Jirafs Documentation

Release 1.13.0

Adam Coddington

March 23, 2016

Contents

Pronounced like 'giraffes', but totally unrelated to wildlife, this library lets you stay out of JIRA as much as possible by letting you edit your JIRA issues as a collection of text files using an interface inspired by `git` and `hg`.

CHAPTER 1

Getting Started

1.1 Installation

It is recommended that you install the program using `pip` while in a Python 3 virtualenv; you can install using `pip` by running:

```
pip install jirafs
```

After Jirafs successfully installs, you'll have access to the `jirafs` command that you can use for interacting with JIRA.

1.2 Working with a JIRA issue

First, you'll need to "clone" the issue you want to work with using Jirafs by running the following (replacing `http://my.jira.server/browse/MYISSUE-1024` with the issue url you are concerned about):

```
jirafs clone http://my.jira.server/browse/MYISSUE-1024
```

The first time you run this command, Jirafs will ask you for a series of details that it will use for communicating with JIRA; don't worry: although all of this information will be stored in a plaintext file at `~/.jirafs_config`, Jirafs will not store your password unless you give it permission to do so.

Once the command runs successfully, it will have created a new folder named after the issue you've cloned, and inside that folder it will place a series of text files representing the issue's contents in JIRA as well as copies of all attachments currently attached to the issue in JIRA.

The following text files are created:

- `fields.jira`: This file will show all currently-set field values for this JIRA issue (except fields written to their own files; see `description.jira` below). You **can** change field values here by editing the field values in the file. See *Editing Issue Fields* for more information.
- `description.jira`: This file will show the issue's current description. You **can** change the issue's description by editing the contents of this file.
- `links.jira`: This file lists all of the links associated with this JIRA issue. You can add new links (or remove links) by adding or removing bulleted items from this list; see *Adding, Removing or Changing Links* for more information.
- `comments.read_only.jira`: This file shows all comments currently posted to this issue. Note that you **cannot** edit the comments in this file.

- `new_comment.jira`: This file starts out empty, but if you would like to add a new comment, you **can** create one by entering text into this file.

In order to update any of the above data or upload an asset, either make the change to a field in `fields.jira`, edit the issue's description in `description.jira`, write a comment into `new_comment.jira`, or copy a new asset into this folder, then run:

```
jirafs status
```

to see both what changes you've marked as ready for being submitted to JIRA as well as which changes you have made, but not yet committed.

Note: Unlike when working with a git repository, you do not need to 'stage' files using a command analogous to git's "add" command when working with a JIRA issue using Jirafs. All uncommitted files will automatically be included in any commit made. This behavior will likely be surprising only to users who have not worked with mercurial (hg).

Once you're satisfied with the changes that are about to be submitted to JIRA, run:

```
jirafs submit
```

Note: `jirafs submit` really just runs `jirafs commit` followed by `jirafs push` (which itself runs `jirafs pull` to get your local copy up-to-date with what it saw in JIRA), so although `jirafs submit` is probably the path you want to take, feel free to use the lower-level more-git-like commands if you want.

Please consider this to be just a simple overview – there are a variety of other commands you can run to have finer-grained control over how the issue folder is synchronized with JIRA; see Common Commands for more details.

Note: If you are a VIM user, there is a *VIM Plugin* available that provides syntax highlighting for JIRA/Confluence's wikimarkup.

1.3 Editing Issue Fields

In most cases, you can simply edit the field's contents directly – just make sure to indent the field contents by four spaces.

For text fields, editing field contents is as simple as typing-in a new value, but many issue fields are are JSON dictionaries or lists that require you to edit the data in a more-structured way. If the data you enter is not valid JSON, when `push`-ing up changes, you will receive an error, but don't worry – if you encounter such an error, edit the contents to be valid JSON, `commit`, and `push` again. You may need to consult with JIRA's documentation to develop an understanding of how to change these values.

