PKP Harvester2

Version 2.0

Technical Reference

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 PKP Harvester2

The PKP Harvester2 is an open-source metadata harvester and aggregator that has been developed by the Public Knowledge Project through its federally funded efforts to expand and improve access to research. Harvester2 has been designed with flexibility in mind and supports multiple harvesting protocols and metadata formats with an emphasis on performance and simplicity of use. In concert with the PKP software suite, including Open Journal Systems and Open Conference Systems, the goal of Harvester2 is to promote open access publishing and contribute to the public good on a global scale.

Version 2.x represents a complete rebuild and rewrite of the PKP Harvester 1.x, based on the platform pioneered by the Public Knowledge Project with Open Journal Systems 2.x.

User documentation for Harvester2 can be found on the Internet at `http://pkp.sfu.ca/harvester2/demo/index.php/index/help`; a demonstration site is available at `http://pkp.sfu.ca/harvester2/demo`.

About This Document

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

PKP Harvester2 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)
- **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 Harvester2 on platforms not listed above.

Design Overview

Harvester2 is designed to be a flexible tool for fetching, storing, indexing and searching data from a variety of different types of sources. Several parts of this process are abstracted using plugins to allow future extensions; for example, metadata schema are each implemented as a plugin and more can be added by simply dropping new plugins into the appropriate directory. Likewise, metadata harvesting protocols, such as the Open Archives Initiative metadata harvesting protocol, are also implemented as plugins.

The Harvester is designed around the concepts of Archives, Records, Entries, and Schemas, and Fields.

Each Record describes a single “entity” of some kind, such as a book, recording, or web page.

Each Archive is a repository of records. An archive may contain a set of records corresponding to a physical collection, such as a library.

A Schema is a standard for describing an entity, such as Dublin Core, MARC, or MODS. Each Schema is composed of a set of Fields, such as “Creator” and “Title”, that can be combined to comprehensively describe a record.

In Harvester2, each Record contains a number of Entries, each in a respective Field, that describe the entity that the Record corresponds to.

Harvester2 is designed to be a remote database of metadata, using the above concepts, that periodically communicates with the source from which the data is obtained. For example, if an organization is managing a journal using Open Journal Systems, a remote site can index and provide searching facilities to the journal (and, simultaneously, many other data sources) using Harvester2. Data is exchanged, for example, using the OAI metadata harvesting protocol; periodically Harvester2 will refresh the data from the journal source.

Introduction

The design of PKP Harvester2 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;
- **Model classes**, which implement PHP objects representing the system's various entities, such as Archives and Records;
- **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.

Coding Conventions

General

- Directories are named using the lowerCamelCase capitalization convention;
- Because Harvester2 supports 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 `AdminHandler` class is in the file `AdminHandler.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. `archives`, `records`) 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/harvester2_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 Site Administrator should be able to input unchecked HTML, and only in certain fields (such as the multiline fields in Administration). For example, when displaying an archive title, always use the following: `{$archive->getTitle()|escape}`
- Limited HTML support can be provided using the Smarty `strip_unsafe_html modifier`, e.g. `{$myVariable|strip_unsafe_html}`

File Structure

The following files are in the root directory of a typical Harvester2 installation:

File/Directory	*Description*
`cache`	Directory containing cached information
`classes`	Directory containing most of Harvester2's PHP code: Model classes, Data Access Objects (DAOs), core classes, etc
`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
`registry`	Directory containing various XML data required by the system: scheduled tasks, available locale names, default crosswalks, 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:

File/Directory	*Description*
	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 Harvester2 make use of the `PATH_INFO` variable. For example, examine the following (fictional) URL:

```
http://www.mylibrary.com/harvester2/index.php/browse/index/all
```

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, Harvester2 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, Harvester2 will generate URLs unlike those used as examples in this document. For example, the URL above would appear as:

```
http://www.mylibrary.com/harvester2/index.php?
  page=browse&op=index&path=all
```

Request Handling Example

Predictably, delegation of request handling occurs based on the request URL. A typical URL for browsing an archive is:

```
http://www.mylibrary.com/harvester2/index.php/browse/index/all
```

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/harvester2/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 (in this case, `browse`) is predefined; if others follow, they serve as parameters to the appropriate handler function.

