale.

All XML data uses UTF-8 encoding and as long as the back-end database is configured to properly handle special characters, they will be stored and displayed as entered.

OJS 2.x has limited support for simultaneous multiple locales for a single journal. For example, articles have a primary locale; however, titles and abstracts can have up to two additional locales.

Internationalization functions are provided by `classes/i18n/Locale.inc.php`. See also `classes/template/TemplateManager.inc.php` (part of the User Interface's support classes) for the implementation of template-based locale translation functions.

Forms

The `Forms` class (`classes/form/Form.inc.php`) and its various subclasses, such as `classes/manager/form/SectionForm.inc.php`, which is used by a Journal Manager to modify a Section, centralize the implementation of common tasks related to form processing such as validation and error handling.

Subclasses of the `Form` class override the constructor, `initData`, `display`, `readInputData`, and `execute` methods to define the specific form being implemented. The role of each function is described below:

- Class constructor: Initialize any variables specific to this form. This is useful, for example, if a form is related to a specific Article; an `Article` object or article ID can be required as a parameter to the constructor and kept as a member variable.
- `initData`: Before the form is displayed, current or default values (if any) must be loaded into the `_data` array (a member variable) so the form class can display them.
- `display`: Just before a form is displayed, it may be useful to assign additional parameters to the form's Smarty template in order to display additional information. This method is overridden in order to perform such assignments.
- `readInputData`: This method is overridden to instruct the parent class which form parameters must be used by this form. Additionally, tasks like

validation can be performed here.
- `execute`: This method is called when a form's data is to be "committed." This method is responsible, for example, for updating an existing database record or inserting a new one(via the appropriate Model and DAO classes).

The best way to gain understanding of the various Form classes is to view a typical example such as the `SectionForm` class from the example above (implemented in `classes/manager/form/SectionForm.inc.php`). For a more complex set of examples, see the various Journal Manager's Setup forms (in the `classes/manager/form/setup` directory).

It is not convenient or logical for all form interaction between the browser and the system to be performed using the `Form` class and its subclasses; generally speaking, this approach is only useful when a page closely corresponds to a database record. For example, the page defined by the `SectionForm` class closely corresponds to the layout of the `sections` database table.

Configuration

Most of OJS 2.x's settings are stored in the database, particularly journal settings in the `journal_settings` table, and are accessed via the appropriate DAOs and Model classes. However, certain system-wide settings are stored in a flat file called `config.inc.php` (which is not actually a PHP script, but is so named to ensure that it is not exposed to remote browsers).

This configuration file is parsed by the `ConfigParser` class (`classes/config/ConfigParser.inc.php`) and stored in an instance of the `Config` class (`classes/config/Config.inc.php`).

Core Classes

The Core classes (in the `classes/core` directory) provide fundamentally important functions and several of the classes upon which much of the functionality of OJS 2.x is based. They are simple in and of themselves, with flexibility being provided through their extension.

- `Core.inc.php`: Provides miscellaneous system-wide functions
- `DataObject.inc.php`: All Model classes extend this class
- `Handler.inc.php`: All Page classes extend this class
- `Registry.inc.php`: Provides a system-wide facility for global values, such as system startup time, to be stored and retrieved
- `Request.inc.php`: Provides a wrapper around HTTP requests, and provides related commonly-used functions
- `String.inc.php`: Provides locale-independent string-manipulation functions and related commonly-used functions

In particular, the Request class (defined in `classes/core/Request.inc.php`) contains a number of functions to obtain information about the remote user and build responses. All URLs generated by OJS to link into itself are built using the `Request::url` function; likewise, all redirects into OJS are built using the `Request::redirect` function.

Database Support

The basic database functionality is provided by the ADODB library (http://adodb.sourceforge.net); atop the ADODB library is an additional layer of abstraction provided by the Data Access Objects (DAOs). These make use of a few base classes in the `classes/db` directory that are extended to provide specific functionality.

- `DAORegistry.inc.php`: This implements a central registry of Data Access Objects; when a DAO is desired, it is fetched through the DAO registry.
- `DBConnection.inc.php`: All database connections are established via this class.
- `DAO.inc.php`: This provides a base class for all DAOs to extend. It provides functions for accessing the database via the `DBConnection` class.

In addition, there are several classes that assist with XML parsing and loading into the database:

- `XMLDAO.inc.php`: Provides operations for retrieving and modifying objects from an XML data source
- `DBDataXMLParser.inc.php`: Parses an XML schema into SQL statements

File Management

As files (e.g. galleys and journal logos) are stored on the server filesystem, rather than in the database, several classes are needed to manage this filesystem and interactions between the filesystem and the rest of the OJS. These classes can be found in the `classes/file` directory.

- `FileManager.inc.php`: The three subsequent file management classes extend this class. It provides the necessary basic functionality for interactions between the web server and the file system.
- `FileWrapper.inc.php`: This implements a wrapper around file access functions that is more broadly compatible than the built-in access methods.
- `ArticleFileManager.inc.php`: This extends `FileManager` by adding features required to manage files associated with a particular article. For example, it is responsible for managing the directory structure associated with article files. See also `ArticleFile` and `ArticleFileDAO`.
- `PublicFileManager.inc.php`: Many files, such as journal logos, are "public" in that they can be accessed by anyone without need for authentication. These files are managed by this class, which extends the `FileManager` class.
- `TemporaryFileManager.inc.php`: This class allows the system to store temporary files associated with a particular user so that they can be maintained across requests. For example, if a user is composing an email with an attachment, the attachment must be stored on the server until the user is finished composing; this may involve multiple requests. `TemporaryFileManager` also extends `FileManager`. See also `TemporaryFile` and `TemporaryFileDAO`.

Scheduled Tasks

OJS 2.x is capable of performing regularly-scheduled automated tasks with the help of the operating system, which is responsible for launching the `tools/runScheduledTasks.php` script via a mechanism like UNIX's `cron`. Scheduled tasks must be enabled in the `config.inc.php` configuration file and the journal's settings.

Automated tasks are configured in `registry/scheduledTasks.xml` and

information like the date of a task's last execution is stored in the `scheduled_tasks` database table.

The `ScheduledTask` model class and the associated `ScheduledTaskDAO` are responsible for managing these database entries. In addition, the scheduled tasks themselves are implemented in the `classes/tasks` directory. Currently, only the `ReviewReminder` task is implemented, which is responsible for reminding reviewers that they have an outstanding review to complete or indicate acceptance of.

These tasks, which extend the `ScheduledTask` model class and are launched by the `runScheduledTasks` tool, must implement the `execute()` method with the task to be performed.

Security

The OJS 2.x security model is based on the concept of roles. The system's roles are predefined (e.g. author, reader, section editor, proofreader, etc) and users are assigned to roles on a per-journal basis. A user can have multiple roles within the same journal.

Roles are managed via the `Role` model class and associated `RoleDAO`, which manage the `roles` database table and provide security checking.

The `Validation` class (`classes/security/Validation.inc.php`) is responsible for ensuring security in interactions between the client browser and the web server. It handles login and logout requests, generates password hashes, and provides many useful shortcut functions for security- and validation-related issues. The `Validation` class is the preferred means of access for these features.

Session Management

Session management is provided by the `Session` model class, `SessionDAO`, and the `SessionManager` class (`classes/session/SessionManager.inc.php`).

While `Session` and `SessionDAO` manage database-persistent sessions for

individual users, SessionManager is concerned with the technical specifics of sessions as implemented for PHP and Apache.

Template Support

Smarty templates (http://smarty.php.net) are accessed and managed via the TemplateManager class (classes/template/TemplateManager.inc.php), which performs numerous common tasks such as registering additional Smarty functions such as {translate ...}, which is used for localization, and setting up commonly-used template variables such as URLs and date formats.

Paging Classes

Several classes facilitate the paged display of lists of items, such as submissions:

```
ItemIterator
ArrayItemIterator
DAOResultFactory
DBRowIterator
VirtualArrayIterator
```

The ItemIterator class is an abstract iterator, for which specific implementations are provided by the other classes. All DAO classes returning subclasses of ItemIterator should be treated as though they were returning ItemIterators.

Each iterator represents a single “page” of results. For example, when fetching a list of submissions from SectionEditorSubmissionDAO, a range of desired row numbers can be supplied; the ItemIterator returned (specifically an ArrayIterator) contains information about that range.

ArrayItemIterator and VirtualArrayIterator provide support for iterating through PHP arrays; in the case of VirtualArrayIterator, only the desired page's entries need be supplied, while ArrayItemIterator will take the entire set of results as a parameter and iterate through only those entries on the current page.

DAOResultFactory, the most commonly used and preferred ItemIterator

subclass, takes care of instantiating Model objects corresponding to the results using a supplied DAO and instantiation method.

`DBRowIterator` is an `ItemIterator` wrapper around the ADODB result structure.

Plugins

There are several classes included with the OJS 2.x distribution to help support a plugin registry. For information on the plugin registry, see the section titled "Plugins".

Common Tasks

The following sections contain code samples and further description of how the various classes interact.

Sending Emails

Emails templates for each locale are stored in an XML file called `dbscripts/xml/data/locale/[localeName]/email_templates_data.xml`. Each email has an identifier (called `email_key` in the XML file) such as `SUBMISSION_ACK`. This identifier is used in the PHP code to retrieve a particular email template, including body text and subject.

