viki-fabric-helpers Documentation

Release 0.0.5

Viki Inc.

July 04, 2014

Contents

Release v0.0.5

A collection of Fabric helper functions used in some of Viki's projects.

CHAPTER 1

Installation

1.1 Installation

1.1.1 pip (requirements.txt)

Append this line to *requirements.txt*:

```
viki-fabric-helpers
```

Followed by running:

```
pip install -r requirements.txt
```

1.1.2 pip (command line installation)

From the command line:

```
pip install viki-fabric-helpers
```

1.1.3 Getting the code (via Github)

Cloning the repository:

```
git clone https://github.com/viki-org/viki-fabric-helpers.git
```

Downloading a tarball:

```
wget -O viki-fabric-helpers.tar.gz https://github.com/viki-org/viki-fabric-helpers/tarball/master
```

Downloading a zipball:

```
wget -O viki-fabric-helpers.zip https://github.com/viki-org/viki-fabric-helpers/zipball/master
```

CHAPTER 2

Configuration

2.1 Configuration

This page describes how you can configure the viki-fabric-helpers library.

2.1.1 The *viki_fabric_config.yml* file

The viki-fabric-helpers library makes use of a *viki_fabric_config.yml* file for configuration. This file is read once, when any module under the *viki.fabric* package is imported (the code is in the *viki/fabric/__init__.py* file).

It is not necessary to provide the *viki_fabric_config.yml* file. If you do provide it however, it should be located in the directory where the main Python script is run.

2.1.2 Contents of the *viki_fabric_config.yml* file

The *viki_fabric_config.yml* file is a YAML file. If you are not familiar with YAML, it is a concise data representation format for common data structures such as dictionaries and lists. viki-fabric-helpers makes use of the PyYAML library for reading YAML files.

Currently, only the *viki.fabric.git* module requires the *viki_fabric_config.yml* file. Refer to *viki.fabric.git - A short guide* for more information.

NOTE: The *viki_fabric_config.yml* file can be used to hold other data as long as their names do not conflict with those used by *viki-fabric-helpers*.

2.1.3 Accessing data in *viki_fabric_config.yml*

On your first import of a module under the *viki.fabric* package, the *viki_fabric_config.yml* is read and its contents are placed inside the *viki_fabric_config* key of the *fabric.api.env* variable. To access the data, you should use the *viki.fabric.helpers.get_in_viki_fabric_config*.

For instance, suppose the *viki_fabric_config.yml* file has the following contents:

```
animals:
  mammals:
    kangaroo: "jumps"
    human: "walks"
  reptiles:
    crocodle: "swims"
    lizard: "climbs"
```

To access the entire *animals* hash:

```
from viki.fabric.helpers import get_in_viki_fabric_config

# obtain the dict {'mammals': ... , 'reptiles': ...}
get_in_viki_fabric_config(["animals"])
```

To get the value of *animals.mammals.kangaroo*:

```
from viki.fabric.helpers import get_in_viki_fabric_config

# obtains the string "jumps"
get_in-viki_fabric_config(["animals", "mammals", "kangaroo"])
```

CHAPTER 3

Usage

3.1 *viki.fabric.docker* - A short guide

The *viki.fabric.docker* module contains several Docker related Fabric tasks that may help reduce the effort required for writing Fabric tasks that involve Docker.

We will be going through an example of writing a Fabric script that builds a Docker image on your local machine, pushes it to the Docker registry, followed by pulling it on a set of servers. There is extensive inline documentation to aid your understanding.