Note: You don't always need to enter values for every field in a JSON dictionary; in some cases, JIRA will infer the missing information for you.

1.4 Adding, Removing or Changing Links

Each line of `links.jira` starts with a bullet (`*`), and although links to other issues (in JIRA terminology – "issue links") and links to arbitrary URLs ("remote links") appear similar, they have slightly different formats.

1.4.1 Issue Links

You can link other issues to your JIRA issue by adding bulleted lines in the following format:

```
* LINK TYPE: TICKET NUMBER
```

So, if there is an issue relationship named "blocks", and your JIRA issue is blocked by a ticket numbered "JFS-284", you could add a line:

```
* Blocks: JFS-284
```

Note: Both the issue relationship and ticket number are case-insensitive, but that if you enter a relationship name that does not exist, you will receive an error message when `push`-ing up your changes. If you see such an error message, don't fret – just change your relationship name to one of the suggested names, `commit`, and `push` again.

1.4.2 Remote Links

You can add links to arbitrary URLs by adding bulleted lines in the following format:

```
* NAME: URL
```

If you, for example, wanted to add a link to your issue that pointed users toward your favorite cat video, you could, for example, add a line:

```
* Cat scares compilation: https://www.youtube.com/watch?v=DBRgFLHra48
```

CHAPTER 2

Common Commands

The following commands are sure to be commonly used. Be sure to check out Advanced Commands if you are curious about less-commonly-used functionality.

Note: Commands marked with an asterisk can be ran from either an issue folder, or from within a folder containing many issue folders.

In the latter case, the command will be ran for every subordinate issue folder.

2.1 clone <source>

Requires a single parameter (`source`) indicating what to clone.

Possible forms include:

- `clone http://my.jira.server/browse/MYISSUE-1024 [PATH]`
- `clone MYISSUE-1024 [PATH]` (will use default JIRA instance)

Create a new issue folder for `MYISSUE-1024` (replace `MYISSUE-1024` with an actual JIRA issue number), and clone the relevant issue into this folder.

Note that you may specify a full URL pointing to an issue, but if you do not specify a full URL, your default JIRA instance will be used; if you have not yet set one, you will be asked to specify one.

Although by default, the issue will be cloned into a folder matching the name of the issue, you may specify a path into which the issue should be cloned by specifying an additional parameter (`PATH` in the example forms above).

2.2 submit

Commit outstanding changes, push them to the remote server, and pull outstanding changes.

This is exactly equivalent to running a `commit` followed by a `push`.

2.3 commit

From within an issue folder, commits local changes and marks them for submission to JIRA next time `push` is run.

Note: Unlike git (but like mercurial), you do not need to stage files by running a command analogous to git's 'add' before committing. The commit operation will automatically commit changes to all un-committed files.

2.4 `pull` *

From within an issue folder, fetches remote changes from JIRA and merges the changes into your local copy. This command is identical to running `fetch` followed by `merge`.

2.5 `push` *

From within an issue folder, discovers any local changes, and pushes your local changes to JIRA.

2.6 `status` *

From within an issue folder, will report both any changes you have not yet committed, as well as any changes that would take place were you to run `jirafs push`.

2.7 `open` *

From within an issue folder, opens the current JIRA issue in your default web browser.

2.8 `subtask <summary>`

From within an issue folder, creates a new subtask of the current JIRA issue.

2.9 `assign [<username>]`

Change the assignee of the JIRA issue to the username specified. If one does not specify a username, the assignee will be set to the currently authenticated user.

2.10 `transition`

From within an issue folder, allows you to transition an issue into any state available in your workflow.

Possible forms include:

- `transition`: The user will be presented with state options for selection at runtime.
- `transition 10`: Transition into the state with the ID of '10'.
- `transition "closed"`: Transition into the state with the name "closed". Note that state names are case-insensitive.

Note: Note that the options available are dependent upon the user account used for authentication.