The following code retrieves and sends the `SUBMISSION_ACK` email, which is sent to authors as an acknowledgment when they complete a submission. (This snippet assumes that the current article ID is stored in `$articleId`.)

```
// Fetch the article object using the article DAO.
$articleDao = &DAORegistry::getDAO('ArticleDAO');
$article = &$articleDao->getArticle($articleId);

// Load the required ArticleMailTemplate class
import('mail.ArticleMailTemplate');

// Retrieve the mail template by name.
$mail = &new ArticleMailTemplate($article, 'SUBMISSION_ACK');
```

```
if ($mail->isEnabled()) {
    // Get the current user object and assign them as the recipient of this message.
    $user = &Request::getUser();
    $mail->addRecipient($user->getEmail(), $user->getFullName());

    // Get the current journal object.
    $journal = &Request::getJournal();

    // This template contains variable names of the form {$variableName} that need to
    // be replaced with the appropriate values. Note that while the syntax is similar
    // to that used by Smarty templates, email templates are not Smarty templates. Only
    // direct variable replacement is supported.
    $mail->assignParams(array(
        'authorName' => $user->getFullName(),
        'authorUsername' => $user->getUsername(),
        'editorialContactSignature' => $journal->getSetting('contactName') .
                "\n" . $journal->getTitle(),
        'submissionUrl' => Request::getPageUrl() .
                '/author/submission/' . $article->getArticleId()
    ));

    $mail->send();
}
```

Database Interaction with DAOs

The following code snippet retrieves an article object using the article ID supplied in the `$articleId` variable, changes the title, and updates the database with the new values.

```
// Fetch the article object using the article DAO.
$articleDao = &DAORegistry::getDAO('ArticleDAO');
$article = &$articleDao->getArticle($articleId);

$article->setTitle('This is the new article title.');

// Update the database with the modified information.
$articleDao->updateArticle($article);
```

Similarly, the following snippet deletes an article from the database.

```
// Fetch the article object using the article DAO.
$articleDao = &DAORegistry::getDAO('ArticleDAO');
$article = &$articleDao->getArticle($articleId);

// Delete the article from the database.
$articleDao->deleteArticle($article);
```

The previous task could be accomplished much more efficiently with the following:

```
// Fetch the article object using the article DAO.
$articleDao = &DAORegistry::getDAO('ArticleDAO');
```

```
$articleDao->deleteArticleById($articleId);
```

Generally speaking, the DAOs are responsible for deleting dependent database entries. For example, deleting an article will delete that article's authors from the database. Note that this is accomplished in PHP code rather than database triggers or other database-level integrity functionality in order to keep database requirements as low as possible.

User Interface

The User Interface is implemented as a large set of Smarty templates, which are called from the various Page classes. (See the section titled “Request Handling”.)

These templates are responsible for the HTML markup of each page; however, all content is provided either by template variables (such as article titles) or through locale-specific translations using a custom Smarty function.

You should be familiar with Smarty templates before working with OJS 2.x templates. Smarty documentation is available from http://smarty.php.net.

Variables

Template variables are generally assigned in the Page or Form class that calls the template. In addition, however, many variables are assigned by the `TemplateManager` class and are available to all templates:

- `defaultCharset`: the value of the “c`lient_charset`” setting from the `[i18n]` section of the `config.inc.php` configuration file
- `currentLocale`: The symbolic name of the current locale
- `baseUrl`: Base URL of the site, e.g. http://www.mylibrary.com
- `requestedPage`: The symbolic name of the requested page
- `pageTitle`: Default name of locale key of page title; this should be replaced with a more appropriate setting in the template
- `siteTitle`: If the user is currently browsing a page associated with a journal, this is the journal title; otherwise the site title from Site Configuration

- `publicFilesDir`: The URL to the currently applicable Public Files directory (See the section titled File Management)
- `pagePath`: Path of the requested page and operation, if applicable, prepended with a slash; e.g. `/user/profile`
- `currentUrl`: The full URL of the current page
- `dateFormatTrunc`: The value of the `date_format_trunc` parameter in the `[general]` section of the `config.inc.php` configuration file; used with the `date_format` Smarty function
- `dateFormatShort`: The value of the `date_format_short` parameter in the `[general]` section of the `config.inc.php` configuration file; used with the `date_format` Smarty function
- `dateFormatLong`: The value of the `date_format_long` parameter in the `[general]` section of the `config.inc.php` configuration file; used with the `date_format` Smarty function
- `datetimeFormatShort`: The value of the `datetime_format_short` parameter in the `[general]` section of the `config.inc.php` configuration file; used with the `date_format` Smarty function
- `datetimeFormatLong`: The value of the `datetime_format_long` parameter in the `[general]` section of the `config.inc.php` configuration file; used with the `date_format` Smarty function
- `currentLocale`: The name of the currently applicable locale; e.g. `en_US`
- `articleSearchByOptions`: Names of searchable fields used by the search feature in the sidebar and on the Search page
- `userSession`: The current Session object
- `isUserLoggedIn`: Boolean indicating whether or not the user is logged in
- `loggedInUsername`: The current user's username, if applicable
- `page_links`: The maximum number of page links to be displayed for a paged list within the current Journal or site context.
- `items_per_page`: The maximum number of items to display per page of a paged list within the current Journal or site context.

Additionally, if the user is browsing pages belonging to a particular journal, the following variables are available:

- `currentJournal`: The currently-applicable journal object (of the `Journal` class)
- `alternateLocale1`: First alternate locale (`alternateLocale2`) journal setting
- `alternateLocale2`: Second alternate locale (`alternateLocale1`) journal setting

- `navMenuItems`: Navigation items (`navItems`) journal setting
- `pageHeaderTitle`: Used by `templates/common/header.tpl` to display journal-specific information
- `pageHeaderLogo`: Used by `templates/common/header.tpl` to display journal-specific information
- `alternatePageHeader`: Used by `templates/common/header.tpl` to display journal-specific information
- `metaSearchDescription`: Current journal's description; used in `meta` tags
- `metaSearchKeywords`: Current journal's keywords; used in `meta` tags
- `metaCustomHeaders`: Current journal's custom headers, if defined; used in `meta` tags
- `stylesheets`: An array of stylesheets to include with the template
- `pageFooter`: Custom footer content to be displayed at the end of the page

If multiple languages are enabled, the following variables are set:

- `enableLanguageToggle`: Set to `true` when this feature is enabled
- `languageToggleLocales`: Array of selectable locales

Functions & Modifiers

A number of functions have been added to Smarty's built-in template functions to assist in common tasks such as localization.

- `translate` (e.g. `{translate key="my.locale.key" myVar="value"}`): This function provides a locale-specific translation. (See the section called Localization.) Variable replacement is possible using Smarty-style syntax; using the above example, if the `locale.xml` file contains:
  ```
  <message key="my.locale.key">myVar equals "{$myVar}".</message>
  ```
  The resulting output will be:
  ```
  myVar equals "value".
  ```
  (Note that only direct variable replacements are allowed in locale files. You cannot call methods on objects or Smarty functions.)
- `assign` (e.g. `{translate|assign:"myVar" key="my.locale.key"}`): Assign a value to a template variable. This example is similar to `{translate ...}`, except that the result is assigned to the specified Smarty variable rather than being displayed to the browser.
- `html_options_translate` (e.g. `{html_options_translate values=$myValuesArray selected=$selectedOption}`): Convert an array of the form
  ```
  array('optionVal1' => 'locale.key.option1', 'optionVal2' => 'locale.key.option2')
  ```

to a set of HTML `<option>...</option>` tags of the form

```
<option value="optionVal1">Translation of "locale.key.option1" here</option>
<option value="optionVal2">Translation of "locale.key.option2" here</option>
```

for use in a Select menu.

- `get_help_id` (e.g. `{get_help_id key="myHelpTopic" url="true"}`): Displays the topic ID or a full URL (depending on the value of the `url` parameter) to the specific help page named.
- `icon` (e.g. `{icon name="mail" alt="..." url="http://link.url.com" disabled="true"}`): Displays an icon with the specified link URL, disabled or enabled as specified. The `name` paramter can take on the values `comment`, `delete`, `edit`, `letter`, `mail`, or `view`.
- `help_topic` (e.g. `{help_topic key="(dir)*.page.topic" text="foo"}`): Displays a link to the specified help topic, with the `text` parameter defining the link contents.
- `page_links`: (e.g. `{page_links iterator=$submissions}`): Displays the page links for the paged list associated with the `ItemIterator` subclass (in this example, `$submissions`).
- `page_info`: (e.g. `{$page_info name="submissions" iterator=$submissions}`): Displays the page information (e.g. page number and total page count) for the paged list associated with the `ItemIterator` subclass (in this case, `$submissions`).
- `iterate`: (e.g. `{$iterate from=submissions item=submission}`): Iterate through items in the specified `ItemIterator` subclass, with each item stored as a smarty variable with the supplied name. (This example iterates through items in the `$submissions` iterator, which each item stored as a template variable named `$submission`.) Note that there are no dollar-signs preceding the variable names -- the specified parameters are variable names, not variables themselves.
- `strip_unsafe_html`: (e.g. `{$myVar|strip_unsafe_html}`): Remove HTML tags and attributes deemed as "unsafe" for general use. This modifier allows certain simple HTML tags to be passed through to the remote browser, but cleans anything advanced that may be used for XSS-based attacks.
- `call_hook`: (e.g. {`call_hook name="Templates::Manager::Index::ManagementPages"`}) Call a plugin hook by name. Any plugins registered against the named hook will be called.

There are many examples of use of each of these functions in the templates provided with OJS 2.x.

Plugins

OJS 2.1 contains a full-fledged plugin infrastructure that provides developers with several mechanisms to extend and modify the system's behavior without modifying the codebase. The key concepts involved in this infrastructure are **categories**, **plugins**, and **hooks**.

A **plugin** is a self-contained collection of code and resources that implements an extension of or modification to OJS. When placed in the appropriate directory within the OJS codebase, it is loaded and called automatically depending on the **category** it is part of.

Each plugin belongs to a single **category**, which defines its behavior. For example, plugins in the `importexport` category (which are used to import or export OJS data) are loaded when the Journal Manager uses the “Import/Export Data” interface or when the command-line tool is launched. Import/export plugins must implement certain methods which are used for delegation of control between the plugin and OJS.

Plugins are loaded when the category they reside in is requested; for example, `importexport` plugins are loaded by the Page class `ImportExportHandler` (implemented in the file `pages/manager/ImportExportHandler.inc.php`). Requests are delegated to these plugins via the methods defined in the `ImportExportPlugin` class, which each plugin in this category extends.

Hooks are used by plugins as a notification tool and to override behaviors built into OJS. At many points in the execution of OJS code, a hook will be called by name – for example, `LoadHandler` in `index.php`. Any plugins that have been loaded and registered against that hook will have a chance to execute code to alter the default behavior of OJS around the point at which that hook was called.

While most of the plugin categories built into OJS define specific tasks like authorization and harvesting tasks, there is a `generic` category for plugins that do not suit any of the other categories. These are more complicated to write but offer much more flexibility in the types of alterations they can make to OJS. Hooks are generally intended for use with plugins in this category.

Objects & Classes

Plugins in OJS 2.x are object-oriented. Each plugin extends a class defining its category's functions and is responsible for implementing them.

