How To Use Fabric To Automate Administration Tasks And Deployments

PostedJanuary 31, 2014 169.6k views Configuration Management Python Ubuntu

Introduction

Let's automate things. Everything.

Let's also figure out a way to do this using a single tool. One that is easy to program, easy to use. And why not do all this with nothing but SSH installed on the remote machine – all commands scripted at a single location for executing locally or on any number of various servers.

Does it not sound brilliant *and fab*? We agree.

In this DigitalOcean article, ***Fabric*** *– the [system] administration and application deployment streamlining library* – is our subject. We will learn how to install this wonderful tool and see just how easy things can become by simply automating mundane management tasks that would otherwise require jumping through hoops with bash hacks and hard-to-maintain, complex scripts.

Glossary

1. What Is Fabric?

1. Fabric and Python Programming Language
2. System/Server Administration
3. Application Deployment

2. How to Install Fabric on a Droplet Running Ubuntu / Debian

3. Fabric's Features and Integration with SSH

1. `run` *(fabric.operations.run)*
2. `sudo` *(fabric.operations.sudo)*
3. `local` *(fabric.operations.local)*
4. `get` *(fabric.operations.get)*
5. `put` *(fabric.operations.put)*
6. `prompt` *(fabric.operations.prompt)*
7. `reboot` *(fabric.operations.reboot)*

4. Fabric's Helpers: Context Managers

1. `cd` *(fabric.context_managers.cd)*
2. `lcd` *(fabric.context_managers.lcd)*
3. `path` *(fabric.context_managers.path)*
4. `settings` *(fabric.context_managers.settings)*

5. `prefix` *(fabric.context_managers.prefix)*

5. Example fabfile For Automating Management Tasks

What Is Fabric?

Fabric is a Python library (i.e. a tool to build *on*) used for interacting with SSH and computer systems [easily] to automate a wide range of tasks, varying from application deployment to general system administration.

Albeit being Python-based, it does not mean that it is used strictly for working with other Python applications or tools. In fact, Fabric is there for you to achieve just about anything regardless of a specific language or a system. As long as the very basic requirements are met, you can take advantage of this excellent library.

Fabric scripts are basic Python files. They are run using the `fab` tool that is shipped with with Fabric. All this does is include (i.e. `import ..`) your script (i.e. instructions to perform) and execute the provided procedure.

Say "Hello Fab!" using Fabric (`fabfile.py`):

```
# def hello(who="world"):
#    print "Hello {who}!".format(who=who)

$ fab hello:who=Fab
Hello Fab!
```

Fabric and Python Programming Language

As we have briefly mentioned above, although Fabric can be used in a very large scale of scenarios, it is a Python based library and *fabfile*s need to be programmed using the Python Programming Language.

Regardless of your experience with any other programming language (including Python), as you make your way through our Fabric articles, you will learn how to work with this tool and it shall not take you long to see how simplistic and wonderful it is.

Python is an extremely popular, widely adopted general purpose (i.e. not created to solve a specific problem) programming language. It can be easily distinguished by the importance it puts on code readability and simplicity. To understand Python, check out the very short Python Enhancement Proposal (PEPs) 20 **The Zen of Python**, followed by bit long Style Guide for Python Code.

In order to have an overall understanding of what programming in Python might be like, you can read a few of the great articles listed at Python Beginner's Guide.

System/Server Administration

One of the key areas for using Fabric is automating the everyday tasks of system (and server) administration. These jobs include pretty much everything that relates to:

- Building a server;

- Its maintenance, and;
- Monitoring.

When you begin working with your very own droplet (which is a fully-fledged virtualised server with full control / access), things that appear as a mystery will quickly start to become familiar to you. As you deploy your applications and start dealing with their maintenance, it is only natural to expect that you will be running into some issues. However, when your application gains popularity and things start to grow, the need of managing multiple droplets and repeating everything over and over again ceases to become fun.

That is exactly when you will wish you had met Fabric years ago.

Application Deployment

Deploying an application (regardless of it being a web site, an API, or a server) usually means setting up a system from scratch (or from a snapshot taken in time), preparing it by updating everything, downloading dependencies, setting up the file structure and permissions, followed by finally uploading your codebase - or downloading it using a SCM such as Git.