The first 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/browse/index.php`; a brief glance at that file indicates that it simply defines a constant identifying the Page class name (in this case, `BrowseHandler`) and loads the PHP file defining that class.

The last fields, `index` and `all` in this case, now come into play. The first identifies the particular function of the Page class that will be called to handle the request. In the above example, this is the `index` method of the `BrowseHandler` class (defined in the `pages/browse/BrowseHandler.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 first field after `index.php`; for example, in the following URL:

```
http://www.mylibrary.com/index.php/browse/index/all
```

the name of the Page class is `BrowseHandler`. (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/browse/BrowseHandler.inc.php`.
- Determine which function is being called by examining the URL. This is the second field after `index.php`, or, in this case, `index`.
- Therefore, the handling code for this request is in the file `pages/user/UserHandler.inc.php`, in the function `profile`.
- Any remaining parts of the URL, in this case "`all`", are passed to this function as parameters.

Database Design

Overview

The major tables involved in storing metadata are represented in the diagram below; arrows indicate many-to-one foreign keys.

Unless otherwise noted, database tables are managed by PHP classes of the same name (except in the singular form). For example, the archives table is managed by the `ArchiveDAO` class; rows are represented using the `Archive` class.

The `archives` table maintains the list of archives known to the system, whether created by the administrator or submitted by users. In addition to a title, description, etc., this table associates each archive with a harvester plugin (such as the OAI harvester plugin).

Metadata is stored in Harvester2 in the `entries` table, regardless of the metadata format being used. Each row in the `entries` table represents a single metadata "item" – for example, a title or a date. This is the finest level of granularity of the particular metadata format being represented.

Each `entry` is associated with a particular row in the `raw_fields` table, and with a particular row in the `archives` table. Additionally, each `entry` can be further described or qualified via entries in the `entry_attributes` table.

Rows in the `entries` and `entry_attributes` tables are managed by the `RecordDAO` class.

`entries` are grouped into `records`, which are represented by the `records` table. Each record represents a single entity which is described by its various entries.

Here, each `record` is also associated with the schema plugin that is responsible for its display.

The `raw_fields` table maintains a list of all fields supported by the various schemas known to Harvester2, and associates each with the schema plugin it relates to (as identified in the `schema_plugins` table).

Note that the entries in the `schema_plugins` and `raw_fields` tables are constructed on demand, largely to provide an efficient identifier for relational mappings, and should be considered to exist only at the whim of the schema plugin code.

Miscellaneous Tables

The `archive_settings`, `plugin_settings`, and `site_settings` tables are used to store a variety of additional pieces of information at the archive, plugin, and site contexts. The specific usage of these tables is mostly dependent on the set of plugins being used.

The `email_templates` and `email_templates_data` tables store the localized body text and other information about system email templates.

The `search_keyword_list`, `search_object_keywords`, and `search_objects` tables provide a full-text index for harvested records.

The `versions` and `sessions` tables, respectively, store Harvester2 version information and login session identification information.