Category	*Base Class*
`generic`	`GenericPlugin` (`classes/plugins/GenericPlugin.inc.php`)
`importexport`	`ImportExportPlugin` (`classes/plugins/ImportExportPlugin.inc.php`)
`auth`	`AuthPlugin` (`classes/plugins/AuthPlugin.inc.php`)
`gateways`	`GatewayPlugin` (`classes/plugins/GatewayPlugin.inc.php`)

Each base class contains a description of the functions that must be implemented by plugins in that category.

Plugins are managed by the `PluginRegistry` class (implemented in `classes/plugins/PluginRegistry.inc.php`). They can register hooks by using the `HookRegistry` class (implemented in `classes/plugins/HookRegistry.inc.php`).

Sample Plugin

The following code listings illustrate a basic sample plugin for the `generic` plugin category. This plugin can be installed by placing all of its files in a directory called `plugins/generic/example`.

This plugin will add an entry to the Journal Manager's list of functions, available by following the “Journal Manager” link from User Home.

Loader Stub

The plugin is loaded by OJS by loading a file in the plugin directory called `index.php`. This is a loader stub responsible for instantiating and returning the plugin object:

```
<?php
require('ExamplePlugin.inc.php');
return new ExamplePlugin();
?>
```

Plugin Object

The plugin object encapsulates the plugin and generally will do most of the work. In this case, since this plugin will be in the `generic` category, the object must extend the `GenericPlugin` class:

```
<?php
import('classes.plugins.GenericPlugin');
class ExamplePlugin extends GenericPlugin {
	function register($category, $path) {
		if (parent::register($category, $path)) {
			HookRegistry::register(
				'Templates::Manager::Index::ManagementPages',
				array(&$this, 'callback')
			);
			return true;
		}
		return false;
	}
	function getName() {
		return 'ExamplePlugin';
	}
	function getDisplayName() {
		return 'Example Plugin';
	}
```

```
        function getDescription() {
                return 'A description of this plugin';
        }
        function callback($hookName, $args) {
                $params =& $args[0];
                $smarty =& $args[1];
                $output =& $args[2];
                $output = '<li>&#187; <a href="http://pkp.sfu.ca">My New Link</a></li>';
                return false;
        }
}
?>
```

The above code illustrates a few of the most important parts of plugins: the `register` function, hook registration and callback, and plugin management.

Registration Function

Whenever OJS loads and registers a plugin, the plugin's `register(...)` function will be called. This is an opportunity for the plugin to register against any hooks it needs, load configuration, initialize data structures, etc. In the above example, all the plugin needs to do (aside from calling the parent class's `register` function) is register against the `Templates::Manager::Index::ManagementPages` hook.

Another common task to perform in the registration function is loading locale data. Locale data should be included in subdirectories of the plugin's directory called `locale/[localeName]/locale.xml`, where `[localeName]` is the standard symbolic name of the locale, such as `en_US` for US English. In order for these data files to be loaded, plugins should call `$this->addLocaleData();` in the registration function after calling the parent registration function.

Hook Registration and Callback

The above example serves as a clear illustration of hook registration and callback; along with the list of hooks below, this should provide all the required information

for extending OJS using hooks. However, there are a few important details that need further examination.

The process by which a plugin registers against a hook is as follows:

```
HookRegistry::register(
      'Templates::Manager::Index::ManagementPages',
      array(&$this, 'callback')
);
```

In the example above, the parameters to `HookRegistry::register` are:

1. The name of the hook. See the complete list of hooks below.
2. The callback function to which control should be passed when the hook is encountered. This is the same callback format used by PHP's `call_user_func` function; see the documentation at http://php.net for more information. It is important that `$this` be included in the array by reference, or you may encounter problems with multiple instances of the plugin object.

The definition of the callback function (named and located in the above registration call) is:

```
function callback($hookName, $args) {
      $params =& $args[0];
      $smarty =& $args[1];
      $output =& $args[2];
      ...
}
```

The parameter list for the callback function is always the same:

1. The name of the hook that resulted in the callback receiving control (which can be useful when several hook registrations are made with the same callback function), and
2. An array of additional parameters passed to the callback. The contents of this array depend on the hook being registered against. Since this is a template hook, the callback can expect the three parameters named above.

The array-based passing of parameters is slightly cumbersome, but it allows hook

calls to compatibly pass references to parameters if desired. Otherwise, for example, the above code would receive a duplicated Smarty object rather than the actual Smarty object and any changes to attributes of the `$smarty` object would disappear upon returning.

Finally, the return value from a hook callback is very important. If a hook callback returns `true`, the hook registry considers this callback to have definitively "handled" the hook and will not call further registered callbacks on the same hook. If the callback returns `false`, other callbacks registered on the same hook after the current one will have a chance to handle the hook call.

The above example adds a link to the Journal Manager's list of management functions. If another plugin (or even the same plugin) was registered to add another link to the same list, and this plugin returned `true`, the other plugin's hook registration would not be called.

Plugin Management

In the example plugin, there are three functions that provide metadata about the plugin: `getName()`, `getDisplayName()`, and `getDescription()`. These are part of a management interface that is available to the Journal Manager under "System Plugins".

The result of the `getName()` call is used to refer to the plugin symbolically and need not be human-readable; however, the `getDisplayName()` and `getDescription()` functions should return localized values. This was not done in the above example for brevity.

The management interface allows plugins to specify various management functions the Journal Manager can perform on the plugin using the `getManagementVerbs()` and `manage($verb, $args)` functions. `getManagementVerbs()` should return an array of two-element arrays as follows:

```
$verbs = parent::getManagementVerbs();
$verbs[] = array('func1', Locale::translate('my.localization.key.for.func1'));
$verbs[] = array('func2', Locale::translate('my.localization.key.for.func2'));
```

Note that the parent call should be respected as above, as some plugin categories provide management verbs automatically.

Using the above sample code, the plugin should be ready to receive the management verbs `func1` and `func2` as follows (once again respecting any management verbs provided by the parent class):

```
function manage($verb, $args) {
	if (!parent::manage($verb, $args)) switch ($verb) {
		case 'func1':
			// Handle func1 here.
			break;
		case 'func2':
			// Handle func2 here.
			break;
		default:
			return false;
	}
	return true;
}
```

Additional Plugin Functionality

There are several additional plugin functionalities that may prove useful:

- **Plugin Settings:** Plugins can store and retrieve settings with a mechanism similar to Journal Settings. Use the Plugin class's `getSetting` and `updateSetting` functions.
- **Templates:** Any plugin can keep templates in its plugin directory and display them by calling:
  `$templateMgr->display($this->getTemplatePath() . 'templateName.tpl');`
  See the native import/export plugin for an example.
- **Schema Management:** By overriding `getInstallSchemaFile()` and

placing the named schema file in the plugin directory, `generic` plugins can make use of OJS's schema-management features. This function is called on OJS install or upgrade.

- **Data Management:** By overriding `getInstallDataFile()` and placing the named data file in the plugin directory, `generic` plugins can make use of OJS's data installation feature. This function is called on OJS install or upgrade.
- **Helper Code:** Helper code in the plugin's directory can be imported using `$this->import('HelperCode'); // imports HelperCode.inc.php`

Hook List

The following list describes all the hooks built into OJS as of release 2.1. Ampersands before variable names (e.g. `&$sourceFile`) indicate that the parameter has been passed to the hook callback in the parameters array by reference and can be modified by the hook callback. The effect of the hook callback's return value is specified where applicable; in addition to this, the hook callback return value will always determine whether or not further callbacks registered on the same hook will be skipped.

Name	*Parameters*	*Description*
`LoadHandler`	`&$page, &$op, &$sourceFile`	Called by OJS's main `index.php` script after the page (`&$page`), operation (`&$op`), and handler code file (`&$sourceFile`) names have been determined, but before `$sourceFile` is loaded. Can be used to intercept browser requests for handling by the plugin. Returning `true` from the callback will prevent OJS from loading the handler stub in `$sourceFile`.
`ArticleEmailLogDAO::_returnLogEntryFromRow`	`&$entry, &$row`	Called after `ArticleEmailLogDAO` builds an `ArticleEmailLogEntry` (`&$entry`) from the database row (`&$row`), but before the entry is passed back to the calling function.
`ArticleEventLogDAO::_returnLogEntryFromRow`	`&$entry, &$row`	Called after `ArticleEventLogDAO` builds an `ArticleEventLogEntry` (`&$entry`) from the

Name	*Parameters*	*Description*
		database row (`&$row`), but before the entry is passed back to the calling function.
`ArticleCommentDAO::_returnArticleCommentFromRow`	`&$articleComment, &$row`	Called after `ArticleCommentDAO` builds an `ArticleComment` (`&$articleComment`) from the database row (`&$row`), but before the comment is passed back to the calling function.
`ArticleDAO::_returnArticleFromRow`	`&$article, &$row`	Called after `ArticleDAO` builds an `Article` (`&$article`) from the database row (`&$row`), but before the article is passed back to the calling function.
`ArticleFileDAO::_returnArticleFileFromRow`	`&$articleFile, &$row`	Called after `ArticleFileDAO` builds an `ArticleFile` (`&$articleFile`) from the database row (`&$row`), but before the article file is passed back to the calling function.
`ArticleGalleyDAO::_returnGalleyFromRow`	`&$galley, &$row`	Called after `ArticleGalleyDAO` builds an `ArticleGalley` (`&$galley`) from the database row (`&$row`), but before the galley is passed back to the calling function.
`ArticleNoteDAO::_returnArticleNoteFromRow`	`&$articleNote, &$row`	Called after `ArticleNoteDAO` builds an `ArticleNote` (`&$articleNote`) from the database row (`&$row`), but before the entry is passed back to the calling function.
`AuthorDAO::_returnAuthorFromRow`	`&$author, &$row`	Called after `AuthorDAO` builds an `Author` (`&$author`) from the database row (`&$row`), but before the author is passed back to the calling function.
`SuppFileDAO::_returnSuppFileFromRow`	`&$suppFile, &$row`	Called after `SuppFileDAO` builds an `SuppFile` (`&$suppFile`) from the database row (`&$row`), but before the supplementary file is passed back to the calling function.
`PublishedArticleDAO::_returnPublishedArticleFromRow`	`&$publishedArticle, &$row`	Called after `PublishedArticleDAO` builds a `PublishedArticle` (`&$publishedArticle`) from the database row (`&$row`), but before the published article is passed back to the calling

Name	*Parameters*	*Description*
		function.
`CommentDAO::_returnCommentFromRow`	`&$comment, &$row, &$childLevels`	Called after `CommentDAO` builds a `Comment` (`&$comment`) from the database row (`&$row`), before fetching `&$childLevels` child comments and returning the comment to the calling function. Returning `true` will prevent OJS from fetching any child comments.
`Request::redirect`	`&$url`	Called before `Request::redirect` performs a redirect to `&$url`. Returning true will prevent OJS from performing the redirect after the hook is finished. Can be used to intercept and rewrite redirects.
`Request::getBaseUrl`	`&$baseUrl`	Called the first time `Request::getBaseUrl` is called after the base URL has been determined but before returning it to the caller. This value is used for all subsequent calls.
`Request::getBasePath`	`&$basePath`	Called the first time `Request::getBasePath` is called after the base path has been determined but before returning it to the caller. This value is used for all subsequent calls.
`Request::getIndexUrl`	`&$indexUrl`	Called the first time `Request::getIndexUrl` is called after the index URL has been determined but before returning it to the caller. This value is used for all subsequent calls.
`Request::getCompleteUrl`	`&$completeUrl`	Called the first time `Request::getCompleteUrl` is called after the complete URL has been determined but before returning it to the caller. This value is used for all subsequent calls.
`Request::getRequestUrl`	`&$requestUrl`	Called the first time `Request::getRequestUrl` is called after the request URL has been determined but before returning it to the caller. This value is used for all subsequent calls.

