fabobjects Documentation

Release 0.0.1

Abiola Rasheed

Feb 21, 2019

Contents

fab-objects is a high-level Python library useful for performing common Unix system administration task, by providing a set of Python Classes for executing shell commands on a remote system via SSH. It presents a unified, familiar API that allows you to logically plan deployments and maintenance.

fab-objects is a light-weight wrapper around the Fabric library.

Note: This documentation assumes you have some python knowledge, that you are running python3.6 or above and that your ssh key is located in `~/.ssh` directory.

---

fab-objects in action:

```
>>> from os import environ
>>> from fabobjects import Ubuntu
>>> from myapp.conf import server_config

>>> # Create an ubuntu server instance
>>> ubuntu_server = Ubuntu(**server_config)

>>> # Update the server
>>> ubuntu_server.update()

>>> # Reboot the server
>>> ubuntu_server.rebootall()

>>> # Install your application
>>> ubuntu_server.install_package('postgresql-9.6')
```

We can run this same code with a different OS by calling the same method on the distro instance and all should just work fine. For example on a FreeBSD:

```
>>> from os import environ
>>> from fabobjects import FreeBSD
>>> from . import server_config

>>> # Create a freebsd server instance
>>> free_bsd_server = FreeBSD(**server_config)

>>> # Update the server
>>> free_bsd_server.update()

>>> # Reboot the server
>>> free_bsd_server.rebootall()

>>> # Install your application
>>> free_bsd_server.install_package('postgresql-9.6')
```

CHAPTER 1

User Guide

Welcome! This tutorial walks you through some common use case and core functionalities of **fab-objects**; for further details, see the API doc sections.

1.1 Installation

Example:

```
pip install git+https://github.com/abiolarasheed/fabobjects.git
```

1.2 Initialize new server with `root` user

For the most part when using Fab-Objects you will create a server instance from one of the built in destros and execute a shell commands by calling a method on that instance. By default, some cloud providers will give you a root user and an ip address to the server you have created, for this example we will assume you have created an `Ubuntu 16.10 x64` with Public IP `199.199.199.99`. Now let ssh into this box and set it up using `root@199.199.199.99`.

```
>>> from os import environ
>>> from fabobjects import Ubuntu

>>> workstation_ip = "58.588.588.58"

>>> ubuntu_server = Ubuntu(hostname="db1", user="root",
...                        domain_name="mydomain.com",
...                        email="yoursysadmin@mydomain.com",
...                        ip='199.199.199.99',
...                        password=environ.get("PASSWORD"),
...                        user_ip=workstation_ip,
...                        )
```

(continues on next page)

(continued from previous page)

```
>>> # 1st lets create an admin user
>>> admin_user = "ubuntu"
>>> ubuntu_server.create_admin_account(admin_user,
...                                    environ.get("PASSWORD"))

>>> # Update and secure the server
>>> ubuntu_server.harden_server()
>>> # Root login disabled, and server secured

>>> # Now login with ubuntu@199.199.199.99 and reboot
>>> ubuntu_server.rebootall()

>>> # Install your application
>>> ubuntu_server.install_package('postgresql-9.6')
```

On some cloud providers an admin user is provided other than the `root` or if you installed the OS your self there is a chance you have an admin user already. See the table below for *aws* default users on some OS.

OS/Distro	Official AMI ssh Username
Amazon Linux	ec2-user
Ubuntu	ubuntu
Debian	admin
RHEL 6.4 and above	ec2-user

Initialize new server with admin user

```
>>> from os import environ
>>> from fabobjects import Ubuntu

>>> admin_user = "ubuntu"
>>> workstation_ip = "58.588.588.58"

>>> ubuntu_server = Ubuntu(hostname="db1", user=admin_user,
...                        domain_name="mydomain.com",
...                        email="yoursysadmin@mydomain.com",
...                        ip='199.199.199.99',
...                        password=environ.get("PASSWORD"),
...                        user_ip=workstation_ip,
...                        )

>>> # Update and secure the server
>>> ubuntu_server.harden_server()

>>> # Root login disabled, and server secured
>>> ubuntu_server.rebootall()

>>> # Install your application
>>> ubuntu_server.install_package('postgresql-9.6')
```

1.3 Running custom shell commands

While fab-objects has a bunch of useful built-in methods, there are times when you want to run a custom command as the `superuser` or as another user or just as your self(current user). To do this fab-objects exposes 3 methods, `sudo`,

`run` and `run_as_user` for executing shell commands on your remote system as below.

Using the server from previous samples:

```
>>> ubuntu_server.user # just making sure we are the admin user
'ubuntu'

>>> ubuntu_server.run("id")  # Run command as current user
'uid=1000(ubuntu) gid=1000(ubuntu) groups=1000(ubuntu),...'

>>> # Run command as another user(postgres user)
>>> ubuntu_server.run_as_user("id", user='postgres')
'uid=116(postgres) gid=122(postgres) groups=122(postgres),...'

>>>  # Run command with sudo
>>> ubuntu_server.sudo('grep postfix /etc/passwd | cut -d: -f6')
'/var/spool/postfix'
```