3.1.1 Example Script

NOTE: This script assumes usage of Fabric **1.9.0**. However, a version of Fabric relatively close to that *should* work as well.

```
# the module where the functions we're covering resides
import viki.fabric.docker as viki_docker
# other helper functions that we'll be needing
import viki.fabric.helpers as fab_helpers

# Fabric library imports
from fabric.api import env
from fabric.decorators import runs_once, task
from fabric.operations import run
from fabric.tasks import execute

# Fabric roles
env.roledefs = {
  "production": ["m01.prod1", "m02.prod1", "m03.prod1"],
  "testing": ["t01.test1", "t02.test1", "t03.test1"]
}

# This Fabric task is decorated with `fabric.decorators.runs_once` because
# it uses `fabric.tasks.execute` to run other Fabric tasks.
#
# Not decorating it with the `fabric.decorators.runs_once` will result in
# the following scenario:
#
#     We run a Fabric task T1 on servers S1, S2 and S3.
#     Fabric task T1 calls two other Fabric tasks T2 and T3.
#     Fabric task T1 is not decorated with `fabric.decorators.runs_once`.
```

```
#
#     What happens:
#
#         S1 runs T1
#             S1 runs T2
#             S2 runs T2
#             S3 runs T2
#             S1 runs T3
#             S2 runs T3
#             S3 runs T3
#         S2 runs T1
#             S1 runs T2
#             S2 runs T2
#             S3 runs T2
#             S1 runs T3
#             S2 runs T3
#             S3 runs T3
#         S3 runs T1
#             S1 runs T2
#             S2 runs T2
#             S3 runs T2
#             S1 runs T3
#             S2 runs T3
#             S3 runs T3
@runs_once
@task
def build_my_repo_docker_image_and_push_to_registry():
  """Fabric task which builds a Docker image for my repository and pushes
  it to the Docker registry (http://index.docker.io).
  """
  # name of the Docker image in namespace/image format
  dockerImageName = "steveJackson/myRepo"

  # Use the 'fabric.tasks.execute' function to run the
  # 'viki.fabric.docker.build_and_push_docker_image' Fabric task.
  # The 'viki.fabric.docker.build_and_push_docker_image' Fabric task is only
  # executed once regardless of the number of hosts or roles you pass to the
  # 'build_my_repo_docker_image_and_push_to_registry' task that we're
  # writing now
  retVal = execute(viki_docker.build_and_push_docker_image,
    # path to the git repository; anything that 'git clone' accepts is
    # acceptable
    "https://github.com/steve-jackson/my-repo.git",

    # name of the Docker image
    dockerImageName,

    # use 'git-crypt' (https://github.com/AGWA/git-crypt) to decrypt the
    # git-crypt'ed files in the cloned repository
    runGitCryptInit=True,

    # Path to the git-crypt key used for the repository
    gitCryptKeyPath="/home/steve/my-repo-gitcrypt-key",

    # the Dockerfile is located inside the 'docker-build' directory of the
    # cloned repository
    relativeDockerfileDirInGitRepo="docker-build",
```

```
    # pass in the hosts and roles supplied to the
    # 'build_my_repo_docker_image_and_push_to_registry' task to the
    # 'viki.fabric.docker.build_and_push_docker_image' task
    hosts=env.hosts, roles=env.roles
  )

  # We did not supply the 'dockerImageTag' keyword argument to the
  # above execute, hence we will need the tag of the newly built Docker
  # image, which is the return value of the task.
  #
  # However, we're using 'fabric.tasks.execute', which collects the return
  # value of all hosts into a dict whose keys are the host strings and the
  # whose values are the return values of the original task for the hosts.
  #
  # Since the 'viki.fabric.docker.build_and_push_docker_image' Fabric task
  # is a local Fabric task which runs once, its return value will be the
  # same for all given hosts.
  # The 'viki.fabric.helpers.get_return_value_from_result_of_execute_runs_once'
  # function is a convenience function to extract a return value from the
  # dict returned by 'fabric.tasks.execute'.
  dockerImageTag = \
    fab_helpers.get_return_value_from_result_of_execute_runs_once(retVal)

  # On each given server, pull the newly built Docker image.
  # This is run once for each server.
  execute(viki_docker.pull_docker_image_from_registry,
    # name of the Docker image in 'namespace/image' format
    dockerImageName,

    # tag of the Docker image; we obtained this above
    dockerImageTag=dockerImageTag,

    # pass in the hosts and roles given to the
    # 'build_my_repo_docker_image_and_push_to_registry' task
    hosts=env.hosts, roles=env.roles
  )
```

Suppose the above script is named *fabfile.py*. To run it for the production machines:

```
fab -R production build_my_repo_docker_image_and_push_to_registry
```

3.2 *viki.fabric.git* - A short guide

This page covers the use of the *viki.fabric.git* module. More detailed documentation for individual functions in this module can be found in the *API Documentation*.