Name	*Parameters*	*Description*
`Request::getQueryString`	`&$queryString`	Called the first time `Request::getQueryString` is called after the query string has been determined but before returning it to the caller. This value is used for all subsequent calls.
`Request::getRequestPath`	`&$requestPath`	Called the first time `Request::getRequestPath` is called after the request path has been determined but before returning it to the caller. This value is used for all subsequent calls.
`Request::getServerHost`	`&$serverHost`	Called the first time `Request::getServerHost` is called after the server host has been determined but before returning it to the caller. This value is used for all subsequent calls.
`Request::getProtocol`	`&$protocol`	Called the first time `Request::getProtocol` is called after the protocol (`http` or `https`) has been determined but before returning it to the caller. This value is used for all subsequent calls.
`Request::getRemoteAddr`	`&$remoteAddr`	Called the first time `Request::getRemoteAddr` is called after the remote address has been determined but before returning it to the caller. This value is used for all subsequent calls.
`Request::getRemoteDomain`	`&$remoteDomain`	Called the first time `Request::getRemoteDomain` is called after the remote domain has been determined but before returning it to the caller. This value is used for all subsequent calls.
`Request::getUserAgent`	`&$userAgent`	Called the first time `Request::getUserAgent` is called after the user agent has been determined but before returning it to the caller. This value is used for all subsequent calls.
`Request::getRequeste`	`&$journal`	Called the first time

Name	*Parameters*	*Description*
dJournalPath		Request::getRequestedJournalPath is called after the requested journal path has been determined but before returning it to the caller. This value is used for all subsequent calls.
[Anything]DAO::Constructor	&$dataSource	Called whenever the named DAO's constructor is called with the specified &$dataSource. This hook should only be used with PHP >= 4.3.0.
[Anything]DAO::[Any function calling DAO::retrieve]	&$sql, &$params, &$value	Any DAO function calling DAO::retrieve will cause a hook to be triggered. The SQL statement in &$sql can be modified, as can the ADODB parameters in &$params. If the hook callback is intended to replace the function of this call entirely, &$value should receive the retrieve call's intended result and the hook should return true. This hook should only be used with PHP >= 4.3.0.
[Anything]DAO::[Any function calling DAO::retrieveCached]	&$sql, &$params, &$secsToCache, &$value	Any DAO function calling DAO::retrieveCached will cause a hook to be triggered. The SQL statement in &$sql can be modified, as can the ADODB parameters in &$params and the seconds-to-cache value in &$secsToCache. If the hook callback is intended to replace the function of this call entirely, &$value should receive the retrieve call's intended result and the hook should return true. This hook should only be used with PHP >= 4.3.0.
[Anything]DAO::[Any function calling DAO::retrieveLimit]	&$sql, &$params, &$numRows, &$offset, &$value	Any DAO function calling DAO::retrieveCached will cause a hook to be triggered. The SQL statement in &$sql can be modified, as can the ADODB parameters in &$params, and the fetch seek and limit specified in &$offset and &$numRows. If the hook callback is intended to replace the function of this call entirely, &$value should receive the retrieve call's intended result and

Name	*Parameters*	*Description*
		the hook should return `true`. This hook should only be used with PHP >= 4.3.0.
`[Anything]DAO::[Any function calling DAO::retrieveRange]`	`&$sql, &$params, &$dbResultRange, &$value`	Any DAO function calling `DAO::retrieveRange` will cause a hook to be triggered. The SQL statement in `&$sql` can be modified, as can the ADODB parameters in `&$params` and the range information in `&$dbResultRange`. If the hook callback is intended to replace the function of this call entirely, `&$value` should receive the retrieve call's intended result and the hook should return `true`. This hook should only be used with PHP >= 4.3.0.
`[Anything]DAO::[Any function calling DAO::update]`	`&$sql, &$params, &$value`	Any DAO function calling `DAO::update` will cause a hook to be triggered. The SQL statement in `&$sql` can be modified, as can the ADODB parameters in `&$params`. If the hook callback is intended to replace the function of this call entirely, `&$value` should receive the retrieve call's intended result and the hook should return `true`. This hook should only be used with PHP >= 4.3.0.
`TemporaryFileDAO::_returnTemporaryFileFromRow`	`&$temporaryFile, &$row`	Called after `TemporaryFileDAO` builds a `TemporaryFile` (`&$temporaryFile`) from the database row (`&$row`), but before the temporary file is passed back to the calling function.
`Locale::_cacheMiss`	`&$id, &$locale, &$value`	Called if a locale key couldn't be found in the locale cache. `&$id` is the key for the missing locale string, `&$locale` is the locale name (e.g. `en_US`). If this hook is to provide the result for this cache miss, the value should be stored in `&$value` and the hook callback should return `true`.
`Install::installer`	`&$installer, &$descriptor,`	Called when the installer is instantiated with

Name	*Parameters*	*Description*
	`&$params`	the descriptor path in `&$descriptor` and the parameters in `&$params`. If the hook returns `true`, the usual initialization of `Installer` attributes will not be performed.
`Installer::destroy`	`&$installer`	Triggered when the installer cleanup method is called.
`Installer::preInstall`	`&$installer, &$result`	Called after the installer's pre-installation tasks are completed but before the success/failure result in `&$result` is returned.
`Installer::postInstall`	`&$installer, &$result`	Called when the installer's post-installation tasks are completed but before the success/failure result in `&$result` is returned.
`Installer::parseInstaller`	`&$installer, &$result`	Called after the installer has completed parsing the installation task set but before the success/failure result in `&$result` is returned.
`Installer::executeInstaller`	`&$installer, &$result`	Called after the installer has executed but before the success/failure result in `&$result` is returned.
`Installer::updateVersion`	`&$installer, &$result`	Called after the installer has updated the system version information but before the success/failure result in `&$result` is returned.
`IssueAction::subscriptionRequired`	`&$journal, &$issue, &$result`	Called after OJS has determined whether or not a subscription is required for viewing `&$issue` in `&$journal` but before the true/false value `&$result` is returned.
`IssueAction::subscribedUser`	`&$journal, &$result`	Called after OJS has determined whether or not the current user is subscribed to `&$journal`, before the true/false value `&$result` is returned.
`IssueAction::subscribedDomain`	`&$journal, &$result`	Called after OJS has determined whether or not the current user comes from a domain subscribing to `&$journal`, before the true/false value `&$result` is returned.

Name	*Parameters*	*Description*
`IssueDAO::_returnIssueFromRow`	`&$issue, &$row`	Called after `IssueDAO` builds an `Issue` (`&$issue`) from the database row (`&$row`), but before the issue is passed back to the calling function.
`IssueDAO::_returnPublishedIssueFromRow`	`&$issue, &$row`	Called after `IssueDAO` builds a published `Issue` (`&$issue`) from the database row (`&$row`), but before the published issue is passed back to the calling function.
`JournalDAO::_returnJournalFromRow`	`&$journal, &$row`	Called after `JournalDAO` builds a `Journal` (`&$journal`) from the database row (`&$row`), but before the journal is passed back to the calling function.
`SectionDAO::_returnSectionFromRow`	`&$section, &$row`	Called after `SectionDAO` builds a `Section` (`&$section`) from the database row (`&$row`), but before the section is passed back to the calling function.
`EmailTemplateDAO::_returnBaseEmailTemplateFromRow`	`&$emailTemplate, &$row`	Called after `EmailTemplateDAO` builds a `BaseEmailTemplate` (`&$emailTemplate`) from the database row (`&$row`), but before the base email template is passed back to the calling function.
`EmailTemplateDAO::_returnLocaleEmailTemplateFromRow`	`&$emailTemplate, &$row`	Called after `EmailTemplateDAO` builds a localized `LocaleEmailTemplate` (`&$emailTemplate`) from the database row (`&$row`), but before the localized email template is passed back to the calling function.
`EmailTemplateDAO::_returnEmailTemplateFromRow`	`&$emailTemplate, &$row`	Called after `EmailTemplateDAO` builds an `EmailTemplate` (`&$emailTemplate`) from the database row (`&$row`), but before the email template is passed back to the calling function.
`Mail::send`	`&$mail, &$recipients, &$subject, &$mailBody, &$headers, &$additionalParameters`	Called just before an email with the specified parameters is sent. If this hook callback is to handle the email sending itself, the callback should return true and OJS's sending function