During the development process, you are also likely to have commands that need to be routinely executed (ex: right before entering a deployment cycle).

Being able to script these tasks (both local and remote) in a logically organized and -- most importantly -- programmable manner proves to be invaluable shortly after you realize how much time is being wasted repeating the same steps constantly, rendering everything error-prone during the process.

This is exactly when Fabric comes to your aid in the form of a Python file that will know *what* to do and *where* to do it.

How to Install Fabric on a Droplet Running Ubuntu / Debian

An easy and cohesive way of installing Fabric is by using the default operating system package manager `aptitude`.

In order to install Fabric using `aptitude`, run the following:

```
sudo aptitude install fabric

# Alternatively, you can also use *pip*:
# pip install fabric
```

Fabric's Features and Integration with SSH

Out of the box, any Python command (or procedure) and module can be utilised through Fabric - given that Fabric is indeed a Python library.

What Fabric really brings to the table is its extensive and excellent integration with SSH that allows streamlining everything using simple scripts (i.e. `fabfile.py`).

In this section, you can find a selection of tools (e.g. functions) that come with Fabric which can be used to interact with environments where commands you specify are executed.

Note: You can see and learn more about Fabric's operations by visiting its documentation on the subject.

run (fabric.operations.run)

Fabric's `run` procedure is used for executing a shell command on one or more remote hosts.

- The output results of *run* can be captured using a variable.
- If command succeeded or failed can be checked using `.failed` and `.succeeded`.

Usage examples:

```
# Create a directory (i.e. folder)
run("mkdir /tmp/trunk/")

# Uptime
run("uptime")

# Hostname
run("hostname")

# Capture the output of "ls" command
result = run("ls -l /var/www")

# Check if command failed
result.failed
```

sudo (fabric.operations.sudo)

Along with `run`, the most widely used Fabric command is probably `sudo`. It allows the execution of a given set of commands and arguments with sudo (i.e. *superuser*) privileges on the remote host.

If sudo command is used with an explicitly specified user, the execution will happen not as root but another (i.e. UID 1010).

Usage examples:

```
# Create a directory
sudo("mkdir /var/www")

# Create a directory as another user
sudo("mkdir /var/www/web-app-one", user="web-admin")

# Return the output
result = sudo("ls -l /var/www")
```

local (fabric.operations.local)

As we have mentioned in our introduction, a single Fabric script (fabfile) can be used to perform actions both on the local machine and remote system(s). For this purpose, Fabric provides the `local` operative to run commands locally.

Unlike run or sudo, however, interacting with the output of `local` the same way is not possible. Either output can be captured or printed -- the switch can be set with `capture` argument.

Local helpers such as the `lcd` context manager (which is used for setting the **local** working directory) are *honoured* with `local`, the same way `run` (or `sudo`) honours the `cd` context manager.

Usage examples:

```
# Create a source distribution tar archive (for a Python App.)
local("python setup.py sdist --formats=gztar", capture=False)

# Extract the contents of a tar archive
local("tar xzvf /tmp/trunk/app.tar.gz")

# Remove a file
local("rm /tmp/trunk/app.tar.gz")
```

get (fabric.operations.get)

The `get` command exists to download (i.e. pull) file(s) from the remote system to the computer where the Fabric is being used. It is similar to how `scp` works and comes in handy when you need to download backups, logging data or some other server related items.

- You can specify the remote path with the `remote_path` argument.
- You can specify the local - download - path with the `local_path` argument.

Usage examples:

```
# Download some logs
get(remote_path="/tmp/log_extracts.tar.gz", local_path="/logs/new_log.tar.gz")

# Download a database back-up
get("/backup/db.gz", "./db.gz")
```

put (fabric.operations.put)

When you need to upload files, `put` command can be used very similarly to `get`. You can again access the results of command's execution with `.failed` or `.succeeded`.

- `local_path` - set the local path.
- `remote_path` - set the remote path.
- `use_sudo` - upload the file to anywhere on the remote machine using a nifty trick: upload to a temporary location then move.
- `mode` - set the file mode (flags).
- `mirror_local` - set the file flags (i.e. make executable) automatically by reading the local file's mode.