Our focus here will be on running the *setup_server_for_git_clone* function. This function is used to setup a server for Git remote operations (such as cloning) involving secret repositories and assumes the following:

- Git remote operations are carried out using SSH
- An SSH private key is used for authentication to gain access to the secret repository

3.2.1 Configuration

Any Python script which imports the *viki.fabric.git* module directly or indirectly will require you to create a YAML file named *viki_fabric_config.yml* **at the directory where the main Python script is invoked.** This YAML file should contain a dict at the key *viki.fabric.git* containing the following keys:

ssh_private_key

Basename of the SSH private key to copy to the server.

ssh_public_key

Basename of the SSH public key to copy to the server.

ssh_keys_local_copy_dir

Folder storing the *ssh_private_key* and *ssh_public_key* files on **your local machine**.

The path to this folder can be relative to where the Python script that imports the *viki.fabric.git* module is run, or an absolute path.

ssh_keys_dir

Folder on the remote server to copy the *ssh_private_key* and *ssh_public_key* files to.

NOTE: This folder is **relative to the $HOME directory** of the **remote user** during Fabric execution. This should normally be set to the string *'.ssh'*.

git_ssh_script_name

Basename of a template git wrapper script on **your local machine**. What this script should contain is outlined later in *this subsection*.

This file will be copied to the *$HOME* directory of the user on the remote server (for such a server and user involved in a Fabric task). You can set this to any valid filename. My personal preference for this value is the string *'gitwrap.sh'*.

git_ssh_script_local_folder

Folder **on your local machine** containing the *git_ssh_script_name* file.

The path to this folder can be relative to where the Python script that imports the *viki.fabric.git* module is run, or an absolute path.

3.2.2 Example YAML file (and what it implies)

```
viki.fabric.git:
  ssh_private_key: "id_github_ssh_key"
  ssh_public_key: "id_github_ssh_key.pub"
  ssh_keys_local_copy_dir: "github-ssh-keys"
  ssh_keys_dir: ".ssh"
  git_ssh_script_name: "gitwrap.sh"
  git_ssh_script_local_folder: "templates"
```

Suppose that Fred, a user of our library, has a Python Fabric File located at */home/fred/freds-repo/fabfile.py*, which he runs from the */home/fred/freds-repo* folder. The above YAML file should be located at */home/fred/freds-repo/viki_fabric_config.yml*.

Based on the contents of the */home/fred/freds-repo/viki_fabric_config.yml* file:

- There should be a */home/fred/freds-repo/github-ssh-keys* folder containing the *id_github_ssh_key* and *id_github_ssh_key.pub* SSH keypair.

- This SSH keypair will be copied to the *$HOME/.ssh* folder on the server during execution of the *setup_server_for_git_clone* Fabric task
- There is a */home/fred/freds-repo/templates* folder containing the *gitwrap.sh* file. We shall take a look at what this file should contain in the next section.

3.2.3 Git SSH Wrapper file

This is the file specified by the value of the *git_ssh_script_name* YAML key, and should contain the following code:

```
#!/bin/bash

ssh -i {{ ssh_private_key_path }} $@
```

The *{{ ssh_private_key_path }}* part of the code will be replaced by the *setup_server_for_git_clone* Fabric task before the script is copied to the server (A temporary file or similar is used, so your file will not be accidentally modified by this task).