Name	*Parameters*	*Description*
		will be skipped.
RTDAO::_returnJournalRTFromRow	&$rt, &$row	Called after RTDAO builds a Reading Tools RT (&$rt) object from the database row (&$row), but before the Reading Tools object is passed back to the calling function.
RTDAO::_returnVersionFromRow	&$version, &$row	Called after RTDAO builds a Reading Tools Version (&$version) object from the database row (&$row), but before the Reading Tools version object is passed back to the calling function.
RTDAO::_returnSearchFromRow	&$search, &$row	Called after RTDAO builds a Reading Tools Search (&$search) object from the database row (&$row), but before the Reading Tools search object is passed back to the calling function.
RTDAO::_returnContextFromRow	&$context, &$row	Called after RTDAO builds a Reading Tools Context (&$context) object from the database row (&$row), but before the Reading Tools context object is passed back to the calling function.
AccessKeyDAO::_returnAccessKeyFromRow	&$accessKey, &$row	Called after AccessKeyDAO builds an AccessKey (&$accessKey) object from the database row (&$row), but before the access key is passed back to the calling function.
RoleDAO::_returnRoleFromRow	&$role, &$row	Called after RoleDAO builds a Role (&$role) object from the database row (&$row), but before the Role is passed back to the calling function.
SiteDAO::_returnSiteFromRow	&$site, &$row	Called after SiteDAO builds a Site (&$site) object from the database row (&$row), but before the Site is passed back to the calling function.
VersionDAO::_returnVersionFromRow	&$version, &$row	Called after VersionDAO builds a Version (&$version) object from the database row

Name	*Parameters*	*Description*
		(`&$row`), but before the Version is passed back to the calling function.
`AuthorAction::deleteArticleFile`	`&$articleFile, &$authorRevisions`	Called before OJS deletes the Author's article file `&$articleFile`.
`AuthorAction::uploadRevisedVersion`	`&$authorSubmission`	Called before OJS uploads a revised version of `&$authorSubmission`.
`AuthorAction::completeAuthorCopyedit`	`&$authorSubmission, &$email`	Called when the Author completes their copyediting step before OJS sends the email `&$email`, if enabled, and flags the copyediting step as completed.
`AuthorAction::copyeditUnderway`	`&$authorSubmission`	Called when the Author indicates that their copyediting step is underway, before OJS flags the underway date.
`AuthorAction::uploadCopyeditVersion`	`&$authorSubmission, &$copyeditStage`	Called when the author uploads a file for `&$authorSubmission` to the supplied `&$copyeditStage` before OJS accepts the file upload.
`AuthorAction::viewLayoutComments`	`&$article`	Called when the author requests the layout comments for the article `&$article`. If the hook registrant wishes to prevent OJS from instantiating and displaying the comment form, it should return `true` from the callback function.
`AuthorAction::postLayoutComment`	`&$article, &$emailComment`	Called when the author attempts to post a layout comment on the article `&$article`. If the hook registrant wishes to prevent OJS from recording the supplied comment, it should return `true` from the callback function.
`AuthorAction::viewEditorDecisionComments`	`&$article`	Called when the author requests the editor decision comments for the article `&$article`. If the hook registrant wishes to prevent OJS from instantiating and displaying the comment form, it should return `true` from the callback function.

Name	*Parameters*	*Description*
`AuthorAction::emailEditorDecisionComment`	`&$authorSubmission, &$email`	Called before OJS sends the editor decision comments for the article `&$authorSubmission` in the email message `&$email`. This hook should only be used on OJS > 2.1.0-1.
`AuthorAction::viewCopyeditComments`	`&$article`	Called when the author requests the copyedit comments for the article `&$article`. If the hook registrant wishes to prevent OJS from instantiating and displaying the comment form, it should return `true` from the callback function.
`AuthorAction::postCopyeditComment`	`&$article, &$emailComment`	Called when the author attempts to post a copyediting comment on the article `&$article`. If the hook registrant wishes to prevent OJS from recording the supplied comment, it should return `true` from the callback function.
`AuthorAction::viewProofreadComments`	`&$article`	Called when the author requests the proofreading comments for the article `&$article`. If the hook registrant wishes to prevent OJS from instantiating and displaying the comment form, it should return `true` from the callback function.
`AuthorAction::postProofreadComment`	`&$article, &$emailComment`	Called when the author attempts to post a proofreading comment on the article `&$article`. If the hook registrant wishes to prevent OJS from recording the supplied comment, it should return `true` from the callback function.
`AuthorAction::downloadAuthorFile`	`&$article, &$fileId, &$revision, &$canDownload, &$result`	Called when the author wishes to download an article file (`&$article`, `&$fileId`, `&$revision`) after OJS has determined whether or not the author has access to that file (modifiable boolean flag `&$canDownload`) but before the download itself begins. If the hook registrant wishes to override OJS's default

Name	*Parameters*	*Description*
		download behavior, it should return `true` from the callback function.
`AuthorSubmissionDAO::_returnAuthorSubmissionFromRow`	`&$authorSubmission, &$row`	Called after `AuthorSubmissionDAO` builds an `AuthorSubmission` (`&$authorSubmission`) object from the database row (`&$row`), but before the Author Submission is passed back to the calling function.
`Action::viewMetadata`	`&$article, &$role`	Called when a user in the given role (`&$role`) wishes to view the metadata for the given article (`&$article`). If the hook registrant wishes to prevent OJS from instantiating and displaying the regular metadata form, it should return `true` from its callback function.
`Action::saveMetadata`	`&$article`	Called before OJS updates the metadata for the specified article `&$article`. If the hook registrant wishes to prevent OJS from performing the update, it should return `true` from its callback function.
`Action::instructions`	`&$type, &$allowed`	Called before OJS displays the instructions of the requested type &$type (`copy`, `layout`, or `proof`); the allowed types are listed in `&$allowed`. If the hook registrant wishes to prevent OJS from displaying the instructions, it should return `true` from its callback function.
`Action::editComment`	`&$article, &$comment`	Called before OJS instantiates and displays the comment edit form for the given article (`&$article`) and comment (`&$comment`). If the hook registrant wishes to prevent OJS from doing this, it should return `true` from its callback function.
`Action::saveComment`	`&$article, &$comment, &$emailComment`	Called when a user attempts to save a comment (`&$comment`) on the article (`&$article`). If the hook registrant wishes to prevent OJS from saving the supplied comment, it should return `true` from the callback function.

Name	*Parameters*	*Description*
`Action::deleteComment`	`&$comment`	Called before OJS deletes the supplied comment. If the hook registrant wishes to prevent OJS from deleting the comment, it should return `true` from the callback function.
`CopyAssignmentDAO::_returnCopyAssignmentFromRow`	`&$copyAssignment, &$row`	Called after `CopyAssignmentDAO` builds a `CopyAssignment` (`&$copyAssignment`) object from the database row (`&$row`), but before the copyediting assignment is passed back to the calling function.
`CopyeditorAction::completeCopyedit`	`&$copyeditorSubmission, &$editAssignments, &$author, &$email`	Called before OJS sends the `COPYEDIT_COMPLETE` email (if enabled) and flags the copyeditor's initial copyediting stage as complete.
`CopyeditorAction::completeFinalCopyedit`	`&$copyeditorSubmission, &$editAssignments, &$email`	Called before OJS sends the `COPYEDIT_FINAL_COMPLETE` email (if enabled) and flags the copyeditor's final copyediting stage as complete.
`CopyeditorAction::copyeditUnderway`	`&$copyeditorSubmission`	Called before OJS flags the copyediting phase for the given submission (`&$copyeditorSubmission`) as underway. If the hook registrant wishes to prevent OJS from performing this flagging and the associated log entry, it should return `true` from its callback.
`CopyeditorAction::uploadCopyeditVersion`	`&$copyeditorSubmission`	Called before OJS uploads a revised version of `&$copyeditorSubmission`.
`CopyeditorAction::viewLayoutComments`	`&$article`	Called when the copyeditor requests the layout comments for the article `&$article`. If the hook registrant wishes to prevent OJS from instantiating and displaying the comment form, it should return `true` from the callback function.
`CopyeditorAction::postLayoutComment`	`&$article, &$emailComment`	Called when the copyeditor attempts to post a layout comment on the article `&$article`. If the hook registrant wishes to prevent OJS from recording the supplied comment, it should

Name	*Parameters*	*Description*
		return `true` from the callback function.
`CopyeditorAction::viewCopyeditComments`	`&$article`	Called when the copyeditor requests the copyediting comments for the article `&$article`. If the hook registrant wishes to prevent OJS from instantiating and displaying the comment form, it should return `true` from the callback function.
`CopyeditorAction::postCopyeditComment`	`&$article, &$emailComment`	Called when the copyeditor attempts to post a copyediting comment on the article `&$article`. If the hook registrant wishes to prevent OJS from recording the supplied comment, it should return `true` from the callback function. This hook should only be used with OJS > 2.1.0-1.
`CopyeditorAction::downloadCopyeditorFile`	`&$submission, &$fileId, &$revision, &$result`	Called when the copyeditor wishes to download an article file (`&$article`, `&$fileId`, `&$revision`) after OJS has determined whether or not the copyeditor has access to that file (modifiable boolean flag `&$canDownload`) but before the download itself begins. If the hook registrant wishes to override OJS's default download behavior, it should return `true` from the callback function. This hook should only be used with OJS > 2.1.0-1.
`CopyeditorSubmissionDAO::_returnCopyeditorSubmissionFromRow`	`&$copyeditorSubmission, &$row`	Called after `CopyeditorSubmissionDAO` builds a `CopyeditorSubmission` (`&$copyeditorSubmission`) object from the database row (`&$row`), but before the copyeditor submission is passed back to the calling function.
`EditAssignmentsDAO::_returnEditAssignmentFromRow`	`&$editAssignment, &$row`	Called after `EditAssignmentsDAO` builds an `EditAssignment` (`&$editAssignment`) object from the database row (`&$row`), but before the editing assignment is passed back to the calling function. This hook should only be used with

Name	*Parameters*	*Description*
		OJS > 2.1.0-1.
`EditorAction::assignEditor`	`&$editorSubmission, &$sectionEditor, &$email`	Called before OJS assigns the specified editor or section editor (`&$sectionEditor`) to the article (`&$editorSubmission`) and sends (if enabled) the supplied email `&$email`.
`EditorSubmissionDAO::_returnEditorSubmissionFromRow`	`&$editorSubmission, &$row`	Called after `EditorSubmissionDAO` builds an `EditorSubmission` (`&$editorSubmission`) object from the database row (`&$row`), but before the editor submission is passed back to the calling function.
`LayoutAssignmentDAO::_returnLayoutAssignmentFromRow`	`&$layoutAssignment, &$row`	Called after `LayoutAssignmentDAO` builds a `LayoutAssignment` (`&$layoutAssignment`) object from the database row (`&$row`), but before the layout assignment is passed back to the calling function.
`LayoutEditorAction::deleteGalley`	`&$article, &$galley`	Called before OJS deletes the specified galley (`&$galley`) for the article (`&$article`). If the hook registrant wishes to prevent OJS from deleting the galley, it should return `true`.
`LayoutEditorAction::deleteSuppFile`	`&$article, &$suppFile`	Called before OJS deletes the specified supplementary file (`&$suppFile`) for the article (`&$article`). If the hook registrant wishes to prevent OJS from deleting the supplementary file, it should return `true`.
`LayoutEditorAction::completeLayoutEditing`	`&$submission, &$layoutAssignment, &$editAssignments, &$email`	Called before OJS flags the layout editing assignment (`&$layoutAssignment`) for the article (`&$submission`) and sends (if enabled) the supplied email `&$email`.
`LayoutEditorAction::viewLayoutComments`	`&$article`	Called when the layout editor requests the layout comments for the article `&$article`. If the hook registrant wishes to prevent OJS from instantiating and displaying the comment form, it should return `true` from the callback function.

