Introduction to Filesystem and GitFS

Max Leske
Software Composition Group
University of Bern, Switzerland

March 22, 2011

Abstract

The purpose of this document is to provide users of Filesystem and GitFS with a short introduction to the main features and the basic use of both systems. When the reader has finished reading he or she will be able to work with any filesystem that supports the Filesystem public protocol and specifically know how to create, read and manipulate Git repositories from within a Pharo Smalltalk image.

Introduction

This document consists of two parts: The first part explains how to use FILESYSTEM and gives examples on common operations; The second part focuses on the GITFS extension of FILESYSTEM and shows how Git repositories can be used from Pharo Smalltalk. A basic knowledge of the concepts and keywords in Git is necessary for reading this document. We recommend the reader study the tutorials available from the Git project website at http://www.git-scm.com.

The requirements to successfully reproduce the examples discussed on the following pages are a Pharo image (the one-click image from the Pharo website at http://www.pharo-project.org is recommended), a basic knowledge of the Pharo environment (this includes a basic knowledge of Smalltalk. Consider reading "Pharo by Example" [**Blac09a**] for an introduction.) and an internet connection to load additional packages.

1 How to Use Filesystem

The purpose of this section is to give a concise overview of how to use the Filesystem framework. We use a disk filesystem for the examples because this type of filesystem is known to most people. Nevertheless, most of the messages and workflows can be equally applied to other filesystems since every filesystem inherits the public protocol of `FSFilesystem`.

To start using Filesystem the relevant Monticello package needs to be loaded into the image. This can be done by evaluating the following in a workspace:

```
Gofer new
   renggli: 'fs';
   package: 'Filesystem';
   load.
```

1.1 Filesystems and references

The disk filesystem is implemented in `FSDiskFilesystem` and its platform specific subclasses. An instance of the filesystem of the current platform can be retrieved by evaluating the following in a workspace:

```
disk := FSDiskFilesystem current.
```

Subclasses of `FSFilesystem` implement many methods, but most of them should not be called directly. These methods implement the low-level behavior of the respective filesystems and are private to the framework. References (instances of `FSReference`) are the central object of the framework and provide the primary mechanism for working with files and directories. We now ask the filesystem for a reference to the working directory and assign it to the variable "working". The working directory is the directory containing the running image.

```
working := disk working.
```

Sending the message `#root` to the filesystem object returns a reference to the root of the filesystem ("/" in Unix, "C:\" in Windows usually). But since we do not want to corrupt any data and permissions might be a problem we will use the working directory instead.

1.2 Navigating the Filesystem

The reference assigned to the variable "working" allows us to browse the filesystem. We list the entries of a directory by sending `#children` to the reference:

```
working children.
```

Sending `#allChildren` to the reference answers a collection of all subdirectories recursively:

```
working allChildren.
```

The slash operator answers a reference to a specific file or directory within the working directory:

```
cache := working / 'package-cache'.
```

Navigating back to the parent is easy. We send `#parent` to the reference. We expect the parent to be the working directory and test for that:

```
parentOfCache := cache parent.
parentOfCache = working.                " --> true "
```

The slash operator interprets the string sent to it as the name of the directory to navigate to. If the string argument contains a slash it is interpreted as part of the directory name and not as a separator. To remedy the situation references understand a second navigation message called `#resolve:`. Resolve expects a string argument in the form used by the "cd" command known from the Unix command line (the same syntax applies to Windows systems):

```
disk working resolve: '/'.              " the same as 'disk root' "
disk working resolve: '.'.              " the same as 'disk working' "
disk working resolve: '/home/leske'.    " an absolute path to a directory or file "
disk working resolve: '../projects'.    " a relative path from the working directory "
```

Evaluating the following expressions reveals information about the cache directory:

```
cache exists.                           " --> true "
cache isFile.                           " --> false "
cache isDirectory.                      " --> true "
cache basename.                         " --> 'package-cache' "
```

Even more information is available via the filesystem entry. The entry is an instance of `FSDirectoryEntry`. Every reference has an associated entry holding the header information:

```
cache entry creation.                   " --> 2010-09-14T10:34:31+00:00 "
cache entry modification.               " --> 2010-09-14T10:34:31+00:00 "
cache entry size.                       " --> 0 (directories have size 0) "
```

The framework also supports *locations*; they are late-bound references that point to a file or directory. When asking to perform a concrete operation a location behaves the same way as a reference. Filesystem supports the following locations:

```
FSLocator desktop.
FSLocator home.
FSLocator image.
FSLocator vmBinary.
FSLocator vmDirectory.
```

A location will dynamically adapt and always point to the place expected even when moving the image to another platform.