3.2.4 Running the *setup_server_for_git_clone* Fabric task

Assume that our imaginary user Fred

- has everything setup as we mentioned above
- has his YAML file located at */home/fred/freds-repo/viki_fabric_config.yml*
- runs the */home/fred/freds-repo/fabfile.py* file (contents right below) from the */home/fred/freds-repo* folder, using this command:

```
fab -H hostOne,hostTwo freds_fabric_task
```

Contents of */home/fred/freds-repo/fabfile.py* Fabric script:

```
from fabric.api import env, task

import os.path
import viki.fabric.git as fabric_git

# Fred uses SSH config
env.use_ssh_config = True

@task
def freds_fabric_task():
  # Fred wishes to setup the current server for handling secret repos
  fabric_git.setup_server_for_git_clone()
  # Fred's other code below
```

Suppose Fred's SSH config file looks like this (see the *env.use_ssh_config* line in the code above to understand why we put this here):

```
Host hostOne
  Hostname 1.2.3.4
  User ubuntu

Host hostTwo
  Hostname 1.2.3.5
  User ubuntu
```

The effect of successfully executing the *setup_server_for_git_clone* Fabric task (it's part of the *freds_fabric_task*):

- For the *ubuntu* user on *hostOne* and *hostTwo*, the *$HOME/.ssh* folder should contain the *id_github_ssh_key* and *id_github_ssh_key.pub* SSH keypair
- A templated *$HOME/gitwrap.sh* should be present for the *ubuntu* user on those 2 servers

Now, the *ubuntu* user on Fred's *hostOne* and *hostTwo* servers are ready for handling some secret git repositories. We shall go into that next.

3.2.5 Working with secret repos after running *setup_server_for_git_clone*

Suppose Fred SSHes into *hostOne* using the *ubuntu* user, and wishes to clone a secret repository whose clone url is *git@github.com:fred/top-secret-repo.git*, he should use this bash command to clone the git repository:

```
GIT_SSH=$HOME/gitwrap.sh git clone git@github.com:fred/top-secret-repo.git
```

In fact, this can be generalized to other Git remote operations for secret repos, such as *git fetch*. The pattern for the command to use is:

```
GIT_SSH=$HOME/gitwrap.sh <git command and args>
```

Which makes me wonder why we named the task *setup_server_for_git_clone*; perhaps this was our original use case.

CHAPTER 4

API Documentation

4.1 API Documentation

4.1.1 viki.fabric.docker

viki.fabric.docker.**construct_tagged_docker_image_name**(*dockerImageName*, *dockerImageTag=None*)

Constructs a tagged docker image name from a Docker image name and an optional tag.

Args: dockerImageName(str): Name of the Docker image in *namespace/image* format

dockerImageTag(str, optional): Optional tag of the Docker image

viki.fabric.docker.**build_docker_image_from_git_repo**()

A Fabric task which **runs locally**; it does the following:

1. clones a given git repository to a local temporary directory and checks out the branch supplied

2. If the *performGitCryptInit* argument is *True*, runs *git-crypt init* to decrypt the files

3. builds a Docker image using the Dockerfile in the *relativeDockerfileDirInGitRepo* directory of the git repository. The Docker image is tagged (details are in the docstring for the *dockerImageTag* parameter).

NOTE: This Fabric task is only run once regardless of the number of hosts/roles you supply.

Args:

gitRepository(str): The git repository to clone; this repository will be supplied to *git clone*

dockerImageName(str): Name of the Docker image in *namespace/image* format

branch(str, optional): The git branch of this repository we wish to build the Docker image from

gitRemotes(dict, optional): A dict where keys are git remote names and values are git remote urls. If supplied, the remotes listed in this dict will override any git remote of the same name in the cloned git repository used to build the Docker image. You should supply this parameter if all the following hold:

1. the *gitRepository* parameter is a path to a git repository on your local filesystem (the cloned repository's *origin* remote points to the git repository on your local filesystem)
2. the Dockerfile adds the cloned git repository
3. when the built Docker image is run, it invokes git to fetch / pull / perform some remote operation from the *origin* remote, which is the git repository on your local filesystem that the current git repository is cloned from. That repository most likely does not exist in Docker, hence the fetch / pull / other remote op fails.

gitSetUpstream(dict, optional): A dict where keys are local branch names and values are the upstream branch / remote tracking branch.

