Chapter 13 - Islandora Community (Developing Islandora)

Islandora software is developed and maintained by a dedicated group of developers, systems administrators, and librarians. As an open source project, Islandora is constantly undergoing improvements in its architecture and adding new features. We welcome feedback from the community and are always seeking new contributors and collaborators.

Here are the spaces that make up Islandora's community:

Git Hub	https://github.com/organizations/Islandora	All of our code repositories are available at this URL. To submit code to Islandora, send a pull request. Our release manager tests and merges code into the main Islandora repository.
Developers Meetings	Skype	Our developers meet weekly to review open tickets and review the status of the code. If you are interested in attending a developer meeting, please submit a request to community@islandora.ca or on the developers Google Group mailing list.
JIRA	https://jira.duraspace.org/browse/ISLANDORA	Our JIRA is the place to submit bugs and feature requests relating to Islandora. It is easy to make an account so you can submit tickets.
Wiki	https://wiki.duraspace.org/display/ISLANDORA/Islandora	This documentation is maintained on a wiki. If you have an account, you can edit the wiki.
Islandora.ca	http://islandora.ca/	At Islandora.ca, you will find our most current news, as well as information about Islandora events and links to our community spaces.

Google Groups (Developers)	http://groups.google.com/group/islandora-dev	A place for developers to answer questions and keep tabs on code development
Google Groups (Users)	http://groups.google.com/group/islandora	A place for users and implementers to ask questions and get support
Facebook	http://www.facebook.com/Islandora	A facebook page where we post updates and events
Twitter	http://twitter.com/Islandora	Our Twitter feed. The first place for news and updates about the Islandora community.

Coding Standards for Islandora

Anyone developing for the Islandora module is to follow the Drupal coding standards (http://drupal.org/coding-standards). This will be tested at commit time by the Coder Module (http://drupal.org/project/coder). In order to make things easier, try to test your code with the coder module before committing code to the islandora GIT. There are also some great pages on the Drupal Wiki (http://drupal.org/node/147789) about how to setup your favorite Integrated Development Environment (IDE) to use the Drupal standards.

Here are some examples:

- Netbeans (http://drupal.org/node/1019816)
- VIM (http://drupal.org/node/29325)

Islandora Module File Structure and Naming

Briefly, filenames should contain lowercase characters and PHP files (except templates) should use underscores to separate words in the filename. For a more detailed outline and explanation, please see: http://github.com/Islandora/islandora/wiki/Islandora-Module-File-Structure-and-Naming

Documentation Standards in Islandora Development

This page covers the standards for documentation in Islandora code. Most of this page is taken directly from the Drupal Doxygen and Comment Formatting Standards (http://drupal.org/node/1354). There have been some modifications specifically for Islandora; when in doubt, refer to the Drupal standards.

There are two types of comments: in-line and headers. In-line comments are comments that are within functions. In Islandora, documentation headers on functions, classes, constants, etc. are specially-formatted comments used to build the API and developer documentation. The system for extracting documentation from the header comments uses the Doxygen generation system, and since the documentation is extracted directly from the sources, it is much easier to keep the documentation consistent with the source code.

There is an excellent Doxygen manual at http://www.stack.nl/~dimitri/doxygen/manual.html . This page describes the Islandora implementation of Doxygen, and our standards for both in-line and header comments.

General documentation standards

These standards apply to both in-line and header comments:

- All documentation and comments should form proper sentences and use proper grammar and punctuation.
- Sentences should be separated by single spaces.
- Comments and variable names should be in English, and use US English spelling (e.g., "color" not "colour").
- All caps are used in comments only when referencing constants, for example TRUE.

- Comments should be word-wrapped if the line length would exceed 80 characters (i. e., go past the 80th column). They should be as long as possible within the 80-character limit.