1.3 Opening Read and Write Streams

`FSReference` provides easy access to streams:

```
stream := (working / 'letter.txt') writeStream.
stream nextPutAll: 'Hello Alice'.
stream close.
stream := (working / 'letter.txt') readStream.
stream contents.                    " --> 'Hello Alice' "
stream close.
```

Note that `#writeStream` overrides any existing file and that `#readStream` throws an exception if the file does not exist. There are also short forms available which ensure the correct closing of the stream:

```
working / 'letter.txt' writeStreamDo: [ :stream | stream nextPutAll: 'Hello Alice' ].
working / 'letter.txt' readStreamDo: [ :stream | stream contents ].
```

1.4 Creating Renaming, Copying and Deleting Files and Directories

Files and directories can be copied between references in the same filesystem (first line of next example) or even across different types of filesystems (second line of next example) such as between a disk filesystem and a memory filesystem:

```
(working / 'letter.txt') copyTo: (working / 'letter_backup.txt').

memory := FSMemoryFilesystem new working.
(working / 'letter.txt') copyTo: (memory / letter_backup.txt).
memory children first basename.       " --> 'letter.txt' "
```

Supposing one wanted to backup an entire directory there needs to be a suitable target directory. Since there are no directories in the memory filesystem just created we will now create one to hold the mentioned backup:

```
backup := memory / 'cache-backup'.
backup createDirectory.
```

Finally, we copy the directory to be backed up to the destination directory:

```
cache copyAllTo: backup.
```

The target directory would also have been created automatically by the message `#copyAllTo:` had it not existed already. From time to time the backup needs to be cleaned up. In this case, we no longer need the backup of the file “letter.txt”:

```
(memory / 'letter_backup.txt') delete.
```

A complete directory tree can be deleted by sending `#deleteAll`:

```
backup deleteAll.
```

1.5 Other Filesystems

The filesystem framework is easily extensible and there are several implementations including a ZIP, a cURL and a Git filesystem. All of these implementations comply with the public protocol of the framework and therefore it does not matter to the user on which filesystem he or she is working. Some filesystems specify additional messages to provide access to functionality beyond the scope of a simple filesystem. The ZIP filesystem for example defines the message `#close` which needs to be sent to the instance of a ZIP archive to flush it.

GitFS is one of the extensions it provides access to Git repositories through Filesystem. The following section gives an overview of the available messages and shows how to perform basic operations on repositories.

2 How to use GitFS

GitFS extends the Filesystem framework to enable work with Git repositories. The class `FSGitFilesystem` defines additional messages to the standard Filesystem protocol to manipulate and read repositories. GitFS is not part of the standard Filesystem package and needs to be loaded into the image separately. The following code loads all the required packages (with their required versions) into the image:

```
Gofer new
   url: 'http://www.squeaksource.com/GitFS';
   package: 'ConfigurationOfPharogenesis';
   load.
((Smalltalk at: #ConfigurationOfFSGit) project version: #stable) load.
```

2.1 Working with Repositories

There are two classes a user of GitFS needs to know: `FSGitRepository` and `FSGitFilesystem`. `FSGitFilesystem` implements the public Filesystem protocol and is the central object for most of the work. `FSGitRepository` encapsulates the knowledge of communication with a Git repository and is the starting point for working with Git.

A speciality about GitFS is that it provides a filesystem on top of another filesystem. Therefore, to instantiate a repository we need a reference to another filesystem. For this introduction we will work on a disk filesystem:

```
   repoReference := FSDiskFilesystem current root resolve: '/Developer/webApp'.
   repo := FSGitRepository on: repoReference.
```

`FSGitRepository` defines four messages to browse the contents of a git repository:

```
   repo head.
```

`#head` answers an `FSGitFilesystem` object representing the head commit of the current repository.

```
   repo branches.
```

`#branches` answers a dictionary of branch names (strings) associated with `FSGitFilesystem` objects. Each object represents the head commit of a branch.

```
   repo tags.
```

`#tags` answers a dictionary of tag names (strings) associated with `FSGitFilesystem` objects. Each object represents a commit referenced by a tag found in the repository.

```
   repo versions.
```

`#versions` answers a collection of `FSGitFilesystem` objects, one for every commit found in the repository (independent of the active branch). Depending on the size of the repository this statement may take several minutes to complete.