If you've supplied the *gitRemotes* parameter, you should supply this as well and add the local branch of interest as a key and its desired remote tracking branch as the corresponding value.

If supplied, the corresponding upstream branch will be set for the local branch using *git branch –set-upstream-to=upstream-branch local-branch* for existing local branches, or *git checkout -b upstream-branch local-branch* for non-existent branches.

Remote tracking branches must be specified in *remote/branch* format. You should supply this parameter if the following hold:

1. You supplied the *gitRemotes* parameter. This means that you are using a git repository on your local filesystem for the *gitRepository* parameter.
2. A git remote operation such as fetch / pull is run when the built Docker image is run.

   Suppose the branch being checked out in git repository inside the Docker is the *master* branch, and that your intention is to fetch updates from the *origin* remote and merge them into the *master* branch. Then you should supply a *{'master': 'origin/master'}* dict for this *gitSetUpstream* parameter so that the upstream branch / remote tracking branch of the *master* branch will be set to the *origin/master* branch (otherwise the *git pull* command will fail).

runGitCryptInit(bool, optional): If *True*, runs *git-crypt init* using the key specified by the *gitCryptKeyPath* parameter

gitCryptKeyPath(str, optional): Path to the git-crypt key for the repository; this must be given an existing path if the *runGitCryptInit* parameter is set to *True*

relativeDockerfileDirInGitRepo(str, optional): the directory inside the git repository that houses the Dockerfile we will be using to build the Docker image; this should be a path relative to the git repository. For instance, if the *base-image* directory within the git repository holds the Dockerfile we want to build, the *relativeDockerfileDirInGitRepo* parameter should be set to the string "base-image". Defaults to "." (the top level of the git repository).

dockerImageTag(str, optional): If supplied, the Docker image is tagged with it. Otherwise, a generated tag in the format *branch-first 12 digits in HEAD commit SHA1* is used. For instance, if *dockerImageTag* is not supplied, *branch* is "master" and its commit SHA1 is 18f450dc8c4be916fdf7f47cf79aae9af1a67cd7, then the tag will be *master-18f450dc8c4b*.

Returns: str: The tag of the Docker image

Raises:

ValueError: if *runGitCryptInit* is True, and either:

- the *gitCryptKeyPath* parameter is not given, or *None* is supplied
- the *gitCryptKeyPath* parameter is a non-existent path

viki.fabric.docker.**push_docker_image_to_registry**()

A Fabric task which **runs locally**; it pushes a local Docker image with a given tag to the Docker registry (http://index.docker.io).

NOTE: This Fabric task is only run once regardless of the number of hosts/roles you supply.

Args: dockerImageName(str): Name of the Docker image in *namespace/image* format

dockerImageTag(str, optional): Tag of the Docker image, defaults to the string "latest"

viki.fabric.docker.**build_docker_image_from_git_repo_and_push_to_registry**()

A Fabric task which **runs locally**; it builds a Docker image from a git repository and pushes it to the Docker

registry (http://index.docker.io). This task runs the *build_docker_image_from_git_repo* task followed by the *push_docker_image_to_registry* task.

NOTE: This Fabric task is only run once regardless of the number of hosts/roles you supply.

Args:

gitRepository(str): Refer to the docstring for the same parameter in *build_docker_image_from_git_repo* Fabric task

dockerImageName(str): Refer to the docstring for the same parameter in the *build_docker_image_from_git_repo* Fabric task

****kwargs: Keyword arguments passed to the *build_docker_image_from_git_repo*** Fabric task

Returns: str: The tag of the built Docker image

viki.fabric.docker.**pull_docker_image_from_registry**()

Pulls a tagged Docker image from the Docker registry.

Rationale: While a *docker run* command for a missing image will pull the image from the Docker registry, it requires any running Docker container with the same name to be stopped before the newly pulled Docker container eventually runs. This usually means stopping any running Docker container with the same name before a time consuming *docker pull*. Pulling the desired Docker image before a *docker stop docker run* will minimize the downtime of the Docker container.

