Open Journal Systems

Version 2.0.1

Technical Reference

Revision 1

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`

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 requirements:

- **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)
- **Linux**, **BSD**, **Solaris**, **Mac OS X** operating systems

Other versions or platforms may work but are not supported and may not have been tested.

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 file, making use of variable replacement

as necessary.

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 explicit numeric or string constants;
- Filenames should match classnames; for example, the `SectionEditorAction` class is in the file `SectionEditorAction.inc.php`;
- Classnames 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.

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.

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 are avoided and special characters are displayed properly. For example, when displaying a username, use the following: `{$user->getUsername()|escape}`

Introduction

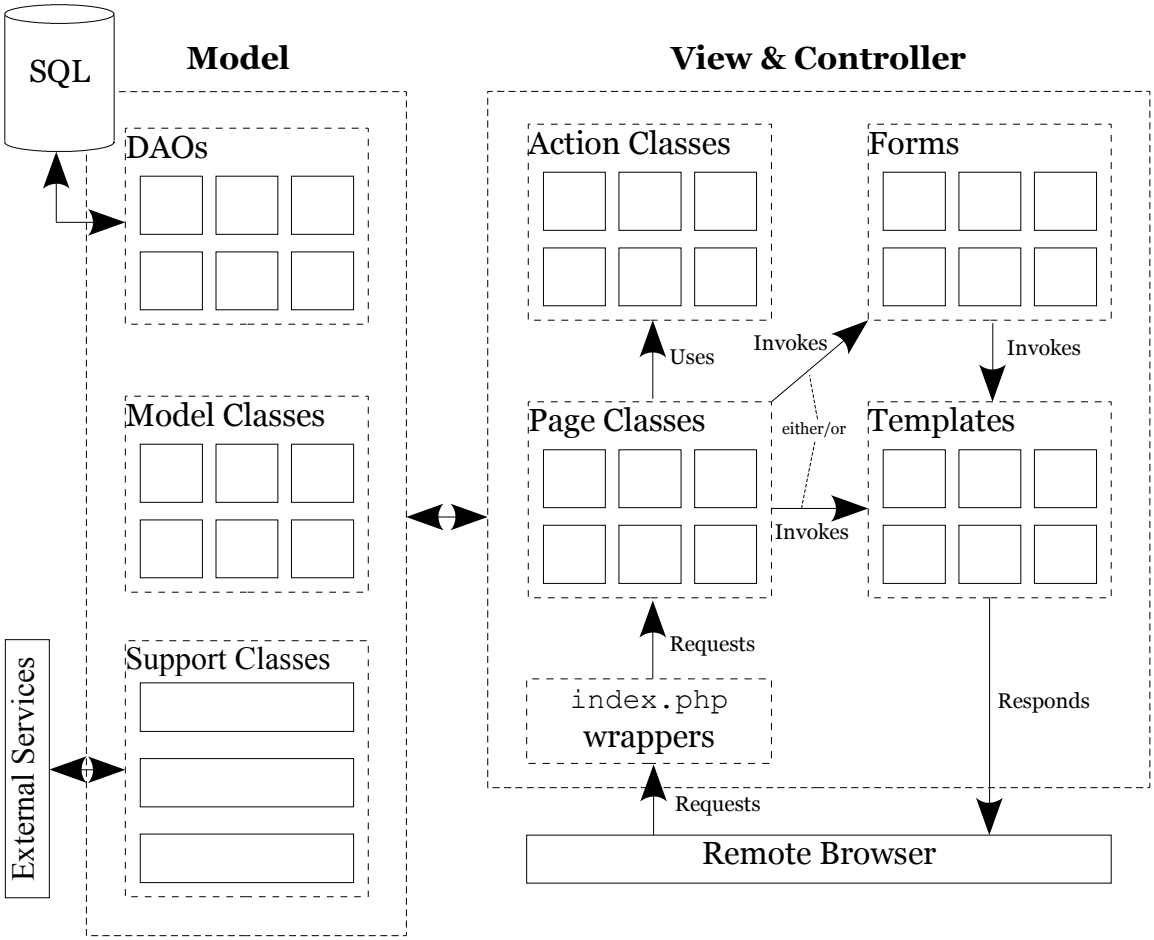
The design of Open Journal Systems 2.x is heavily structured for maintainability, flexibility and robustness. For this reason it may seem complex when first approached. Those familiar with Sun's Enterprise Java Beans technology or the Model-View-Controller (MVC) pattern will note many similarities.

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***
`classes`	Directory containing most of the OJS 2.x PHP code: Model classes, Data Access Objects (DAOs), core classes, etc
`config.TEMPLATE.inc.php`	Sample configuration file

File/Directory	*Description*
`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.