In-line comment standards

Non-header or in-line comments are strongly encouraged. A general rule of thumb is that if you look at a section of code and think "Wow, I don't want to try to describe that", you need to comment it before you forget how it works. Comments should be on a separate line immediately before the code line or block they reference. For example:

```
// Unselect all other contact categories.
db_query('UPDATE{contact}SET selected = 0');
```

If each line of a list needs a separate comment, the comments may be given on the same line and may be formatted to a uniform indent for readability.

C style comments (/* */) and standard C++ comments (//) are both fine, though the former is discouraged within functions (even for multiple lines, repeat the // single-line comment). Use of Perl/shell style comments (#) is discouraged.

General header documentation syntax

To document a block of code, such as a file, function, class, method, constant, etc., the syntax we use is:

```
/*\*
\* Documentation here.
\*/
```

Doxygen will parse any comments located in such a block.

Our style is to use as few Doxygen-specific commands as possible, so as to keep the source legible. Any mentions of functions, classes, file names, topics, etc. within the documentation will automatically link to the referenced code, so typically no markup need be introduced to produce links.

Doxygen directives - general notes

```
/*\*
\* Summary here; one sentence on one line (should not, but can
exceed 80 chars).
\*
\* A more detailed description goes here.
\*
\* A blank line forms a paragraph. There should be no trailing white-space
\* anywhere.
\*
\* @param $first
\*   "@param" is a Doxygen directive to describe a function parameter. Like some *
other directives, it takes a term/summary on the same line and a
\*   description (this text) indented by 2 spaces on the next line. All
\*   descriptive text should wrap at 80 chars, without going over.
\*   Newlines are NOT supported within directives; if a newline would be before
\*   this text, it would be appended to the general description above.
\* @param $second
\*   There should be no newline between multiple directives (of the same type).
\* @param $third
\*   (optional) TRUE if Third should be done. Defaults to FALSE.
\*   Only optional parameters are explicitly stated as such. The description
\*   should clarify the default value if omitted.
\*
\* @return *   "@return" is a different Doxygen directive to describe the return value
of
\*   a function, if there is any.
\*/
function mymodule_foo($first, $second, $third = FALSE) {
}
```

Lists

```
\* @param $variables
\*   An associative array containing:
\*   - tags: An array of labels for the controls in the pager:
\*     - first: A string to use for the first pager element.
\*     - last: A string to use for the last pager element.
\*   - element: (optional) Integer to distinguish between multiple pagers on one
\*     page. Defaults to 0 (zero). *   - style: Integer for the style, one of the
following constants:
\*     - PAGER_FULL: (default) Full pager.
\*     - PAGER_MINI: Mini pager.
\*   Any further description - still belonging to the same param, but not part
```

```
\*   of the list.
\*
\* This no longer belongs to the param.
```

Lists can appear anywhere in Doxygen. The documentation parser requires you to follow a strict syntax to make them appear correctly in the parsed HTML output:

- A hyphen is used to indicate the list bullet. The hyphen is aligned with (uses the same indentation level as) the paragraph before it, with no newline before or after the list.
- No newlines between list items in the same list.
- Each list item starts with the key, followed by a colon, followed by a space, followed by the key description. The key description starts with a capital letter and ends with a period.
- If the list has no keys, start each list item with a capital letter and end with a period.
- The keys should not be put in quotes unless they contain colons (which is unlikely).
- If a list element is optional or default, indicate (optional) or (default) after the colon and before the key description.
- If a list item exceeds 80 chars, it needs to wrap, and the following lines need to be aligned with the key (indented by 2 more spaces).
- For text after the list that needs to be in the same block, use the same alignment /indentation as the initial text.
- Again: within a Doxygen directive, or within a list, blank lines are NOT supported.
- Lists can appear within lists, and the same rules apply recursively.

See Also sections

```
/*\*
\* (rest of function/file/etc. header)
\*
\* @see foo_bar()
\* @see ajax.inc
\* @see MyModuleClass
\* @see MyClass::myMethod()
\* @see groupname
\* @see [http://drupal.org/node/1354]
\*/
```

The @see directive may be used to link to (existing) functions, files, classes, methods, constants, groups/topics, URLs, etc. @see directives should always be placed on their own line, and generally at the bottom of the documentation header. Use the same format in @see that you would for automatic links (see below).