CHAPTER 3

Advanced Commands

You will probably not have a need to use the below commands, but they are available for adventurous users.

3.1 `fetch`

Fetch upstream changes from JIRA, but do not apply them to your local copy. To apply the fetched changes to your local copy, run `merge`.

3.2 `merge`

From within an issue folder, merges previously-fetched but unmerged changes into your local copy.

3.3 `diff`

From within an issue folder, will display any local changes that you have made.

3.4 `field <field name>`

Write the content of the field named `field name` to the console. Useful in scripts for gathering, for example, the ticket's `summary` field.

Note that you can also access subkeys in fields containing JSON by using a dotpath.

3.5 `setfield <field name> <value>`

Set the value of the field named `field name` to the value `value`. This is useful for programmatically changing the status of various fields.

Note that you can also access subkeys in fields containing JSON by using a dotpath.

3.6 match <field name> <value>

Return a status code of `0` if the field `field name` matches the value `value`. This is useful for allowing you to programmatically perform certain actions on fields matching certain values – for example: moving resolved issues into an archive folder.

As with all commands, check `--help` for this command; you'll find utilities allowing you to invert the check (for returning `0` when the check does **not** match) and utilities for executing a command when the field does not match.

Note that you can also access subkeys in fields containing JSON by using a dotpath.

3.7 log

From within an issue folder, will print out the log file recording actions Jirafs has performed for this ticket folder.

3.8 config

Get, set, or list configuration values. Requires use of one of the following sub-options:

- `--get <SETTING_NAME>`: Get the value of this specific parameter name.
- `--set <SETTING_NAME> <VALUE>`: Set the value of this specific parameter.
- `--list`: List all settings currently configured in the current context. When used within an issue folder, will list this issue's settings, but when used outside of an issue folder, will display only global configuration.

You may also use the `--global` argument to ensure that configuration changes or lists use or affect only the global configuration.

3.9 plugins

List, activate, or deactivate plugins by name.

Plugins provides several sub-options:

- `--verbose`: Display information about each plugin along with its name.
- `--enabled-only`: List only plugins that are currently enabled.
- `--disabled-only`: List only plugins that are available, but not currently enabled.
- `--enable=PLUGIN_NAME`: Enable a plugin by name for the current issue folder.
- `--disable=PLUGIN_NAME`: Disable a plugin by name for the current issue folder.
- `--global`: Used with `--enable` or `--disable` above, will enable or disable a plugin globally. Note: per-folder settings always take priority.

3.10 build

Run build scripts for any installed plugins. This will occur automatically during `commit`, but if you need to examine the output of a build before uploading, you can use this command to preview the results.

3.11 git

From within an issue folder, will provide direct access to this issue folder's internal git repository. This interface is not intended for non-developer use; please make sure you know what you're doing before performing git operations directly.

3.12 debug

From within an issue folder, will open up a python shell having access to a variable named `folder` holding the Python object representing the ticket folder you are currently within.

3.13 search_users <term>

Search for users matching the specified search term. This is particularly useful if you're not sure what somebody's username and you were hoping to mention them in a ticket so they get an e-mail notification.

3.14 create

Creates a new issue. Provides the following options:

- `--summary`: The summary to use for your new issue.
- `--description`: The description to use for your new issue.
- `--issuetype`: The issue type to use for your new issue (defaults to 'Task').
- `--project`: The project key to use for your new issue. This is the short, capitalized string you see next to issues. For example, if your tickets were named something like KITTENS-12084, 'KITTENS' is the project key.
- `--quiet`: Do not prompt user to provide values interactively.

If any of the above values are not specified, the user will be prompted to provide them interactively.