2.2 First Steps with the Git filesystem

The idea behind using `FSGitFilesystem` objects is to modify the working copy of a Git repository and then commit those changes. Here is a simple example of how to modify the working copy. In a first step we assign an instance of `FSGitFilesystem` to the variable "workingCopy". The `FSGitFilesystem` object contains the working copy of the head commit of the created repository (the repository is empty so the working copy is too):

```
workingCopy := repo head.
```

We then assign a reference (instance of `FSReference`) to the variable "newFile". The reference points to a file that does not yet exists. We create it by writing some content to it:

```
newFile := workingCopy root / 'newfile.txt'.
newFile writeStreamDo: [ :stream |
   stream nextPutAll: 'some content' ].
```

Note that `workingCopy root` answers the root of the Git filesystem and not the root of the disk filesystem. If you made a mistake and simply want to have a clean workingCopy again you can send `#reset` to the workingCopy:

```
workingCopy reset.
```

Assuming that the changes are fine, the file is committed to the repository thus saving the state of the working copy. The message `#commit` expects a message string describing the changes made:

```
workingCopy commit: 'created a new file'.
```

Committing the changes will create and update objects and references in your repository as needed. After the commit has succeeded the `FSGitFilesystem` object (the one referenced by the variable workingCopy) will point to that commit with all the recent changes reflected in the working copy.

The contents of the file we committed can be retrieved using the Filesystem protocol:

```
fileContents := newFile readStreamDo: [ :stream |
   stream contents asString ].
```

2.3 Advanced manipulations

To provide a better picture of what Git is capable of we will create a second commit so that the repository holds two versions of the working copy:

```
anotherFile := workingCopy root resolve: 'dir1/anotherFile.txt'.
anotherFile writeStreamDo: [ :stream |
   stream nextPutAll: 'beginning of another file.' ].
workingCopy commit: 'added a second file'.
```

Evaluating the following will now answer a collection of four references: "/", "/newFile.txt", "/dir1" and "/dir1/anotherFile.txt":

```
workingCopy root allChildren.
```

Once you are satisfied with your changes you might want to commit and at the same time mark that commit specially. GitFS offers convenience methods for this scenario:

```
workingCopy
   commit: 'a commit message'
   andTag: 'version 1.0'.
```

`#commit:andTag:` creates a commit and a tag referencing that commit (at the moment only light tags are supported). From now on it will be easy to identify the commit that was used for the release of version 1.0. You might also choose to tag another commit you made earlier:

```
workingCopy
   tagRevision: workingCopy parents first
   with: 'introduces cool new feature'.
```

Version 1.0 will probably not be the last version. It might therefore be convenient to do new work on a separate branch to separate versions clearly:

```
workingCopy
   branch: 'my fork'
   message: 'a commit message'.
```

`#branch:message:` creates a commit with the current changes and a new branch referencing that commit. If the branch already exists it will be updated to the current commit. Of course you can also commit to a new branch and tag that commit with only one message if you like:

```
workingCopy
   branch: 'my fork'
   message: 'alpha release'
   andTag: '1.1 alpha'.
```

When comparing versions it can be useful to know the signature hash of an object to determine if two objects are equal. Send `#nameOf:` to an `FSGitFilesystem` object to find the name for any object in a commit:

```
buggyVersion := workingCopy nameOf: (workingCopy root / 'aFileWithABug.php') path.
```

You could also use the signature of an object in a commit message to tell other developers where a bug occurred:

```
workingCopy message.
```

`#message` answers the commit message of current commit, e.g. "I fixed the string conversion bug that was introduced in object ebf14a3157efd4ffbf840538414b6a6aa6e19c50".

References

[Blac09a] Andrew Black et al. *Pharo by Example*. Square Bracket Associates, 2009.