Links

```
/*\*
\* (rest of function/file/etc. header)
\*
\* See also @link group_name Link text @endlink
\*/
```

The @link directive may be used to output a HTML link in line with the text.

You can also use this to make a link to a particular URL (but it is only necessary if you want the link text to be something other than the URL itself). Example:

```
\* @link [http://example.com] Link text goes here @endlink
```

Automatic links

```
\* This function invokes hook_foo() on all implementing modules.
\* It calls MyClass::methodName(), which includes foo.module as
\* a side effect.
```

Any function, file, constant, class, etc. in the documentation will automatically be linked to the page where that item is documented (assuming that item has a doxygen header). For functions and methods, you must include () after the function/method name to get the link. For files, just put the file name in (not the path to the file).

Code samples

```
\* Example usage:
\* @code
```

```
\*   mymodule_print('Hello World\!');
\* @endcode
\* Text to immediately follow the code block.
```

Code examples can be embedded in the Doxygen documentation using @code and @endcode directives. Any code in between will be output preformatted.

Todos

```
/*\*
\* @todo Remove this in D8.
\* @todo Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam
\*   nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed
\*   diam voluptua.
\*/
```

To document known issues and development tasks in code, @todo statements may be used. Each @todo should form an atomic task. They should wrap at 80 chars, if required. Additionally, any following lines should be indented by 2 spaces, to clarify where the @todo starts and ends.

```
// @todo Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam
//   nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat,
//   sed diam voluptua.
// We explicitly delete all comments.
comment_delete($cid);
```

@todo statements in inline comments follow the same rules as above, especially regarding indentation of subsequent comments.

Documenting files

Each file should start with a comment describing what the file does. For example:

```
/*\*
\* @file
\* The theme system, which controls the output of Drupal.
\*
```

```
\* The theme system allows for nearly all output of the Drupal system to be
\* customized by user themes.
\*/
```

The line immediately following the @file directive is a **summary** that will be shown in the list of all file in the generated documentation. If the line begins with a verb, that verb should be in present tense, e.g., "Handles file uploads." Further description may follow after a blank line.

Documenting functions and methods

All functions and methods, whether meant to be private or public, should be documented. A function documentation block should immediately precede the declaration of the function itself. For example:

```
/*\*
\* Verifies the syntax of the given e-mail address.
\*
\* Empty e-mail addresses are allowed. See RFC 2822 for details.
\*
\* @param $mail
\*   A string containing an email address.
\*
\* @return
\*   TRUE if the address is in a valid format, and FALSE if it isn't.
\*/
function valid_email_address($mail) {
```

After the long description, each parameter should be listed with a @param directive, with a description indented by two extra spaces on the following line or lines. If there are no parameters, omit the @param section entirely. Do not include any blank lines in the @param section.

After all the parameters, a @return directive should be used to document the return value if there is one. There should be a blank line between the @param section and @return directive. If there is no return value, omit the @return directive entirely.

Functions that are easily described in one line may use the function summary only, for example:

```
/*\*
\* Converts an associative array to an anonymous object.
\*/
function mymodule_array2object($array) {
```

If the abbreviated syntax is used, the parameters and return value must be described within the one-line summary.

If the data type of a parameter or return value is not obvious or expected to be of a special class or interface, it is recommended to specify the data type in the @param or @return directive:

```
/*\*
\* Executes an arbitrary query string against the active database.
\*
\* Do not use this function for INSERT, UPDATE, or DELETE queries. Those should
\* be handled via the appropriate query builder factory. Use this function for
\* SELECT queries that do not require a query builder.
\*
\* @param object $query
\*   The prepared statement query to run. Although it will accept both named and
\*   unnamed placeholders, named placeholders are strongly preferred as they are
\*   more self-documenting.
\* @param array $args
\*   An array of values to substitute into the query. If the query uses named
\*   placeholders, this is an associative array in any order. If the query uses
\*   unnamed placeholders \(?), this is an indexed array and the order must match
\*   the order of placeholders in the query string.
\* @param array $options
\*   An array of options to control how the query operates.
\*
\* @return DatabaseStatementInterface
\*   A prepared statement object, already executed.
\*
\* @see DatabaseConnection::defaultOptions()
\*/
function db_query($query, array $args = array(), array $options = array()) {
// ...
}
```