CHAPTER 4

Configuration

Settings affecting all issues are set in the following files:

- `~/.jirafs_config`: Global configuration values affecting all issues.
- `~/.jirafs_ignore`: Global list of patterns to ignore completely; these files differ from `.jirafs_local` below in that they **will not** be tracked in the underlying git repository. See *Ignore File Format* for details.
- `~/.jirafs_local`: Global list of patterns to ignore when looking through issue directories for files to upload to JIRA. Note that these files **will** continue to be tracked in the underlying git repository. See *Ignore File Format* for details.
- `~/.jirafs_remote_ignore`: A list of patterns to ignore when looking through files attached to a JIRA issue. Files matching any of these patterns will not be downloaded. See *Ignore File Format* for details.

You may also add any of the below files into any issue directory (in this example, `MYISSUE-1024`):

- `MYISSUE-1024/.jirafs/config`: Configuration overrides for this specific issue folder. Settings set in this file will override – for this folder only – any values you have set in `~/.jirafs_config`.
- `MYISSUE-1024/.jirafs_ignore`: A list of patterns to ignore completely; these files differ from `.jirafs_local` below in that they **will not** be tracked in the underlying git repository. See *Ignore File Format* for details.
- `MYISSUE-1024/.jirafs_local`: A list of patterns to ignore when looking through this specific issue directory. This list of patterns is in addition to patterns entered into `~/.jirafs_ignore` above. Note that these files **will** continue to be tracked in the underlying git repository. See *Ignore File Format* for details.
- `MYISSUE-1024/.jirafs_remote_ignore`: A list of patterns to ignore when looking through files attached to this specific JIRA issue. Files matching any of these patterns will not be downloaded. These patterns are in addition to the patterns entered into `~/.jirafs_remote_ignore` above. See *Ignore File Format* for details.

4.1 Using an untrusted HTTPS certificate

If your JIRA instance uses a self-signed certificate or you are working in an enterprise environment having a non-standard certificate authority, you can manually configure your JIRA connection to either not verify the certificate, or to instead use a non-standard certificate authority certificate.

1. First, find the configuration section in your `~/.jirafs_config` named after the address of your JIRA server.
2. Then, after the lines starting with `username` and `password`, add a line reading `verify = <VALUE>` replacing `<VALUE>` with one of two options:
    - If your JIRA instance uses a self-signed certificate: the string `false`.

- If your JIRA instance's certificate uses a non-standard certificate authority, the absolute path to a place on your computer where your certificate authority's certificate is stored.

For example:

```
[https://jira.mycompany.org]
username = myusername
password = mypassword
verify = /path/to/certificate/or/false
```

4.2 Disabling "Save JIRA Password" prompt

If you would never like to save your JIRA password in Jirafs, you can disable the "Save JIRA Password" prompt by setting the `ask_to_save` setting to `false` in the `main` section of your `~/.jirafs_config` file.

For example:

```
[main]
ask_to_save = false
```

CHAPTER 5

Using Plugins

- Enable the plugin for a given ticket folder:

```
jirafs plugins --enable=my_plugin_name
```

- Enable the plugin globally:

```
jirafs plugins --global --enable=my_plugin_name
```

5.1 Writing your own Plugins

Jirafs plugins come in two different varieties:

- "Folder Plugins" are used for altering the behavior of existing commands when interacting with a Ticket Folder. They can be disabled or enabled on a per-folder basis, too.
- "Command Plugins" are used for adding new commands to Jirafs. These are always enabled when installed.

Note: All existing Jirafs commands ('clone', 'pull', 'push', etc.) are "Command Plugins".

5.1.1 Writing Plugins

For a working example of a folder plugin, check out Jirafs-Pandoc's Github Repository.

Setuptools Entrypoint

- Add a setuptools entrypoint to your plugin's `setup.py`:

```
entry_points={
  'jirafs_plugins': [
      "my_plugin_name = module.path:ClassName"
  ]
}
```

- Write a subclass of `jirafs.plugin.Plugin` implementing one or more methods using the interface described in *Folder Plugin API*.