Name	*Parameters*	*Description*
`LayoutEditorAction::postLayoutComment`	`&$article, &$emailComment`	Called when the layout editor attempts to post a layout comment on the article `&$article`. If the hook registrant wishes to prevent OJS from recording the supplied comment, it should return `true` from the callback function.
`LayoutEditorAction::viewProofreadComments`	`&$article`	Called when the layout editor requests the proofreading comments for the article `&$article`. If the hook registrant wishes to prevent OJS from instantiating and displaying the comment form, it should return `true` from the callback function.
`LayoutEditorAction::postProofreadComment`	`&$article, &$emailComment`	Called when the layout editor attempts to post a proofreading comment on the article `&$article`. If the hook registrant wishes to prevent OJS from recording the supplied comment, it should return `true` from the callback function.
`LayoutEditorAction::downloadFile`	`&$article, &$fileId, &$revision, &$canDownload, &$result`	Called when the layout editor wishes to download an article file (`&$article`, `&$fileId`, `&$revision`) after OJS has determined whether or not the layout editor has access to that file (modifiable boolean flag `&$canDownload`) but before the download itself begins. If the hook registrant wishes to override OJS's default download behavior, it should pass a success boolean into `&$result` and return `true` from the callback function.
`LayoutEditorSubmissionDAO::_returnLayoutEditorSubmissionFromRow`	`&$submission, &$row`	Called after `LayoutEditorSubmissionDAO` builds a `LayoutEditorSubmission` (`&$submission`) object from the database row (`&$row`), but before the submission is passed back to the calling function.
`ProofAssignmentDAO::_returnProofAssignmentFromRow`	`&$proofAssignment, &$row`	Called after `ProofAssignmentDAO` builds a `ProofAssignment` (`&$proofAssignment`) object from the database row (`&$row`), but

Name	*Parameters*	*Description*
		before the proofreading assignment is passed back to the calling function.
`ProofreaderAction::selectProofreader`	`&$userId, &$article, &$proofAssignment`	Called before OJS designates the supplied user (`&$userId`) as a proofreader of the article (`&$article`) with the specified proof assignment (`&$proofAssignment`). If the hook registrant wishes to prevent OJS from performing its usual actions, it should return `true` from its callback.
`ProofreaderAction::queueForScheduling`	`&$article, &$proofAssignment`	Called when the proofreader with the given assignment (`&$proofAssignment`) queues the supplied article for scheduling (`&$article`). If the hook registrant wishes to prevent OJS from performing its usual actions, it should return `true` from its callback.
`ProofreaderAction::proofreadEmail`	`&$proofAssignment, &$email, &$mailType`	Called before OJS sends a proofreader email of the specified `&$mailType` (e.g. `PROOFREAD_LAYOUT_COMPLETE`) and flags the appropriate dates given the supplied `&$proofAssignment`.
`ProofreaderAction::authorProofreadingUnderway`	`&$submission, &$proofAssignment`	Called before OJS flags an author's proofreading assignment (`&$proofAssignment`) as underway for the article `&$submission`. If the hook registrant wishes to prevent OJS from flagging the assignment as underway, it should return `true` from its callback.
`ProofreaderAction::proofreaderProofreadingUnderway`	`&$submission, &$proofAssignment`	Called before OJS flags a proofreader's proofreading assignment (`&$proofAssignment`) as underway for the article `&$submission`. If the hook registrant wishes to prevent OJS from flagging the assignment as underway, it should return `true` from its callback.
`ProofreaderAction::layoutEditorProofread`	`&$submission, &$proofAssign`	Called before OJS flags a layout editor's

Name	Parameters	Description
ingUnderway	ment	proofreading assignment (&$proofAssignment) as underway for the article &$submission. If the hook registrant wishes to prevent OJS from flagging the assignment as underway, it should return true from its callback.
ProofreaderAction::downloadProofreaderFile	&$submission, &$fileId, &$revision, &$canDownload, &$result	Called when the proofreader wishes to download an article file (&$article, &$fileId, &$revision) after OJS has determined whether or not the proofreader has access to that file (modifiable boolean flag &$canDownload) but before the download itself begins. If the hook registrant wishes to override OJS's default download behavior, it should pass a success boolean into &$result and return true from the callback function.
ProofreaderAction::viewProofreadComments	&$article	Called when the proofreader requests the proofreading comments for the article &$article. If the hook registrant wishes to prevent OJS from instantiating and displaying the comment form, it should return true from the callback function.
ProofreaderAction::postProofreadComment	&$article, &$emailComment	Called when the proofreader attempts to post a proofreading comment on the article &$article. If the hook registrant wishes to prevent OJS from recording the supplied comment, it should return true from the callback function.
ProofreaderAction::viewLayoutComments	&$article	Called when the proofreader requests the layout comments for the article &$article. If the hook registrant wishes to prevent OJS from instantiating and displaying the comment form, it should return true from the callback function.
ProofreaderAction::postLayoutComment	&$article, &$emailCommen	Called when the proofreader attempts to post a

Name	Parameters	Description
	t	layout comment on the article &$article. If the hook registrant wishes to prevent OJS from recording the supplied comment, it should return true from the callback function.
ProofreaderSubmissionDAO::_returnProofreaderSubmissionFromRow	&$submission, &$row	Called after ProofreaderSubmissionDAO builds a ProofreaderSubmission (&$submission) object from the database row (&$row), but before the submission is passed back to the calling function.
ReviewAssignmentDAO::_returnReviewAssignmentFromRow	&$reviewAssignment, &$row	Called after ReviewAssignmentDAO builds a ReviewAssignment (&$reviewAssignment) object from the database row (&$row), but before the review assignment is passed back to the calling function.
ReviewerAction::confirmReview	&$reviewerSubmission, &$email, &$decline	Called before OJS records a reviewer's accepted/declined status (&$decline) on the supplied &$reviewAssignment and sends the editor the email &$email (if enabled).
ReviewerAction::recordRecommendation	&$reviewerSubmission, &$email, &$recommendation	Called before OJS records a reviewer's recommendation (&$recommendation) on the supplied &$reviewAssignment and sends the editor the email &$email (if enabled).
ReviewerAction::uploadReviewFile	&$reviewAssignment	Called before OJS updates the review file for the given &$reviewAssignment with the uploaded file.
ReviewerAction::deleteReviewerVersion	&$reviewAssignment, &$fileId, &$revision	Called before OJS deletes the supplied reviewer file (&$reviewAssignment, &$fileId, &$version). If the hook registrant wishes to prevent OJS from deleting, it should return true from its callback.
ReviewerAction::viewPeerReviewComments	&$user, &$article, &$reviewId	Called before OJS displays the peer review comments to the reviewer (&$user) for the given article (&$article) and review ID (&$reviewId). If the hook registrant wishes to

Name	*Parameters*	*Description*
		prevent OJS from displaying the reviews, it should return `true` from its callback.
`ReviewerAction::postPeerReviewComment`	`&$user, &$article, &$reviewId, &$emailComment`	Called before records a new comment on the given review ID (`&$reviewId`) by the reviewer (`&$user`) on an article (`&$article`). If the hook registrant wishes to prevent OJS from recording the comment, it should return `true` from its callback.
`ReviewerAction::downloadReviewerFile`	`&$article, &$fileId, &$revision, &$canDownload, &$result`	Called when the reviewer wishes to download an article file (`&$article`, `&$fileId`, `&$revision`) after OJS has determined whether or not the reviewer has access to that file (modifiable boolean flag `&$canDownload`) but before the download itself begins. If the hook registrant wishes to override OJS's default download behavior, it should pass a success boolean into `&$result` and return `true` from the callback function.
`ReviewerAction::editComment`	`&$article, &$comment, &$reviewId`	Called before OJS instantiates and displays the comment edit form for the reviewer for a given article (`&$article`) and comment (`&$comment`). If the hook registrant wishes to prevent OJS from doing this, it should return `true` from its callback function.
`ReviewerSubmissionDAO::_returnReviewerSubmissionFromRow`	`&$reviewerSubmission, &$row`	Called after `ReviewerSubmissionDAO` builds a `ReviewerSubmission` (`&$reviewerSubmissionDAO`) object from the database row (`&$row`), but before the review assignment is passed back to the calling function.
`SectionEditorAction::designateReviewVersion`	`&$sectionEditorSubmission`	Called before OJS designates the original file as the review version for the specified article (`&$sectionEditorSubmission`). To prevent OJS from performing this task, the hook registrant should return `true` from its callback

Name	*Parameters*	*Description*
		function.
`SectionEditorAction::changeSection`	`&$sectionEditorSubmission, &$sectionId`	Called before OJS changes the section of the submission (`&$sectionEditorSubmission`) to the section with the given section ID (`&$sectionId`). To prevent OJS from performing this task, the hook registrant should return `true` from its callback function.
`SectionEditorAction::recordDecision`	`&$sectionEditorSubmission, &$editorDecision`	Called before OJS records a decision (`&$editorDecision`) for a submission (`&$sectionEditorSubmission`). To prevent OJS from performing this task, the hook registrant should return `true` from its callback function.
`SectionEditorAction::addReviewer`	`&$sectionEditorSubmission, &$reviewerId`	Called before OJS creates a new review assignment for the specified reviewer (`&$reviewerId`) on a submission (`&$sectionEditorSubmission`). To prevent OJS from performing this task, the hook registrant should return `true` from its callback function.
`SectionEditorAction::clearReview`	`&$sectionEditorSubmission, &$reviewAssignment`	Called before OJS clears a review (`&$reviewAssignment`) on a submission (`&$sectionEditorSubmission`). To prevent OJS from performing this task, the hook registrant should return `true` from its callback.
`SectionEditorAction::notifyReviewer`	`&$sectionEditorSubmission, &$reviewAssignment, &$email`	Called before OJS flags the notification of a reviewer with a pending review (`&$reviewAssignment`) on a submission (`&$sectionEditorSubmission`), sending the associated reviewer request email `&$email` (if enabled).
`SectionEditorAction::cancelReview`	`&$sectionEditorSubmission, &$reviewAssignment, &$email`	Called before OJS cancels a review (`&$reviewAssignment`) on a submission (`&$sectionEditorSubmission`), sending the associated cancellation email (`&$email`) if