Class Reference

Class Hierarchy

All classes and subclasses of the major Harvester2 objects are listed below. Indentation indicates inheritance.

```
CacheManager
CommandLineTool
      dbXMLtoSQL
      genTestLocale
      harvest
      rebuildSearchIndex
      upgradeTool
Config
ConfigParser
Core
DAO
      ArchiveDAO
      ArchiveSettingsDAO
      CrosswalkDAO
      EmailTemplateDAO
      FieldDAO
      PluginSettingsDAO
      RecordDAO
      SchemaDAO
      SearchDAO
      SessionDAO
      SiteSettingsDAO
      VersionDAO
DAORegistry
DBConnection
DBDataXMLParser
DBResultRange
DataObject
      Archive
      BaseEmailTemplate
            EmailTemplate
      Crosswalk
      Field
      HelpToc
      HelpTopic
      HelpTopicSection
      Mail
            MailTemplate
      Record
      Schema
      Site
      Version
FileWrapper
```

```
      FTPFileWrapper
      HTTPFileWrapper
            HTTPSFileWrapper
Form
      ArchiveForm
      CrosswalkForm
      InstallForm
      SiteSettingsForm
      UpgradeForm
FormError
FormValidator
      FormValidatorArray
      FormValidatorCustom
      FormValidatorInSet
      FormValidatorLength
      FormValidatorRegExp
            FormValidatorAlphaNum
            FormValidatorEmail
GenericCache
      FileCache
      MemcacheCache
Handler
      AboutHandler
      AddHandler
      AdminHandler
            AdminArchiveHandler
            AdminCrosswalkHandler
            AdminFunctionsHandler
            AdminLanguagesHandler
            AdminSettingsHandler
      BrowseHandler
      HelpHandler
      IndexHandler
      InstallHandler
      LoginHandler
      RecordHandler
      SearchHandler
Harvester
      OAIHarvester
Help
HookRegistry
Installer
      Install
      Upgrade
ItemIterator
      ArrayItemIterator
      DAOResultFactory
      DBRowIterator
      VirtualArrayIterator
Locale
Plugin
      HarvesterPlugin
            JunkHarvesterPlugin
            OAIHarvesterPlugin
```

```
      PostprocessorPlugin
            TestPostprocessorPlugin
      PreprocessorPlugin
            TestPreprocessorPlugin
      SchemaPlugin
            DublinCorePlugin
            MarcPlugin
            ModsPlugin
PluginRegistry
Registry
Request
SMTPMailer
SchemaMap
Search
SearchIndex
SessionManager
String
TemplateManager
Validation
VersionCheck
XMLCustomWriter
XMLDAO
      HelpTocDAO
      HelpTopicDAO
XMLNode
XMLParser
XMLParserHandler
      DublinCoreXMLHandler
      MarcXMLHandler
      ModsXMLHandler
      OAIXMLHandler
      SchemaMapHandler
      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/browse/BrowseHandler.inc.php` contains the code implementing the class `BrowseHandler`, which handles requests such as `http://www.mylibrary.com/harvester2/browse/index`. This is a fairly typical Page class responsible for allowing the user to choose and browse records in an archive.

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.)

Model Classes

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

Methods provided by Model classes are largely get/set methods to retrieve and store information, such as the `getTitle()` and `setTitle($title)` methods of the `Archive` 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 `Archive` class (`classes/archive/Archive.inc.php`) has an associated DAO called `ArchiveDAO` (`classes/archive/ArchiveDAO.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 archive DAO:

```
$archiveDao = &DAORegistry::getDAO('ArchiveDAO');
```

Then, to use it to retrieve an archive with the ID stored in `$archiveId`:

```
$archive = &$archiveDao->getArchive($archiveId);
```

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/SMTPMailer.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 (either via PHP's `mail()` function or via the custom `SMTPMailer` class). It is extended by the class `MailTemplate` to add support for template-based messages.

Internationalization

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.

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/admin/form/ArchiveForm.inc.php`, which is used by a Site Administrator to modify an Archive, 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 Archive; an `Archive` object or archive 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 `ArchiveForm` class from the example above (implemented in `classes/admin/form/ArchiveForm.inc.php`).

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 `ArchiveForm` class closely corresponds to the layout of the `archives` database table.

Configuration

Most of Harvester2's settings are stored in the database in the

archive_settings, site_settings, and plugin_settings tables, 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 Harvester2 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 Harvester2 to link into itself are built using the Request::url function; likewise, all redirects into Harvester2 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

Security

Harvester2 uses a simple security model. The only authenticated user is the site administrator, who can choose a username and password. All other users are unauthenticated and have the same level of access.

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 `ItemIterator`s.

Each iterator represents a single "page" of results. For example, when fetching a list of records from `RecordDAO`, 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 Harvester2 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 `NEW_ARCHIVE_NOTIFY`. 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 `NEW_ARCHIVE_NOTIFY` email, which is sent to non-Administrator submitters as an acknowledgment when they enter a new archive. (This snippet assumes that `$archiveId` is set to the new archive's ID.)

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

// Retrieve the mail template by name.
$mail = &new MailTemplate('NEW_ARCHIVE_NOTIFY');

if ($mail->isEnabled()) {
   // Get the site object and assign the contact person as the recipient
   $site =& Request::getSite();
   $mail->addRecipient($site->getContactEmail(), $site->getContactName());

   // 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(
      'archiveTitle' => 'This is the title of the archive',
      'siteTitle' => $site->getTitle(),
      'loginUrl' => Request::url('admin', 'manage', $archiveId)
   ));

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

Database Interaction with DAOs

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