Request Handling Example

Predictably, delegation of request handling occurs based on the request URL. A typical (and fictional) 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 purpose is to ensure that the system is properly configured and 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 examing 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*
`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_keyword_index`	`article_id, keyword_id, type, assoc_id`	Provides an index associating keywords with articles they appear in
`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

Table Name	Primary Key	Description
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
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
issues	issue_id	Stores information about particular issues of 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

Table Name	*Primary Key*	*Description*
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
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

Table Name	*Primary Key*	*Description*
`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`.

```
Action
        AuthorAction
        CopyeditorAction
        LayoutEditorAction
        ProofreaderAction
        ReviewerAction
        SectionEditorAction
                EditorAction
DAO
        ArticleCommentDAO
        ArticleDAO
        ArticleEmailLogDAO
        ArticleEventLogDAO
        ArticleFileDAO
        ArticleGalleyDAO
        ArticleNoteDAO
        ArticleSearchDAO
        AuthorDAO
        AuthorSubmissionDAO
        CommentDAO
        CopyAssignmentDAO
        CopyeditorSubmissionDAO
        CurrencyDAO
        EditAssignmentDAO
        EditorSubmissionDAO
        EmailTemplateDAO
        IssueDAO
        JournalDAO
```

```
        JournalSettingsDAO
        LayoutAssignmentDAO
        LayoutEditorSubmissionDAO
        NotificationStatusDAO
        OAIDAO
        ProofAssignmentDAO
        ProofreaderSubmissionDAO
        PublishedArticleDAO
        RTDAO
        ReviewAssignmentDAO
        ReviewerSubmissionDAO
        RoleDAO
        ScheduledTaskDAO
        SectionDAO
        SectionEditorSubmissionDAO
        SectionEditorsDAO
        SessionDAO
        SiteDAO
        SubscriptionDAO
        SubscriptionTypeDAO
        SuppFileDAO
        TemporaryFileDAO
        UserDAO
        VersionDAO
DataObject
        Article
                AuthorSubmission
                CopyeditorSubmission
                LayoutEditorSubmission
                ProofreaderSubmission
                PublishedArticle
                ReviewerSubmission
                SectionEditorSubmission
                        EditorSubmission
        ArticleComment
        ArticleEmailLogEntry
        ArticleEventLogEntry
        ArticleFile
                ArticleGalley
                        ArticleHTMLGalley
                ArticleNote
                SuppFile
        Author
        BaseEmailTemplate
                EmailTemplate
                LocaleEmailTemplate
        Comment
        CopyAssignment
        Currency
        EditAssignment
        HelpToc
        HelpTopic
        HelpTopicSection
        Issue
        Journal
        LayoutAssignment
        Mail
                MailTemplate
                        ArticleMailTemplate
        ProofAssignment
        ReviewAssignment
        Role
        Section
```

```
        Session
        Site
        Subscription
        SubscriptionType
        TemporaryFile
        User
                ImportedUser
        Version
FileManager
        ArticleFileManager
        PublicFileManager
        TemporaryFileManager
Form
        ArticleGalleyForm
        AuthorSubmitForm
                AuthorSubmitStep1Form
                AuthorSubmitStep2Form
                AuthorSubmitStep3Form
                AuthorSubmitStep4Form
                AuthorSubmitStep5Form
        AuthorSubmitSuppFileForm
        ChangePasswordForm
        CommentForm
                CopyeditCommentForm
                EditorDecisionCommentForm
                LayoutCommentForm
                PeerReviewCommentForm
                ProofreadCommentForm
        CommentForm
                CopyeditCommentForm
                EditorDecisionCommentForm
                LayoutCommentForm
                PeerReviewCommentForm
                ProofreadCommentForm
        ContextForm
        EditCommentForm
        EmailTemplateForm
        InstallForm
        IssueForm
        JournalSetupForm
                JournalSetupStep1Form
                JournalSetupStep2Form
                JournalSetupStep3Form
                JournalSetupStep4Form
                JournalSetupStep5Form
        JournalSiteSettingsForm
        LanguageSettingsForm
        MetadataForm
        ProfileForm
        RegistrationForm
        SearchForm
        SectionForm
        SiteSettingsForm
        SubscriptionForm
        SubscriptionTypeForm
        SuppFileForm
        UpgradeForm
        UserManagementForm
        VersionForm
FormValidator
        FormValidatorArray
        FormValidatorCustom
        FormValidatorInSet
```

```
        FormValidatorLength
        FormValidatorRegExp
                FormValidatorAlphaNum
                FormValidatorEmail
Handler
        AboutHandler
        AdminHandler
                AdminFunctionsHandler

                AdminJournalHandler
                AdminLanguagesHandler
                AdminSettingsHandler
        ArticleHandler
                RTHandler
        AuthorHandler
                SubmissionCommentsHandler
                SubmitHandler
                TrackSubmissionHandler
        CommentHandler
        CopyeditorHandler
                SubmissionCommentsHandler
                SubmissionCopyeditHandler
        HelpHandler
        IndexHandler
        InformationHandler
        InstallHandler
        IssueHandler
        LayoutEditorHandler
                SubmissionCommentsHandler
                SubmissionLayoutHandler
        LoginHandler