Args: dockerImageName(str): Name of the Docker image in *namespace/image* format

dockerImageTag(str, optional): Tag of the Docker image to pull, defaults to the string *latest*

4.1.2 viki.fabric.helpers

viki.fabric.helpers.**run_and_get_output**(*cmdString*, *hostString=None*, *useSudo=False*, *captureStdout=True*, *captureStderr=True*)

Runs a command and grabs its stdout and stderr, without all the Fabric associated stuff and other crap (hopefully).

Args: cmdString(str): Command to run

hostString(str, optional): This should be passed the value of *env.host_string*

useSudo(bool, optional): If *True*, *sudo* will be used instead of *run* to execute the command

Returns:

dict: A Dict with 2 keys: "stdout": list(str) if captureStdout==True, *None* otherwise

"stderr": list(str) if captureStderr==True, *None* otherwise

```
>>> run_and_get_output("ls")
{ "stdout": ["LICENSE", "README.md", "setup.py"], "stderr": [] }
```

viki.fabric.helpers.**run_and_get_stdout**(*cmdString*, *hostString=None*, *useSudo=False*)

Runs a command and grabs its output from standard output, without all the Fabric associated stuff and other crap (hopefully).

Args: cmdString(str): Command to run

hostString(str, optional): This should be passed the value of *env.host_string*

useSudo(bool, optional): If *True*, *sudo* will be used instead of *run* to execute the command

Returns: list of str: List of strings from running the command

```
>>> run_and_get_stdout("ls")
["LICENSE", "README.md", "setup.py"]
```

viki.fabric.helpers.**get_home_dir**()

Returns the home directory for the current user of a given server.

Returns:

str: the path to the home directory of the current host, or the string "$HOME"

```
>>> get_home_dir()
"/home/ubuntu"
```

viki.fabric.helpers.**download_remote_file_to_tempfile**(*remoteFileName*)

Downloads a file from a server to a tempfile.NamedTemporaryFile.

NOTE: This function calls the *close* method on the NamedTemporaryFile.

NOTE: The caller is reponsible for deleting the NamedTemporaryFile.

Args: remoteFileName(str): name of the file on the server

Returns:

str: name of the temporary file whose contents is the same as the file on the server

```
>>> downloadedFileName = download_remote_file_to_tempfile(
      "/home/ubuntu/a/search.rb"
    )
>>> with open(downloadedFileName, "r") as f:
      # do some processing here...
>>> os.unlink(downloadedFileName) # delete the file
```

viki.fabric.helpers.**copy_file_to_server_if_not_exists**(*localFileName*, *serverFileName*)

Copies a file to the server if it does not exist there.

Args: localFileName(str): local path of the file to copy to the server

serverFileName(str): path on the server to copy to

```
>>> copy_file_to_server_if_not_exists("helpers.py",
      os.path.join("my-repo", "helpers.py")
    )
```

viki.fabric.helpers.**is_dir**(*path*)

Checks if a given path on the server is a directory.

Args: path(str): path we wish to check

Returns: bool: True if the given path on the server is a directory, False otherwise

```
>>> is_dir("/home/ubuntu")
True
```

viki.fabric.helpers.**update_package_manager_package_lists**()

Updates the package list of the package manager (currently assumed to be apt-get)

```
>>> update_package_manage_package_lists()
```

viki.fabric.helpers.**install_software_using_package_manager**(*softwareList*)

Installs a list of software using the system's package manager if they have not been installed. Currently this assumes *apt-get* to be the package manager.

Args: softwareList(list of str): list of software to install

```
>>> install_software_using_package_manager(
      ["vim", "openjdk-6-jdk", "unzip"]
    )
```

viki.fabric.helpers.**is_installed_using_package_manager**(*software*)

Determines if a given software is installed on the system by its package manager (currently assumed to be apt-get).

