VGIT2 – a partial git implementation for OpenVMS IA64

V0.9.5, 28-JAN-2019

The vgit2 kit is a "give it a try" partial implementation of the git version control system tool for OpenVMS. You should be able to download a copy of the tool from the VSI secure FTP site using instructions provided. The tool is provided in a single ZIP file (`VGIT-28-JAN-2019.ZIP`). This release of the tool includes additional functionality, including a basic implementation of the “stash” command, and functionality to interact with remote repositories via a proxy server.

Note that the tool is for OpenVMS ia64 only at this time (OpenVMS 8.4 or higher), although it should be readily possible to build it for Alpha 8.4. Also, note that the ZIP file was created on OpenVMS, so we would suggest extracting the contents on a VMS system.

The contents of the ZIP file are as follows:

```
Archive:  SYS$SYSDEVICE:[BIGGLES.vgit]VGIT-28-JAN-2019.ZIP;1
  Length      Date    Time    Name
---------  ---------- -----   ----
  8508416  01-28-2019 10:13   vgit2.exe
   236061  11-27-2017 15:26   cert.pem
---------                     -------
  8744477                     2 files
```

Vgit2.exe is the UNIX-style version of the tool. There is also an OpenVMS CLI-based version that has largely equivalent functionality. The UNIX-style version is a little more user friendly in that it contains some built-in usage notes (more on this later), and it is likely that the CLI-based version will be discontinued.

The certificate file (cert.pem) found in the provided ZIP file is required if you are going to be interacting with some service like GitHub or BitBucket.

To get started, extract the contents of the ZIP file on your OpenVMS server and copy the files into your preferred location. Next, perform the following tasks (or add the commands to login.com):

- Define a foreign command for vgit2:

  ```
  $ vgit2 :== $device:[path]vgit2.exe
  ```

- Define the logical name git$ssl_certs to point to the certificate file:

  ```
  $ define git$ssl_certs device:[path]cert.pem
  ```

Next you’ll need to run the following command to set up some basic configuration details:

```
$ vgit2 config user -n username -e youremail
```

After this command completes, you should see in your login directory the file `git.config`, and it should contain the details that you specified in the above command. It is better to create this file

before doing anything else to avoid being told by the program to do this the first time you come to do anything that interacts with a repository. Note that the values you specify are not particularly critical as they can be overridden if/when necessary; however if you will primarily be interacting with some service such as GitHub then you should specify your username for that service and the associated email address.

If you will need to interact via a proxy server with remote repositories hosted on services such as GitHub and BitBucket now would also be a good time to define details of your proxy server, which can be done using the following command, substituting the correct URL as necessary:

```
$ vgit2 config http.proxy http://myproxy.server:8080
```

As mentioned above, the UNIX-style version has some level of built-in usage notes. For example, running the program without specifying a command or specifying an invalid command, you will get the following output:

```
$ vgit2
Usage: EXEC14$DKA200:[CAMERON.vgit]vgit2.exe;2 <command> [options] [arguments]
Command summary:
        add [-s] [-v] [-u] <file> ...
        blame [-L <line-range>] [-F] [-M] [-C] [<commit-range>] <path>
        branch [[-d <branch-name>] | <branch-name>]
        checkout [-s safe | create | force] [-a conflicts] [-r untracked | ignored] [-b ref]
[<file> ...]
        clone [-b <name>] <uri> [<path>]
        commit [-a] [-m <message> | -F <filename>]
        config user [-s global | local] [-n <username>] [-e <email>]
        config http.proxy [-s global | local] <url>
        diff [-s] [-c] [<commit> <commit>] [--] [<path>...]
        fetch [<remote>]
        init [-b] [-q] [-s true | false | group | all | world | umask] [-t <file>] [-n (no
initial commit)] [<path>]
        ls-remote <repository>
        merge [-s safe | create | force]
        pull
        push [-t | <refspec>]
        rebase <upstream> <branch>
        remote add <name> <uri>
        remote show
        rm [-b] [-s <stage>] <file>
        show-index
        show-ref
        stash [push | pop | drop | clear]
        status [-b] [-f short | long | porcelain]
        tag -d <tag-name> | [-f] [-m <message>] <tag-name> [<commit> | <object>]
        tag list [-l <number>] [pattern]
        reset [--soft | --mixed | --hard] [<commit>]
        log [<options>] [<revision-range>] [-- <path>...]
        version

        Repositories and your login directory must be on an ODS-5 file systems
        Files must be stream-lf
```

This is a basic summary of the available commands. If you now want a few more details on a specific command, run the program specifying the command in question and supply some invalid option or leave out a required parameter. For example:

```
$ vgit2 init -h
init: illegal option -- h
Usage: init [-b] [-q] [-s true | false | group | all | world | umask] [-t <file>] [-n]
[<path>]

Options:
        -b           Create a bare repository (has no working tree).
        -q           Quiet (only display errors and warnings).
        -s <value>   Repsoitory sharing (permitted values are "true", "false", "group",
                     "all", "world", and "umask")
```

```
        -t <file>    Specify a template directory.
        -n           Do not perform an initial commit.

Arguments:
        <path>       Directory where the repository is to be created (the directory is created
                     if necessary). If not specified, the current location is used.
```

You will notice that some commands map reasonably well to real git (although some options may not be available); other commands do not map quite so well (currently).

Note that it is possible to create a primary repository on shared disk in an OpenVMS cluster, and have team members clone the repo into their private work areas and push changes back to the primary repo so that others can sync off that by doing a pull or whatever. However, chances are that you will most likely want to use the tool with some service like GitHub or BitBucket, in which case typical usage will be pretty much exactly as per real git (with a few syntactical differences).

Some other points to note:

1. All files must be stream-lf. The program (currently) will exit with an error if you try to add a file to the repo that is not stream-lf. If for any reason the “add” command fails, note that nothing will have been added to the repository (it’s an all or nothing scenario), and you can safely fix up the problem (maybe you have to convert a file to stream-lf) and re-run the “add” command.

    - When adding some number of files we would suggest using the “-v” (verbose) flag. This will allow you to see which file the add operation is failing on (probably because the file is not stream-lf).

2. ODS-5 only.

3. You may notice that it will be a somewhat slow process to do an initial load or to clone repositories with a large number of files, but you only need to do this once, after which committing, pushing, and pulling changes should not be too painful. The “status” command can also be a bit slow if you have a large number of files in your project.

4. If you want to use key-based authentication (as opposed to username/password over HTTPS, for example), you will need to define the logical names git$id_rsa and git$id_rsa_pub to map to the private and public key files, respectively (and of course your public key will need to be registered with the service (GitHub or whatever)).

We are sure you will encounter things that may not work as expected, but rather than trying to pre-empt every possible scenario, it is probably better for you to give the tool a try and contact us if/when you have problems and/or if you would like additional functionality added, and we will try to oblige!