Name	*Parameters*	*Description*
		enabled.
SectionEditorAction::remindReviewer	&$sectionEditorSubmission, &$reviewAssignment, &$email	Called before OJS reminds a reviewer of a pending review assignment (&$reviewAssignment) on a submission (&$sectionEditorSubmission), sending the associated reminder email (&$email).
SectionEditorAction::thankReviewer	&$sectionEditorSubmission, &$reviewAssignment, &$email	Called before OJS thanks a reviewer for completing their review assignment (&$reviewAssignment) on a submission (&$sectionEditorSubmission), also sending the email &$email (if enabled).
SectionEditorAction::rateReviewer	&$reviewAssignment, &$reviewer, &$quality	Called before OJS records a quality rating (&$quality) for a reviewer (&$reviewer) on a review assignment (&$reviewAssignment). To prevent OJS from recording the rating, the hook registrant should return true from its callback.
SectionEditorAction::makeReviewerFileViewable	&$reviewAssignment, &$articleFile, &$viewable	Called before OJS records a new visibility setting (&$viewable) for a reviewer file (&$articleFile) belonging to a review assignment (&$reviewAssignment). To prevent OJS from recording the new setting, the hook registrant should return true from its callback.
SectionEditorAction::setDueDate	&$reviewAssignment, &$reviewer, &$dueDate, &$numWeeks	Called before OJS sets the due date on a reviewer (&$reviewer)'s review assignment (&$reviewAssignment) to &$dueDate. To prevent OJS from setting the due date, the hook registrant should return true from its callback.
SectionEditorAction::unsuitableSubmission	&$sectionEditorSubmission, &$author, &$email	Called before OJS records an author (&$author)'s submission (&$sectionEditorSubmission) as unsuitable, sending &$email (if enabled).

Name	*Parameters*	*Description*
`SectionEditorAction::notifyAuthor`	`&$sectionEditorSubmission, &$author, &$email`	Called before OJS sends an author notification email (`&$email`) to an author (`&$author`) regarding a submission (`&$sectionEditorSubmission`).
`SectionEditorAction::setReviewerRecommendation`	`&$reviewAssignment, &$reviewer, &$recommendation, &$acceptOption`	Called before a reviewer recommendation (`&$recommendation`) is recorded on a review assignment (`&$reviewAssignment`) for the reviewer `&$reviewer`. To prevent the recommendation from being recorded, the hook registrant should return `true` from its hook callback.
`SectionEditorAction::setCopyeditFile`	`&$sectionEditorSubmission, &$fileId, &$revision`	Called before OJS sets the copyeditor file for the submission `&$sectionEditorSubmission` to `&$fileId` with revision `&$revision`. To prevent this change from taking place, the hook registrant should return `true` from its callback.
`SectionEditorAction::resubmitFile`	`&$sectionEditorSubmission, &$fileId, &$revision`	Called before OJS resubmits a file (`&$fileId` and `&$revision`) belonging to submission `&$sectionEditorSubmission` for review. To prevent this action from taking place, the hook registrant should return `true` from its callback.
`SectionEditorAction::selectCopyeditor`	`&$sectionEditorSubmission, &$copyeditorId`	Called before OJS designates the user with ID `&$copyeditorId` as copyeditor for the submission `&$sectionEditorSubmission`. To prevent this from taking place, the hook registrant should return `true` from its callback.
`SectionEditorAction::notifyCopyeditor`	`&$sectionEditorSubmission, &$copyeditor, &$email`	Called before OJS notifies the copyeditor `&$copyeditor` about their copyediting assignment for submission `&$sectionEditorSubmission`, sending `&$email` if enabled.
`SectionEditorAction::initiateCopyedit`	`&$sectionEditorSubmission`	Called before flagging the beginning of an editor copyediting stage on submission `&$sectionEditorSubmission`. To prevent this from taking place, the hook registrant should

Name	*Parameters*	*Description*
		return `true` from its callback.
`SectionEditorAction::thankCopyeditor`	`&$sectionEditorSubmission, &$copyeditor, &$email`	Called before OJS thanks a copyeditor (`&$copyeditor`) for contributing to the submission `&$sectionEditorSubmission`, sending the email &`$email` if enabled.
`SectionEditorAction::notifyAuthorCopyedit`	`&$sectionEditorSubmission, &$author, &$email`	Called before OJS flags notification of an author (`&$author`) of their copyediting assignment on a submission (`&$sectionEditorSubmission`), sending the email `&$email` if enabled.
`SectionEditorAction::thankAuthorCopyedit`	`&$sectionEditorSubmission, &$author, &$email`	Called before OJS records thanking an author (`&$author`) for their copyediting contribution to `&$sectionEditorSubmission`, sending `&$email` if enabled.
`SectionEditorAction::notifyFinalCopyedit`	`&$sectionEditorSubmission, &$copyeditor, &$email`	Called before OJS records notifying the copyeditor (`&$copyeditor`) of their final copyediting stage for the submission `&$sectionEditorSubmission`, sending `&$email` if enabled.
`SectionEditorAction::thankFinalCopyedit`	`&$sectionEditorSubmission, &$copyeditor, &$email`	Called before OJS records thanking a copyeditor (`&$copyeditor`) for their final-round copyediting contribution to submission `&$sectionEditorSubmission`, sending `&$email` if enabled.
`SectionEditorAction::uploadReviewVersion`	`&$sectionEditorSubmission`	Called before OJS stores a new review version for the submission `&$sectionEditorSubmission`. To prevent OJS from doing this, the hook registrant should return `true` from its callback.
`SectionEditorAction::uploadEditorVersion`	`&$sectionEditorSubmission`	Called before OJS stores a new editing version for the submission `&$sectionEditorSubmission`. To prevent OJS from doing this, the hook registrant should return `true` from its callback.

Name	*Parameters*	*Description*
`SectionEditorAction::uploadCopyeditVersion`	`&$sectionEditorSubmission`	Called before OJS stores a new copyediting version for the submission `&$sectionEditorSubmission`. To prevent OJS from doing this, the hook registrant should return `true` from its callback.
`SectionEditorAction::completeCopyedit`	`&$sectionEditorSubmission`	Called before OJS records a completion date for copyediting on a submission (`&$sectionEditorSubmission`) performed for the editor (when the use of copyeditors is disabled). To prevent OJS from recording this, the hook registrant should return `true` from its callback.
`SectionEditorAction::completeFinalCopyedit`	`&$sectionEditorSubmission`	Called before OJS records a completion date for the final copyediting stage on a submission (`&$sectionEditorSubmission`) performed for the editor (when the use of copyeditors is disabled). To prevent OJS from recording this, the hook registrant should return `true` from its callback.
`SectionEditorAction::archiveSubmission`	`&$sectionEditorSubmission`	Called before OJS archives a submission (`&$sectionEditorSubmission`). If this action should not be performed, the hook registrant should return `true` from its callback.
`SectionEditorAction::restoreToQueue`	`&$sectionEditorSubmission`	Called before OJS restores a submission (&$sectionEditorSubmission) from the archives into an active queue. If this action should not be performed, the hook registrant should return `true` from its callback.
`SectionEditorAction::updateSection`	`&$submission, &$sectionId`	Called before OJS moves the article `&$submission` into the section with ID `&$sectionId`. To prevent OJS from performing this action, the hook registrant should return `true` from its callback.
`SectionEditorAction::uploadLayoutVersion`	`&$submission, &$layoutAssignment`	Called before OJS stores a new layout version of the submission `&$submission` with the

Name	Parameters	Description
		given layout assignment `&$layoutAssignment`. To prevent OJS from performing this action, the hook registrant should return `true` from its callback.
`SectionEditorAction::assignLayoutEditor`	`&$submission, &$editorId`	Called before OJS assigns the layout editor with user ID `&$editorId` to submission `&$submission`. To prevent OJS from performing this action, the hook registrant should return `true` from its callback.
`SectionEditorAction::notifyLayoutEditor`	`&$submission, &$layoutEditor, &$layoutAssignment, &$email`	Called before OJS flags notification of the layout editor `&$layoutEditor` for the submission `&$submission`, sending the email `&$email` if enabled.
`SectionEditorAction::thankLayoutEditor`	`&$submission, &$layoutEditor, &$layoutAssignment, &$email`	Called before OJS records thanking the layout editor `&$layoutEditor` for their work on the submission `&$submission`, sending the email `&$email` if enabled.
`SectionEditorAction::deleteArticleFile`	`&$submission, &$fileId, &$revision`	Called before OJS deletes an article file (`&$fileId`, `&$revision`) for the submission `&$submission`. To prevent OJS from performing this action, the hook registrant should return `true` from its callback.
`SectionEditorAction::addSubmissionNote`	`&$articleId, &$articleNote`	Called before OJS adds a submission note (`&$articleNote`) to a submission with article ID `&$articleId`. To prevent OJS from performing this action, the hook registrant should return `true` from its callback.
`SectionEditorAction::removeSubmissionNote`	`&$articleId, &$noteId, &$fileId`	Called before OJS removes the submission note (`&$noteId`) and, if present, the associated file (`&$fileId`) from the submission with article ID `&$articleId`. To prevent OJS from performing this action, the hook registrant should return `true` from its callback.
`SectionEditorAction::updateSubmissionNot`	`&$articleId, &$articleNote`	Called before OJS saves the changes to a

Name	Parameters	Description
e		submission note on the article with ID &$articleId, already performed on the object &$articleNote but not committed to database. The new attached file, if one has been uploaded, has not been stored yet. To prevent OJS from storing the changes, the hook registrant should return true from its callback.
SectionEditorAction::clearAllSubmissionNotes	&$articleId	Called before OJS removes all submission notes and, if present, the associated files from the submission with article ID &$articleId. To prevent OJS from performing this action, the hook registrant should return true from its callback.
SectionEditorAction::viewPeerReviewComments	&$article, &$reviewId	Called before OJS displays the peer review comments to the editor or section editor for the given article (&$article) and review ID (&$reviewId). If the hook registrant wishes to prevent OJS from displaying the reviews, it should return true from its callback.
SectionEditorAction::postPeerReviewComment	&$article, &$reviewId, &$emailComment	Called before OJS records a new comment on the given review ID (&$reviewId) by the editor or section editor on an article (&$article). If the hook registrant wishes to prevent OJS from recording the comment, it should return true from its callback.
SectionEditorAction::viewEditorDecisionComments	&$article	Called when the Editor or Section Editor requests the editor decision comments for the article &$article. If the hook registrant wishes to prevent OJS from instantiating and displaying the comment form, it should return true from the callback function.
SectionEditorAction::postEditorDecisionComment	&$article, &$emailComment	Called before OJS records a new editor comment on the submission &$article. To prevent OJS from performing this action, the hook callback should return true.