Primitive data types, such as int or string, are not specified. It is recommended to specify classes and interfaces. If the parameter or return value is an array, or (anonymous/generic) object, you can specify the type if it would add clarity to the documentation. If for any reason, a primitive data type needs to be specified, use the lower-case data type name, e.g. "array". Also, make sure to use the most general class/interface possible. E.g., document the return value of db_select() to be a SelectQueryInterface, not a particular class that implements SelectQuery.

Documenting classes and interfaces

Each class and interface should have a doxygen documentation block, and each member variable, constant, and function/method within the class or interface should also have its own documentation block. Example:

```
/*\*
\* Represents a prepared statement.
\*/
interfaceDatabaseStatementInterfaceextendsTraversable{

/*\*
\* Executes a prepared statement.
\*
\* @param array $args
\*   Array of values to substitute into the query.
\* @param array $options
\*   Array of options for this query.
\*
\* @return
\*   TRUE on success, FALSE on failure.
\*/
public functionexecute($args= array(),$options= array());
}

/*\*
\* Represents a prepared statement.
\*
\* Default implementation of DatabaseStatementInterface.
\*/
classDatabaseStatementBaseextendsPDOStatementimplementsDatabaseStatementInterface{

/*\*
```

```
\* The database connection object for this statement DatabaseConnection.
\*
\* @var DatabaseConnection
\*/
public$dbh;

/*\*
\* Constructs a DatabaseStatementBase object.
\*
\* @param DatabaseConnection $dbh
\*   Database connection object.
\*/
protected function__construct($dbh) {
// Function body here.
}

/*\*
\* Implements DatabaseStatementInterface::execute().
\*
\* Optional explanation of the specifics of this implementation goes here.
\*/
public functionexecute($args= array(),$options= array()) {
// Function body here.
}

/*\*
\* Returns the foo information.
\*
\* @return object
\*   The foo information as an object.
\*
\* @throws MyFooUndefinedException
\*/
public functiongetFoo() {
// Function body here.
}

/*\*
\* Overrides PDOStatement::fetchAssoc().
\*
\* Optional explanation of the specifics of this override goes here.
\*/
public functionfetchAssoc() {
// Call PDOStatement::fetch to fetch the row.
return$this->fetch(PDO::FETCH_ASSOC);
}
}
```

Notes:

- Leave a blank line between class declaration and first docblock.
- Use a 3rd person verb to begin the description of a class, interface, or method (e.g. Represents not Represent).
- For a member variable, use @var to tell what data type the variable is.
- Use @throws if your method can throw an exception, followed by the name of the exception class. If the exception class is not specific enough to explain why the exception will be thrown, you should probably define a new exception class.
- Make sure when documenting function and method return values, as well as member variable types, to use the most general class/interface possible. E.g., document the return value of db_select() to be a SelectQueryInterface, not a particular class that implements SelectQuery.

Documenting Islandora interaction with Fedora

Whenever we interact with Fedora in the comments using either:

- @throws
- @return

We need to document what is returned on error and what state the Fedora is left in, so that the error can be reported through Drupal instead of failing silently and whoever is calling the API can make a choice about what to do next.

Generating Islandora documentation with Doxygen

In order to generate the documentation you need the doxygen package installed.

Download the Doxyfile to your modules/fedora_repository folder and run the command:

doxygen Doxyfile

after the command finishes running the documentation can be found in *documentation/html /index.html*

Within the Doxyfile if you change the line:

EXTRACT_ALL = YES

to

EXTRACT_ALL = NO

Then doxygen will only extract information from documented files, instead of trying to extract information from all files.