Args: software(str): The name of the software

Return:

bool: Returns True if the software is installed on the system using the package manager, False otherwise

```
>>> is_installed_using_package_manager("python")
True
```

viki.fabric.helpers.**setup_vundle**(*homeDir=None*)

Clones the Vundle vim plugin (https://github.com/gmarik/Vundle.vim) to the server (if it hasn't been cloned), pulls updates, checkout v0.10.2, and installs vim plugins managed by Vundle.

Args:

homeDir(str, optional): home directory for the server. If not supplied or if *None* is supplied, the return value of the *get_home_dir* function is used

```
>>> setup-vundle()
```

viki.fabric.helpers.**is_program_on_path**(*program*)

Determines if a program is in any folder in the PATH environment variable.

Args: program(str): Name of the program

Return:

bool: True if the program is in some folder in the PATH environment variable, False otherwise

```
>>> is_program_on_path("python")
True
```

viki.fabric.helpers.**install_docker_most_recent**()

Installs the most recent version of docker (https://www.docker.io) using the http://get.docker.io shell script, and adds the current user to the docker group.

NOTE: This function assumes that the bash shell exists, and that the user has sudo privileges.

viki.fabric.helpers.**get_return_value_from_result_of_execute_runs_once**(*retVal*)

Extracts one return value of a Fabric task decorated with *fabric.decorators.run_once* and ran with *fabric.tasks.execute*; this Fabric task should have the same return value for all hosts.

Refer to the **Example Script** in *viki.fabric.docker - A short guide* for an example of when to use this function.

Args:

retVal(dict): The return value of *fabric.tasks.execute(some_fabric_task, ...)*. *some_fabric_task* should be a Fabric task that only has local operations and is decorated with *fabric.decorators.runs_once*.

viki.fabric.helpers.**get_in_env**(*keyList*, *default=None*)

Obtains the value under a series of nested keys in *fabric.api.env*; the value of every key in *keyList* 9except for the final key) is expected to be a dict.

Args: keyList(list of str): list of keys under *fabric.api.env*

default(obj, optional): The default value to return in case a key lookup fails

```
>>> env
{'liar': {'pants': {'on': 'fire'}, 'cuts': {'tree': 'leaf'}}, 'feed': {'ready': 'roll'}}
>>> env_get_nested_keys(["liar"])
{'pants': {'on': 'fire'}, 'cuts': {'tree': 'leaf'}}
>>> env_get_nested_keys(["liar", "cuts"])
{'tree': 'leaf'}
>>> env_get_nested_keys(["feed", "ready"])
'roll'
>>> env_get_nested_keys(["feed", "ready", "roll"])
None
>>> env_get_nested_keys(["liar", "on"])
None
>>> env_get_nested_keys(["liar", "liar", "pants", "on", "fire"])
None
>>> env_get_nested_keys(["liar", "liar", "pants", "on", "fire"], "argh")
'argh'
```

viki.fabric.helpers.**get_in_viki_fabric_config**(*keyList*, *default=None*)

Calls *get_in_env*, but with the 0th element of the *keyList* set to *VIKI_FABRIC_CONFIG_KEY_NAME*.

Args:

keyList(list of str): list of keys under the *VIKI_FABRIC_CONFIG_KEY_NAME* key in *fabric.api.env*; **do not** include *VIKI_FABRIC_CONFIG_KEY_NAME* as the 0th element of this list

default(obj, optional): The default value to return in case a key lookup fails

```
>>> env
{'viki_fabric_config': {'hierarchy': {'of': 'keys', 'king': {'pin': 'ship'}}}}
>>> get_in_viki_fabric_config(["hierarchy"])
{"of": "keys", "king": {"pin": {"ship"}}}
>>> get_in_viki_fabric_config(["hierarchy", "of"])
'keys'
>>> get_in_viki_fabric_config(["hierarchy", "of", "keys"])
None
>>> get_in_viki_fabric_config(["hierarchy", "notthere"])
None
>>> get_in_viki_fabric_config(["hierarchy", "pin"])
None
>>> get_in_viki_fabric_config(["hierarchy", "pin"], "useThis")
'useThis'
```

viki.fabric.helpers.**env_has_nested_keys**(*keyList*)