```
// Fetch the archive object using the archive DAO.
$archiveDao = &DAORegistry::getDAO('ArchiveDAO');
$archive = &$archiveDao->getArchive($archiveId);

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

// Update the database with the modified information.
$archiveDao->updateArchive($archive);
```

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

```
// Fetch the archive object using the archive DAO.
$archiveDao = &DAORegistry::getDAO('ArchiveDAO');
$archive = &$archiveDao->getArchive($archiveId);

// Delete the archive from the database.
$archiveDao->deleteArchive($archive);
```

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

```
// Delete the archive using the archive DAO.
$archiveDao = &DAORegistry::getDAO('ArchiveDAO');
$archiveDao->deleteArchiveById($archiveId);
```

Generally speaking, the DAOs are responsible for deleting dependent database entries. For example, deleting an archive should delete that archive's records and entries from the database. Note that this is accomplished in PHP code rather than using 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 archive titles) or through locale-specific translations using a custom Smarty function.

You should be familiar with Smarty templates before working with Harvester2 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
- `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`: Site title from Site Configuration
- `pagePath`: Path of the requested page and operation, if applicable, prepended with a slash; e.g. `/browse/index`
- `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`
- `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.
- `items_per_page`: The maximum number of items to display per page of a paged list.

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::Hook::Name::Here"}`) 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 Harvester2.

Plugins

The PKP Harvester2 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 Harvester2. When placed in the appropriate directory within the 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 `schemas` category (which implement functions specific to

a particular metadata schema) are loaded whenever a schema-specific function is used (such as when a record is viewed). These plugins must implement certain methods which are used for delegation of control between the plugin and Harvester2.

Hooks are used by plugins as a notification tool and to override behaviors built into Harvester2. At many points in the execution of Harvester2 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 Harvester2 around the point at which that hook was called.

While most of the plugin categories built into Harvester2 relate to specific aspects of the system, such as harvester protocols and schemas, 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 Harvester2. Most hooks are generally intended for use with plugins in this category (although any plugin category can register against any hook, with the only limitation being that the category must be loaded in order to be effective).

Objects & Classes

Plugins in Harvester2 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`)
`harvesters`	`HarvesterPlugin` (`classes/plugins/HarvesterPlugin.inc.php`)
`preprocessors`	`PreprocessorPlugin` (`classes/plugins/PreprocessorPlugin.inc.php`)
`postprocessors`	`PostprocessorPlugin` (`classes/plugins/PostprocessorPlugin.inc.php`)
`schemas`	`SchemaPlugin` (`classes/plugins/SchemaPlugin.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`).

Registration Function

Whenever Harvester2 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.

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

As described above, plugins will usually register against hooks in the plugin's `register(...)` function. When registering against a hook, the plugin must specify a callback function; when a hook call is encountered the hook registry will call, in the order in which they were registered, all callbacks registered against the hook. This process can be interrupted by any particular callback by returning a `true` from the callback function.

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

```
HookRegistry::register(
      'Templates::Hook::Name::Here',
      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.

If another plugin (or even the same plugin) was registered again against the same hook, and the first registrant returned `true` from the hook callback, second callback would not be called.

Plugin Management

In the plugin class, there are three functions that provide metadata about the plugin: `getName()`, `getDisplayName()`, and `getDescription()`. These are part of a plugin management interface that is available to the Administrator.