        ManagerHandler
                EmailHandler
                FilesHandler
                ImportExportHandler
                JournalLanguagesHandler
                PeopleHandler
                SectionHandler
                SetupHandler
                SubscriptionHandler
        OAIHandler
        ProofreaderHandler
                SubmissionCommentsHandler
                SubmissionProofreadHandler
        RTAdminHandler
                RTContextHandler
                RTSearchHandler
                RTSetupHandler
                RTVersionHandler
        ReviewerHandler
                SubmissionCommentsHandler
                SubmissionReviewHandler
        SearchHandler
        SectionEditorHandler
                EditorHandler
                        IssueManagementHandler
                SubmissionCommentsHandler
                SubmissionEditHandler
        UserHandler
                EmailHandler
                ProfileHandler
                RegistrationHandler
Installer
```

```
        Install
        Upgrade
ItemIterator
        ArrayItemIterator
        DAOResultFactory
        DBRowIterator
        VirtualArrayIterator
OAI
        JournalOAI
OAIIdentifier
        OAIRecord
OAIMetadataFormat
        OAIMetadataFormat_DC
        OAIMetadataFormat_MARC
        OAIMetadataFormat_MARC21
        OAIMetadataFormat_RFC1807
Plugin
        ImportExportPlugin
                NativeImportExportPlugin
                SampleImportExportPlugin
                UserImportExportPlugin
RT
        JournalRT
RTAdmin
        JournalRTAdmin
ScheduledTask
        ReviewReminder
SearchFileParser
        SearchHelperParser
        SearchTextParser
                SearchHTMLParser
Smarty
        TemplateManager
XMLDAO
        HelpTocDAO
        HelpTopicDAO
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 an 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 `{translate key="[keyName]"}` or `{assign_translate var=[varName] key="[keyName"]}` directives (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` directory, 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 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, are used to 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. The `SectionForm` class from the example above (implemented in `classes/manager/form/SectionForm.inc.php`), is fairly typical. 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, it is 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

Database Support

The basic database functionality is provided by the ADODB library; 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
- `UserXMLParser.inc.php`: Handles XML import of user data for the "Import Users" feature

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.
- `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`. Schedule tasks must be enabled in the `configuration.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 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 `ItemIterator`s.

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 "`client_charset`" setting from the `[i18n]` section of the `config.inc.php` configuration file
- `baseUrl`: Base URL of the site, e.g. `http://www.mylibrary.com`
- `pageTitle`: Default name of locale key of page title; this should be replaced with a more appropriate setting in the template
- `indexUrl`: URL to the `index.php` file in the root directory of the OJS source tree; e.g. `http://www.mylibrary.com/index.php`
- `pageUrl`: URL to the current Page class; e.g., for the User page class, `http://www.mylibrary.com/index.php/user`
- `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)
- `requestPageUrl`: `pageUrl` with the requested operation appended, if applicable; e.g. `http://www.mylibrary.com/index.php/user/profile`
- `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
- `pageStyleSheet`: Name of custom stylesheet, if defined, for this journal
- `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

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, of the `locale.xml` file contains:
  ```
  <message key="my.locale.key">myVar equals "{$myVar}".</message>
  ```
  The resulting output will be:
  ```
  myVar equals "value".
  ```
  (Note that only direct variable replacements are allowed in locale files. You cannot call methods on objects or Smarty functions.)
- `assign_translate` (e.g. `{assign_translate var=myVar key="my.locale.key"}`): Similar to `{translate ...}`, except that the result is assigned to the specified Smarty variable rather than being displayed to the browser.
- `html_options_translate` (e.g. `{html_options_translate values=$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.

There are many examples of use of each of these functions, which provide good practical examples of use.

Plugins

A plugin registry has been introduced to OJS 2.0.1. Currently it is used to provide data import/export features via two plugins, `NativeImportExportPlugin` and `UserImportExportPlugin`, which provide article and issue import/export, and user import/export, respectively. (These plugins are documented elsewhere.)

Plugins are managed by the `PluginRegistry` class (implemented in `classes/plugins/PluginRegistry.inc.php`). All plugins must be subclasses of the `Plugin` class (implemented in `classes/plugins/Plugin.inc.php`).

Plugins are organized into categories; each category of plugin hooks into the OJS 2.x codebase at a different point, and as such can provide specific kinds of functions.

For example, the `importexport` category contains plugins that implement data import/export functionalities. The plugins reside in the `plugins/importexport` directory, each in its own subdirectory.

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.

Currently `importexport` is the only implemented category; in future releases additional categories will be made available, supporting (for example) additional search engines and more flexible metadata.