Folder Plugin API

The following properties **must** be defined:

- `MIN_VERSION`: The string version number representing the minimum version of Jirafs that this plugin will work with.
- `MAX_VERSION`: The string version number representing the maximum version of Jirafs that this plugin is compatible with. Note: Jirafs uses semantic versioning, so you may set this value accordingly.

The following methods may be defined for altering Jirafs behavior.

Alteration Methods

- `alter_filter_ignored_files(filename_list)`:
  - Further filter the list of files to be processed by reducing this list further.
  - Return further filtered `filename_list`.
- `alter_new_comment(comment)`:
  - Alter the returned comment.
  - Return an altered `comment` string.
- `alter_remotely_changed(filename_list)`:
  - Alter the list of remotely changed files if necessary.
  - Return an altered `filename_list`.
- `alter_file_upload((filename, file_like_object, ))`:
  - **DEPRECATED**: Use **run_build_process** and **get_ignore_globs** instead.
  - Alter a file pre-upload.
  - Return a new tuple of `(filename, file_like_object)`.
- `alter_file_download((filename, file_content, ))`:
  - Alter a file pre-save from JIRA.
  - Return a new tuple of `(filename, file_like_object)`.
- `alter_get_remote_file_metadata(file_metadata)`:
  - Alter remote file metadata dictionary after retrieval.
  - Return an altered `file_metadata` dictionary.
- `alter_set_remote_file_metadata(file_metadata)`:
  - Alter remote file metadata dictionary before storage.
  - Return an altered `file_metadata` dictionary.
- `alter_status_dict(status_dict)`:
  - Executed after running `status`.
  - `status_dict` dictionary (see tests and source for details):
    - `uncommitted`: A dictionary containing uncommitted changes.
    - `ready`: A dictionary of changes ready for submission to JIRA.

        * `up_to_date`: A boolean value indicating whether the current `master` branch is up-to-date with changes fetched in the `jira` branch.
    - Return an altered `status_dict`.
- `get_ignore_globs`:
    - Executed when building the ignore glob list. Use this if your plugin compiles files of one type to another so you can prevent the source file from being uploaded.
- `run_build_process`:
    - Will be executed for every plugin before running `commmit` or when running `build`. Although this is intended for use when autocompiling files from one format to another for JIRA upload, feel free to experiment.

Note: For technical reasons, both `alter_file_upload` and `alter_file_download` accept a single tuple argument containing the filename and object rather than two arguments.

Pre/Post Command Methods

All commands (including user-installed commands) can have plugins altering their behavior by defining `pre_*COMMAND*` and `post_*COMMAND*` methods. For the below, please replace `*COMMAND*` with the command your plugin would like to alter the behavior of.

- `pre_*COMMAND*(**kwargs)`:
    - Executed before handling `*COMMAND*`. Receives (as `**kwargs`) all parameters that will be passed-in to the underlying command.
    - You may alter the parameters that will be passed-in to the underlying command by returning a new or altered `**kwargs` dictionary.
    - Return `None` or the original `**kwargs` dictionary to pass original arguments to the command without alteration.
- `post_*COMMAND*(returned)`:
    - Executed after handling `*COMMAND*`. Receives as an argument the result returned by the underlying command.

Note: Although the return values of commands are not in the scope of this specification, many commands return a `jirafs.utils.PostStatusResponse` instance.

Such an instance is a named tuple containing two properties:

- (bool) `new`: Whether the command's action had an effect on the underlying git repository.
- (string) `hash`: The hash of the relevant repository branch's head commit following the action.

Properties

The plugin will have the following properties and methods at its disposal:

- `self.ticketfolder`: An instance of `jirafs.ticketfolder.TicketFolder` representing the jira issue that this plugin is currently operating upon.

- `self.get_configuration()`: Returns a dictionary of configuration settings for this plugin.
- `self.get_metadata()`: Returns a dictionary containing metadata stored for this plugin.
- `self.set_metadata(dict)`: Allows plugin to store metadata. Data **must** be JSON serializable.