Determines if *fabric.api.env* has a set of nested keys; the value of each key in *keyList* (except for the final key) is expected to be a dict

Args: keyList(list of str): list of keys under *env*

Returns:

bool: True if *fabric.api.env* contains the series of nested keys, False otherwise

```
>>> env
{'whos': {'your': 'daddy', 'the': {'man': {'not': 'him'}}}}
>>> env_has_nested_keys(['whos'])
True
>>> env_has_nested_keys(['your'])
```

```
False
>>> env_has_nested_keys(['whos', 'your'])
True
>>> env_has_nested_keys(['whos', 'your', 'daddy'])
False
>>> env_has_nested_keys(['whos', 'the', 'man'])
True
>>> env_has_nested_keys(['whos', 'man', 'not'])
False
```

4.1.3 viki.fabric.git

viki.fabric.git.**is_dir_under_git_control**(*dirName*)

Determines if a directory on a server is under Git control.

Args:

dirName(str): directory name on the server for which we wish to determine whether it's under Git control

Returns:

bool: True if the directory on the server is under Git control, False otherwise.

```
>>> is_dir_under_git_control("/home/ubuntu/viki-fabric-helpers")
True
```

viki.fabric.git.**setup_server_for_git_clone**()

Fabric task that sets up the ssh keys and a wrapper script for GIT_SSH to allow cloning of private Github repositories.

Args:

homeDir(str, optional): home directory for the server. If not supplied or if *None* is supplied, the return value of the *fabric_helpers.get_home_dir* function is used

For a Python Fabric script that imports the *viki.fabric.git* module using:

```
import viki.fabric.git
```

we can use this Fabric task from the command line, like so:

```
fab -H host1,host2,host3 viki.fabric.git.setup_server_for_git_clone
```

Alternatively, for a Python Fabric script that imports the *viki.fabric.git* module using:

```
import viki.fabric.git as fabric_git
```

we can use this Fabric task from the command like, like so:

```
fab -H host1,host2,host3 fabric_git.setup_server_for_git_clone
```

This function can also be called as a normal function (hopefully from within another Fabric task).

viki.fabric.git.**is_fabtask_setup_server_for_git_clone_run**(*homeDir=None*, *printWarnings=True*)

Determines if the *setup_server_for_git_clone* Fabric task has been run.

This task checks for the existence of some files on the server to determine whether the *setup_server_for_git_clone* task has been run.

Args:

homeDir(str, optional): home directory for the server. If not supplied or if *None* is supplied, the return value of the *fabric_helpers.get_home_dir* function is used

printWarnings(boolean): true if the *setup_server_for_git_clone* task has been run, false otherwise.

Returns:

bool: True if the *setup_server_for_git_clone* Fabric task has been run for the current server, False otherwise.

```
>>> is_fabtask_setup_server_for_git_clone_run()
False # along with some other output before this return value
```

viki.fabric.git.**get_git_ssh_script_path**(**args*, ***kwargs*)

Returns the path to the git ssh script

Args:

homeDir(str, optional): home directory for the server. If not supplied or if *None* is supplied, the return value of the *fabric_helpers.get_home_dir* function is used

Returns: str: the path to the git ssh script

```
>>> get_git_ssh_script_path()
"/home/ubuntu/git_ssh_wrap.sh"
```

viki.fabric.git.**local_git_branch_exists**(*branch*)

Determines if a branch exists in the current git repository on your local machine.

NOTE: The current working directory is assumed to be inside the git repository of interest.

Args: branch(str): Name of the branch

Returns: bool: True if the given branch exists, False otherwise

```
>>> local_git_branch_exists("master")
True
>>> local_git_branch_exists("non_existent_branch")
False
```

CHAPTER 5

Indices and tables

- *genindex*
- *modindex*
- *search*

V