Name	*Parameters*	*Description*
SectionEditorAction::emailEditorDecisionComment	&$sectionEditorSubmission, &$send	Called before OJS emails the author an editor decision comment on a submission (&$sectionEditorDecision).
SectionEditorAction::blindCcReviewsToReviewers	&$article, &$reviewAssignments, &$email	Called before OJS anonymously sends the reviews (&$reviewAssignments) email (&$email) to reviewers for the article &$article.
SectionEditorAction::viewCopyeditComments	&$article	Called when the editor or section editor requests the copyediting comments for the article &$article. If the hook registrant wishes to prevent OJS from instantiating and displaying the comment form, it should return true from the callback function.
SectionEditorAction::postCopyeditComment	&$article, &$emailComment	Called when the editor or section editor attempts to post a copyediting comment on the article &$article. If the hook registrant wishes to prevent OJS from recording the supplied comment, it should return true from the callback function.
SectionEditorAction::viewLayoutComments	&$article	Called when the section editor or editor requests the layout comments for the article &$article. If the hook registrant wishes to prevent OJS from instantiating and displaying the comment form, it should return true from the callback function.
SectionEditorAction::postLayoutComment	&$article, &$emailComment	Called when the section editor or editor attempts to post a layout comment on the article &$article. If the hook registrant wishes to prevent OJS from recording the supplied comment, it should return true from the callback function.
SectionEditorAction::viewProofreadComments	&$article	Called when the editor or section editor requests the proofreading comments for the article &$article. If the hook registrant wishes to prevent OJS from instantiating and

Name	*Parameters*	*Description*
		displaying the comment form, it should return `true` from the callback function.
`SectionEditorAction::postProofreadComment`	`&$article, &$emailComment`	Called when the editor or section editor attempts to post a proofreading comment on the article `&$article`. If the hook registrant wishes to prevent OJS from recording the supplied comment, it should return `true` from the callback function.
`SectionEditorAction::acceptReviewForReviewer`	`&$reviewAssignment, &$reviewer`	Called before OJS records the editor's acceptance of a review assignment (`&$reviewAssignment`) on behalf of a reviewer (`&$reviewer`). To prevent OJS from performing this action, the hook registrant should return `true` from its callback.
`SectionEditorAction::uploadReviewForReviewer`	`&$reviewAssignment, &$reviewer`	Called before OJS stores an editor's review upload for a review assignment (`&$reviewAssignment`) on behalf of a reviewer (`&$reviewer`). To prevent OJS from performing this action, the hook registrant should return `true` from its callback.
`SectionEditorSubmissionDAO::_returnSectionEditorSubmissionFromRow`	`&$sectionEditorSubmission, &$row`	Called after `SectionEditorSubmissionDAO` builds a `SectionEditorSubmission` (`&$sectionEditorSubmission`) object from the database row (`&$row`), but before the submission is passed back to the calling function.
`SectionEditorSubmissionDAO::_returnReviewerUserFromRow`	`&$user, &$row`	Called after `SectionEditorSubmissionDAO` builds a `User` (`&$user`) object from the database row (`&$row`), but before the submission is passed back to the calling function. The use of this hook is not recommended as it may be removed in the future.
`CurrencyDAO::_returnCurrencyFromRow`	`&$currency, &$row`	Called after `CurrencyDAO` builds a `Currency` (`&$currency`) object from the database row

Name	*Parameters*	*Description*
		(`&$row`), but before the currency is passed back to the calling function.
`SubscriptionDAO::_returnSubscriptionFromRow`	`&$subscription, &$row`	Called after `SubscriptionDAO` builds a `Subscription` (`&$subscription`) object from the database row (`&$row`), but before the subscription is passed back to the calling function.
`SubscriptionTypeDAO::_returnSubscriptionTypeFromRow`	`&$subscriptionType, &$row`	Called after `SubscriptionTypeDAO` builds a `SubscriptionType` (`&$subscriptionType`) object from the database row (`&$row`), but before the subscription type is passed back to the calling function.
`TemplateManager::display`	`&$templateMgr, &$template, &$sendContentType, &$charset`	Called before the template manager (`&$templateMgr`) sends the content type header with the given content type (`&$sendContentType`) and character set (`&$charset`) and displays a template (`&$template`). To prevent OJS from performing this action, the hook registrant should return `true` from its callback.
`UserDAO::_returnUserFromRow`	`&$user, &$row`	Called after `UserDAO` builds a `User` (`&$user`) object from the database row (`&$row`), but before the user is passed back to the calling function.
`GroupDAO::_returnGroupFromRow`	`&$group, &$row`	Called after `GroupDAO` builds a `Group` (`&$group`) object from the database row (`&$row`), but before the group is passed back to the calling function.
`GroupMembershipDAO::_returnMemberFromRow`	`&$membership, &$row`	Called after `GroupMembershipDAO` builds a `GroupMembership` (`&$membership`) object from the database row (`&$row`), but before the group membership is passed back to the calling function.
`Templates::About::Index::People`	`&$params, &$templateMgr, &$output`	Called at the end of the bulleted list in the

Name	*Parameters*	*Description*
		People section of the About page, within the `<ul class="plain">...</ul>` tag.
`Templates::About::Index::Policies`	`&$params, &$templateMgr, &$output`	Called at the end of the bulleted list in the Policies section of the About page, within the `<ul class="plain">...</ul>` tag.
`Templates::About::Index::Submissions`	`&$params, &$templateMgr, &$output`	Called at the end of the bulleted list in the Submissions section of the About page, within the `<ul class="plain">...</ul>` tag.
`Templates::About::Index::Other`	`&$params, &$templateMgr, &$output`	Called at the end of the bulleted list in the Other section of the About page, within the `<ul class="plain">...</ul>` tag.
`Templates::Admin::Index::SiteManagement`	`&$params, &$templateMgr, &$output`	Called at the end of the bulleted list in the Site Management section of the site administration page, within the `<ul class="plain">...</ul>` tag.
`Templates::Admin::Index::AdminFunctions`	`&$params, &$templateMgr, &$output`	Called at the end of the bulleted list in the Admin Functions section of the site administration page, within the `<ul class="plain">...</ul>` tag.
`Templates::Editor::Index::Submissions`	`&$params, &$templateMgr, &$output`	Called at the end of the bulleted list in the Submissions section of the editor's page, within the `<ul class="plain">...</ul>` tag.
`Templates::Editor::Index::Issues`	`&$params, &$templateMgr, &$output`	Called at the end of the bulleted list in the Issues section of the editor's page, within the `<ul class="plain">...</ul>` tag.
`Templates::Manager::Index::ManagementPages`	`&$params, &$templateMgr, &$output`	Called at the end of the bulleted list in the Management Pages section of the journal manager's page, within the `<ul class="plain">...</ul>` tag.
`Templates::Manager::Index::Users`	`&$params, &$templateMgr, &$output`	Called at the end of the bulleted list in the Users section of the journal manager's page, within the `<ul class="plain">...</ul>` tag.
`Templates::Manager::Index::Roles`	`&$params, &$templateMgr, &$output`	Called at the end of the bulleted list in the

Name	*Parameters*	*Description*
		Roles section of the journal manager's page, within the `<ul class="plain">...</ul>` tag.
`Templates::User::Index::Site`	`&$params, &$templateMgr, &$output`	Called after the site management link is displayed (if applicable) in the user home, within the `<ul class="plain">...</ul>` tag.
`Templates::User::Index::Journal`	`&$params, &$templateMgr, &$output`	Called at the end of the bulleted list displaying the roles for each journal in the user home, within the `<ul class="plain">...</ul>` tag.
`Templates::Admin::Index::MyAccount`	`&$params, &$templateMgr, &$output`	Called at the end of the bulleted list in the My Account section of the user home, within the `<ul class="plain">...</ul>` tag.

Translating OJS

To add support for other languages, XML files in the following directories must be translated and placed in an appropriately named directory (using ISO locale codes, e.g. `fr_FR`, is recommended):

- `locale/en_US`: This directory contains the main locale file with the majority of localized OJS text.
- `dbscripts/xml/data/locale/en_US`: This directory contains localized database data, such as email templates.
- `help/en_US`: This directory contains the help files for OJS.
- `registry/locale/en_US`: This directory contains additional localized information such as a country list.
- `rt/en_US`: This directory contains the Reading Tools.
- `plugins/[plugin category]/[plugin name]/locale`, where applicable: These directories contain plugin-specific locale strings.

The only critical files that need translation for the system to function properly are found in `locale/en_US`, `dbscripts/xml/data/locale/en_US`, and `registry/locale/en_US`.

New locales must also be added to the file `registry/locales.xml`, after which they can be installed in the system through the site administration web interface.

Translations can be contributed back to PKP for distribution with future releases of OJS.

Special Thanks

The Public Knowledge Project wishes to acknowledge the contributions of the following community members:

- Ramón Fonseca: Portuguese (`pt_BR`) translation
- Sergio Ruiz Pérez: Spanish (`es_ES`) translation

Obtaining More Information

For more information, see the PKP web site at http://pkp.sfu.ca. There is an OJS support forum available at http://pkp.sfu.ca/support/forum; this is the preferred method of contacting the OJS team. Please be sure to search the forum archives to see if your question has already been answered.

If you have a bug to report, see the bug tracking system at http://pkp.sfu.ca/bugzilla.

The team can be reached by email at pkp-support@sfu.ca.