Macro Plugin API

Macro plugins are special kinds of plugins that are instead subclasses of either `jirafs.plugin.BlockElementMacroPlugin` or `jirafs.plugin.VoidElementMacroPlugin`, but same setuptools entrypoints apply as are described in *Setuptools Entrypoint*.

Block Element Macros

Block element macros are macros that wrap a body of text – for example:

```
{my-macro}
Some content
{my-macro}
```

Note that – following JIRA's markup conventions, the macro both begins and ends with the name of your macro. Your macro class needs to have only one method – `execute_macro` which receives both the text content wrapped by the two `{my-macro}` markers, as well as any parameters (as keyword arguments).

Note: See *Parameters* for more information about parameters.

Your `execute_macro` method is expected to return text that should be sent to JIRA instead of your macro.

Void Element Macros

Void element macros and block element macros share a lot of similarities, except that void element macros do not need to be closed; for example:

```
{my-void-element-macro}
```

Your `execute_macro` method is expected to return text that should be sent to JIRA instead of your macro. Note that the method signature remains identical to that of a block element macro, but instead of receiving the content of the block, you will receive `None`.

Parameters

Both block and void elements can receive any number of parameters; they're specified following JIRA's conventions in which each parameter is separated by a pipe, and the key and value (if specified) are separated by an equal sign; for example the following void element has three parameters:

```
{flag-image:country_code=US|size=300|alternate}
```

- `country_code`: `US`
- `size`: `300`
- `alternate`: `True`

Note: All parameters – except `True` in the third example above – are passed as strings, and `True` is only a default value for parameters that do not have a value specified.

Example Macro Plugin

The following plugin isn't exactly useful, but it does demonstrate the basic functionality of a plugin:

```
class Plugin(BlockElementMacroPlugin):
    COMPONENT_NAME = 'upper-cased'

    def execute_macro(self, data, prefix='', **kwargs):
        return prefix + data.upper()
```

When you enter the following text into a JIRA ticket field:

```
{upper-cased:prefix=Hello, }
my name is Adam.
{upper-cased}
```

the following content will be sent to JIRA instead:

```
Hello, MY NAME IS ADAM.
```

Warning: Note that it's always a good idea to make sure your `execute_macro` method has a final parameter of `**kwargs`! In future versions of Jirafs, we may add more keyword arguments that will be sent automatically.

5.1.2 Writing Command Plugins

For a working example of a command plugin, check out the source of Jirafs existing commands.

Setuptools Entrypoint

- Add a setuptools entrypoint to your plugin's `setup.py`:

```
    entry_points={
      'jirafs_commands': [
          "my_command_name = module.path:ClassName"
      ]
    }
```

- Write a subclass of `jirafs.plugin.CommandPlugin` implementing one or more methods using the interface described in *Plugin API*.

Plugin API

The following properties **must** be defined:

- `MIN_VERSION`: The string version number representing the minimum version of Jirafs that this plugin will work with.
- `MAX_VERSION`: The string version number representing the maximum version of Jirafs that this plugin is compatible with. Note: Jirafs uses semantic versioning, so you may set this value accordingly.

The following methods may be defined to control the behavior of your command plugin:

- `handle(self, args, folder, jira, path, **kwargs)`: (**REQUIRED**) This method (and methods called from here) is where you should write the bulk of your plugin's functionality. `handle` receives several keyword arguments:
  - `args`: An instance of `argparse.Namespace` holding arguments specified on the command line. See `add_arguments` and `parse_arguments` for details.
  - `folder`: A `jirafs.ticketfolder.TicketFolder` instance corresponding with the current path. If you are writing a command that does not require a ticketfolder, set an attribute on your class named `AUTOMATICALLY_INSTANTIATE_FOLDER` to `False` (Note that this option makes the value of `TRY_SUBFOLDERS` irrelevant) and this value will always be `None` whether or not your command was invoked from within a ticket folder.
  - `jira`: A callable (accepting, optionally, the string domain of a JIRA instance) which will return an instance of `jira.client.JIRA` corresponding with the domain you've specified, or the default JIRA connection if no JIRA domain was specified.
  - `path`: The string path from which this command was called. This can be used to create a `jirafs.ticketfolder.TicketFolder` instance representing the current ticket folder if so desired.
  - `**kwargs`: Keyword arguments **may** be added in the future; it is extremely important that your `handle` method accept arbitrary keyword arguments in order to prevent your plugin from breaking when new keyword arguments are added in the future.
- `add_arguments(self, parser)`: Using this method, you can add arguments that your command requires. Follow the guidelines in Python's `argparse` documentation for an overview of how arguments are handled.
  - `parser`: An `argparse.ArgumentParser` instance.
- `parse_arguments(self, parser, extra_arguments)`: Potentially useful as a method to place argument validation.
  - `parser`: An `argparse.ArgumentParser` instance. Note that this instance will have already had attached all arguments added in the `add_arguments` method above.
  - `extra_arguments`: A list of string arguments unused by Jirafs.

You may also use any of the following properties to alter the behavior of Jirafs:

- `TRY_SUBFOLDERS`: Set this class property to `True` if this command should be applied to all Jirafs ticket folders in subdirectories in the event that the current folder is not a ticket folder.
- `RUN_FOR_SUBTASKS`: Set this class property to `True` if you would like your command to be automatically executed for subtask when being executed for a ticket having subtasks.

Example Plugin

```
import pydoc

from jirafs.plugin import CommandPlugin


class Command(CommandPlugin):
    """ Run a git command against this ticketfolder's underlying GIT repo """
    MIN_VERSION = '1.15'
    MAX_VERSION = '1.99.99'
```

```
    def handle(self, args, folder, **kwargs):
        return self.cmd(folder, *self.git_arguments)

    def parse_arguments(self, parser, extra_args):
        args, git_arguments = parser.parse_known_args(extra_args)
        self.git_arguments = git_arguments
        return args

    def main(self, folder, *git_arguments):
        result = folder.run_git_command(*git_arguments)
        pydoc.pager(result)
        return result
```

CHAPTER 6

Using Macros

Macros are special kinds of plugins that perform simple functions for transforming text you enter into fields into something else when submitting them to JIRA.

6.1 Existing Macros

- jirafs-list-table: Make JIRA tables a little more easily by using a simple list-based syntax.

6.2 Writing your own Macros

Macros are really just special kinds of plugins; you can find more information about writing your own plugins in *Macro Plugin API*.

CHAPTER 7

Interesting Details

7.1 Ignore File Format

The files `.jirafs_local`, `.jirafs_ignore` and `.jirafs_remote_ignore` use a subset of the globbing functionality supported by `git`'s `gitignore` file syntax. Specifically, you can have comments, blank lines, and globbing patterns of files that you would not like to upload.

For example, if you'd like to ignore files having a `.diff` extension, and would like to add a comment indicating why those are ignored, you could enter the following into any `*_ignore` file:

```
# Hide diffs I've generated for posting to reviewboard
*.diff
```

7.2 Directory Structure

Each issue folder includes a hidden folder named `.jirafs` that stores metadata used by Jirafs for this issue. There may be many things in this folder, but two highlights include the following files/folders:

- `git`: The issue folder is tracked by a git repository to enable future features, provide for a way of easily rolling-back or reviewing an issue's previous state.
- `operation.log`: This file logs all operations engaged in on this specific issue folder. You can review this log to see what `jirafs` has done in the past.

7.3 VIM Plugin

If you're a vim user, I recommend you install my fork of the confluencewiki.vim plugin; if you do so, comment and description field files will use JIRA/Confluence's WikiMarkup for syntax highlighting.