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 Administrator 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 Archive 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 Harvester2's schema-management features. This function is called on Harvester2 install or upgrade.
- **Data Management:** By overriding `getInstallDataFile()` and placing the named data file in the plugin directory, `generic` plugins can make use of Harvester2's data installation feature. This function is called on Harvester2 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 Harvester2 as of release 2.0. 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, &$sourceFile`	Called by Harvester2's main `index.php` script after the page (`&$page`), operation (`&$op`), and handler code file (`&$sourceFile`) names have been determined, but before `$sourceFile` is loaded. Can be used to intercept browser requests for handling by the plugin. Returning `true` from the callback will prevent Harvester2 from loading the handler stub in `$sourceFile`.
`ArchiveForm::ArchiveForm`	`&$archiveForm, $harvesterPluginName`	Called at the end of the Archive form's constructor; the archive form object and current harvester plugin name are passed in as parameters.
`ArchiveForm::display`	`&$archiveForm, &$templateMgr, $harvesterPluginName`	Called just before the Archive form is displayed. The archive form object, template manager (extended Smarty object), and harvester plugin name are passed as parameters.
`ArchiveForm::initData`	`&$archiveForm, &$archive, $harvesterPluginName`	Called after the archive form's data is initialized for the given archive with the given harvester plugin name, but before the form is overridden with any posted values the user may have already supplied.
`ArchiveForm::getParameterNames`	`&$archiveForm, &$parameterNames, &$harvesterPluginName`	Called before the archive form returns a list of parameter names. This hook can be used to extend the list of parameters included on the archive form.

Name	***Parameters***	***Description***
`archiveForm::execute`	`&$archiveForm, &$archive, $harvesterPluginName`	Called after the archive form has updated or created the current archive in response to a user's submission of the form but before, if applicable, Harvester2 sends an email to the administrator notifying them of a new archive submission.
`CrosswalkForm::CrosswalkForm`	`&$crosswalkForm`	Called at the end of the Crosswalk form's constructor.
`CrosswalkForm::execute`	`&$crosswalkForm, &$crosswalk`	Called after the crosswalk form has updated a crosswalk in response to an administrator request.
`ArchiveDAO::_returnArchiveFromRow`	`&$archive, &$row`	Called after an Archive object is created from the given database row, before it is returned to the caller.
`[something]DAO::Constructor`	`&$dao, &$dataSource`	Called when a DAO is constructed with the given data source. To prevent the default constructor behavior from occurring, the hook registrant should return `true` from its callback function. 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

Name	***Parameters***	***Description***
		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, `&$value` should receive the retrieve call's intended result and the hook should return `true`. This hook should only be used with PHP >= 4.3.0.
`[Anything]DAO::[Any function calling DAO::retrieveRange]`	`&$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, `&$value` should receive the retrieve call's intended result and the hook should return `true`. This hook should only be used with PHP >= 4.3.0.
`Locale::_cacheMiss`	`&$id, &$locale, &$value`	Called when a locale key cannot be found in the current locale cache. This can be used to extend

Name	***Parameters***	***Description***
		the default set of locale data with additional keys. To override the default behavior, the hook registrant should specify a return value in `&$value` and return `true` from the callback function.
`Installer::Installer`	`&$installer, &$descriptor, &$params`	Called in the constructor of the Installer class. To prevent the default behavior of the Installer's constructor, the callback registrant should return `true` from its callback function.
`Installer::preInstall`	`&$installer, &$result`	Called during the Installer's pre-installation phase. The hook registrant has the opportunity to alter the return value (`&$result`)of the `preInstall` function.
`Installer::postInstall`	`&$installer, &$result`	Called during the Installer's post-installation phase. The hook registrant has the opportunity to alter the return value (`&$result`)of the `postInstall` function.
`Installer::parseInstaller`	`&$installer, &$result`	Called after the Installer parses the installation script. The hook registrant has the opportunity to alter the return value (`&$result`) of the `parseInstaller` function.
`Installer::executeInstaller`	`&$installer, &$result`	Called after the Installer executes the installation script. The hook registrant has the opportunity to alter the return value (`&$result`) of the `executeInstaller` function.
`Installer::updateVersion`	`&$installer, &$result`	Called after the Installer updates the version of the installation. The hook registrant has the opportunity to alter the return value (`&$result`) of the `updateVersion` function.
`SchemaPlugin::indexRecord`	`&$archive, &$record, &$field, &$value, &$attributes`	Called before a schema plugin updates the indexing of an entry. To prevent the schema plugin from indexing the entry (i.e. if the hook registrant indexes it itself), the hook registrant should return `true` from the callback function.