Usage examples:

```
# Upload a tar archive of an application
```

```
put("/local/path/to/app.tar.gz", "/tmp/trunk/app.tar.gz")

# Use the context manager `cd` instead of "remote_path" arg.
# This will upload app.tar.gz to /tmp/trunk/
with cd("/tmp"):
    put("local/path/to/app.tar.gz", "trunk")

# Upload a file and set the exact mode desired
upload = put("requirements.txt", "requirements.txt", mode=664)

# Verify the upload
upload.succeeded
```

prompt (fabric.operations.prompt)

When you find yourself in need of some extra flexibility working with Fabric, `prompt` will come to your rescue. This command does exactly what its name suggests and asks the user (i.e. one that is running the script) to input a certain data to use during the successive execution.

If you are using a single file to manage with multiple applications, for example, you can use `prompt` to set one to perform the actions.

Before starting with anything, `prompt` can also be used to query the port number to use.

Usage examples:

```
# Prompt the user
port_number = prompt("Which port would you like to use?")

# Prompt the user with defaults and validation
port_number = prompt("Which port?", default=42, validate=int)
```

reboot (fabric.operations.reboot)

The `reboot` command is also self explanatory: it is used to reboot the remote system. By default, it waits two minutes (i.e. 120 seconds -> `wait=120`) before doing its job.

Usage examples:

```
# Reboot the remote system
reboot()

# Reboot after 30 seconds
reboot(wait=30)
```

Fabric's Helpers: Context Managers

Fabric's *context managers* are used with the Python's `with` statement. The reason for this is how sessions between execution of commands are **not** kept between *shell-less* connections.

Note: You can see and learn more about Fabric's context managers by visiting its documentation on the subject.

cd (fabric.context_managers.cd)

`cd` context manager allows keeping the directory state (i.e. where the following block of comments are to be executed). It is similar to running the *cd* command during an SSH session and running various different commands.

Usage examples:

```
# The *cd* context manager makes enwrapped command's
# execution relative to the stated path (i.e. "/tmp/trunk")
with cd("/tmp/trunk"):
    items = sudo("ls -l")

# It is possible to "chain" context managers
# The run commands gets executed, therefore at "/tmp/trunk"
with cd("/tmp"):
    with cd("/trunk"):
        run("ls")
```

lcd (fabric.context_managers.lcd)

The `lcd` context manager (local cd) works very similarly to one above (cd); however, it only affects the local system's state.

Usage examples:

```
# Change the local working directory to project's
# and upload a tar archive
with lcd("~/projects/my_project"):
    print "Uploading the project archive"
    put("app.tar.gz", "/tmp/trunk/app.tar.gz")
```

path (fabric.context_managers.path)

`path` context managers alters the PATH variable.

settings (fabric.context_managers.settings)

When you need to temporarily (i.e. for a certain command chain), you can use the `settings` statement (i.e. override `env` values).

Usage examples:

```
# Perform actions using a different *user*
with settings(user="user1"):
    sudo("cmd")
```

prefix (fabric.context_managers.prefix)

`prefix` statement does what its name suggests and wraps given `run` and `sudo` command with the specified one.

Usage examples:

```
with prefix("cmd arg."):
    run("./start")
# cmd arg. && ./start
```

Example fabfile For Automating Management Tasks

To begin learning how to program a *fabfile*s to automate a simple management task, let's create an empty `fabfile.py`.

Run the following command to create a `fabfile.py` using the text editor nano:

```
nano fabfile.py
```

Append the following code block updating the system and installing memcached:

```
# Fabfile to:
#    - update the remote system(s)
#    - download and install an application

# Import Fabric's API module
from fabric.api import *


env.hosts = [
    'server.domain.tld',
  # 'ip.add.rr.ess
  # 'server2.domain.tld',
]
# Set the username
env.user   = "root"

# Set the password [NOT RECOMMENDED]
# env.password = "passwd"

def update_upgrade():
    """
        Update the default OS installation's
        basic default tools.
                                        """
    run("aptitude    update")
    run("aptitude -y upgrade")

def install_memcached():
    """ Download and install memcached. """
    run("aptitude install -y memcached")

def update_install():

    # Update
    update_upgrade()

    # Install
    install_memcached()
```

Save and exit using CTRL+X and confirm with with Y.

Now you can start automating your mundane server management tasks using Fabric and its features explained here.

```
# Automate everything!
fab update_install
```