Name	*Parameters*	*Description*
`VersionDAO::_returnVersionFromRow`	`&$version, &$row`	Called after a Version object is created from the given database row, before it is returned to the caller.
`TemplateManager::display`	`&$templateMgr, &$template, &$sendContentType, &$charset`	Called before the given template manager (extended Smarty object) displays the given template. To prevent the template from being displayed, the hook registrant should return `true` from the callback function.
`Request::redirect`	`&$url`	Called before `Request::redirect` performs a redirect to `&$url`. Returning true will prevent Harvester2 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`	`&$userAgent`	Called the first time `Request::getUserAgent` is called after the user agent has been determined but before returning it to the caller. This value is used for all subsequent calls.
`Request::getRequestedJournalPath`	`&$journal`	Called the first time

Name	*Parameters*	*Description*
		`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.
`FieldDAO::_returnFieldFromRow`	`&$field, &$row`	Called after a Field object is created from the given database row, before it is returned to the caller.
`CrosswalkDAO::_returnCrosswalkFromRow`	`&$field, &$row`	Called after a Crosswalk object is created from the given database row, before it is returned to the caller.
`RecordDAO::_returnRecordFromRow`	`&$record, &$row`	Called after a Record object is created from the given database row, before it is returned to the caller.
`SchemaDAO::_returnSchemaFromRow`	`&$schema, &$row`	Called after a Schema object is created from the given database row, before it is returned to the caller.
`Harvester::insertEntry`	`&$archive, &$record, &$field, &$value, &$attributes`	Called before Harvester2 inserts an entry in the given field of the given record of the given archive with the given attributes. To prevent the default behavior from occurring, the hook registrant should return `true` from its callback function.
`Mail::send`	`&$mail, &$recipients, &$subject, &$mailBody, &$headers, &$additionalParameters`	Called before Harvester2 sends the email message with the given parameters. To prevent this from occurring (i.e. if the hook callback takes care of sending the message itself), it should return `true` from its callback function.
`EmailTemplateDAO::_returnEmailTemplateFromRow`	`&$emailTemplate, &$row`	Called after an email template object is created from the given database row, before it is returned to the caller.
`RTDAO::_returnVersionFromRow`	`&$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

Name	***Parameters***	***Description***
		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.
`Template::Admin::Archives::displayHarvesterForm`	`&$params, &$smarty, &$output`	Called after the built-in parts of the archive form have been displayed; this hook can be used to extend the form with additional fields. `$params['plugin']` contains the name of the harvester plugin.
`Template::Admin::Index::SiteManagement`	`&$params, &$smarty, &$output`	Called at the end of the items list in the Site Management bulleted list.
`Template::Admin::Index::AdminFunctions`	`&$params, &$smarty, &$output`	Called at the end of the items list in the Admin Functions bulleted list.
`Template::Browse::ArchiveInfo::DisplayExtendedArchiveInfo`	`&$params, &$smarty, &$output`	Called after the built-in parts of the archive information template have been displayed; this hook can be used to add information to this page. `$params['archive']` contains the archive object in question.

Translating Harvester2

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 Harvester2 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 Harvester2.
- `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`, and `dbscripts/xml/data/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 Harvester2.

Obtaining More Information

For more information, see the PKP web site at `http://pkp.sfu.ca`. There is a Harvester2 support forum available at `http://pkp.sfu.ca/support/forum`; this is the preferred method of contacting the Harvester2 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`.