1.4 Managing Firewalls

Firewalls are one of the many layers of security one can use to secure a server, fab-objects ships with a few firewall method. below are some ways of setting up and managing your firewalls:

```
>>> # Install firewall and allow user_ip in on ssh
>>> ubuntu_server.install_firewall(user_ip="58.588.588.58")

>>> # Allow all on http
>>> ubuntu_server.firewall_allow_form_to(host="all",
...                                      proto='tcp',
...                                      port=80)

>>> # Allow more than one rule
>>> rules = ["ufw allow https", "ufw allow postgres"]
>>> ubuntu_server.configure_firewall(rules=rules)

>>> # View current firewall rules
>>> print(ubuntu_server.view_firewall_rules())

     To          Action      From
     --          ------      ----
[ 1] 22          LIMIT IN    58.588.588.58
[ 2] 80/tcp      ALLOW IN    Anywhere
[ 3] 443/tcp     ALLOW IN    Anywhere
[ 4] 5432/tcp    ALLOW IN    Anywhere

>>> # Delete rule 4 (postgres)
>>> ubuntu_server.delete_firewall_number(4)

>>> # View status and rules
>>> print(ubuntu_server.firewall_status())

Status: active
Logging: on (low)
Default: deny (incoming), allow (outgoing), disabled (routed)
New profiles: skip
     To          Action      From
```

(continues on next page)

(continued from previous page)

```
     --          ------      ----
[ 1] 22          LIMIT IN    58.588.588.58
[ 2] 80/tcp      ALLOW IN    Anywhere
[ 3] 443/tcp     ALLOW IN    Anywhere
```

1.5 File Handling

Another common task you may often want to perform over SSH are file manipulation, creation and transfer. Fab-object exposes a few method to cover most use case and here are a few:

```
>>> ubuntu_server.put  # Upload files
>>> ubuntu_server.get  # Download files

>>> # Compress/Uncompress files
>>> ubuntu_server.compress    # Compress files
>>> ubuntu_server.uncompress  # Uncompress files

>>> # Comment/Uncomment/Append/Replace
>>> ubuntu_server.comment
>>> ubuntu_server.uncomment
>>> ubuntu_server.append
>>> ubuntu_server.sed

>>> # Checks if a file exist on the remote server
>>> ubuntu_server.exists

>>> # Checks if a folder exist on the remote server
>>> ubuntu_server.dir_exists(location)

>>> # Create folder if it does not exist on the remote server
>>> ubuntu_server.dir_ensure(location, recursive=False)

>>> # Creates file/folder or Delete file/folder on the remote server
>>> ubuntu_server.make_or_del

>>> # Create symbolic link
>>> ubuntu_server.create_symbolic_link
```

To see the usage and parameters of all this methods please see the API documentation.

1.6 Application Management

Installing, starting, stopping, reloading and uninstalling are some of the every day task you'll often want to perform on your remote host, doing it with fab-objects makes it simple and os independent. For example, let install and manage `nginx` http-server:

```
>>> # Check if nginx is installed
>>> ubuntu_server.is_package_installed('nginx')
False

>>> # Install nginx
>>> ubuntu_server.install_package('nginx')
```

(continues on next page)

(continued from previous page)

```
>>> # Start Nginx
>>> ubuntu_server.service_start('nginx')
>>> # Nginx is ready to start accepting requests

>>> # Stop nginx
>>> ubuntu_server.service_stop('nginx')

>>> # You can also Restart(stop/start) nginx
>>> ubuntu_server.service_restart('nginx')

>>> # Reload after changing nginx config
>>> ubuntu_server.service_reload('nginx')

>>> # Check nginx status
>>> ubuntu_server.service_status('nginx')

>>> # uninstall nginx if you are an apache guy
>>> # ubuntu_server.uninstall_package('nginx')
```

1.7 Bring it all together

Now that we have installed nginx and its up and running just fine, lets setup our custom domain and upload our ssl certs so that our site can now run using our site domain name over `https`:

```
>>> from os.path import as pjoin

>>> # SSL settings
>>> local_ssl_tar_file = "ssl.tar.gz"
>>> remote_ssl_dir = "/etc/nginx/ssl/"
>>> remote_tmp_ssl_dir = "/tmp/ssl/"
>>> remote_ssl_tar_file = pjoin(remote_tmp_ssl_dir,
...                             local_ssl_tar_file)

>>> # Create tmp folder(/tmp/ssl) to hold our cert
>>> ubuntu_server.dir_ensure(remote_tmp_ssl_dir)

>>> # Create final folder(/etc/nginx/ssl) to hold our cert
>>> ubuntu_server.dir_ensure(remote_ssl_dir)

>>> # Upload ssl cert to our server's /tmp/ssl
>>> ubuntu_server.put(local_path=local_ssl_tar_file,
...                   remote_path=remote_tmp_ssl_dir,
...                   use_sudo=True)

>>> # Uncompress ssl tar file and place content in nginx dir
>>> ubuntu_server.uncompress(remote_ssl_tar_file,
                             output_dir=remote_ssl_dir)

>>> # Clean up by deleting the tar file
>>> ubuntu_server.make_or_del(remote_ssl_tar_file,
...                           make=False,
...                           use_sudo=True)
```

From the code above we uploaded our ssl cert from our workstation to our remote server, then placed it in a location

where nginx can begin to use it.

Next we will create a configuration for our site and load it so nginx knows where to server our site from lets:

```
>>> # mydomain settings

>>> local_zip_file = "mydomain.com.conf.zip"

>>> remote_tmp_zip_file = pjoin("/tmp", local_zip_file)
>>> remote_config_file = '/etc/ngix/sites-available/mydomain.com.conf'
>>> remote_enable_file = '/etc/ngix/sites-enable/mydomain.com.conf'

>>> # Upload config for mydomain.com
>>> ubuntu_server.put(local_path=local_zip_file,
...                   remote_path="/tmp/",
...                   use_sudo=True)

>>> # Unzip config file
>>> ubuntu_server.uncompress(remote_tmp_zip_file, file_type="zip")

>>> # Clean up by deleting the zip file now after unzipping it
>>> ubuntu_server.make_or_del(remote_tmp_zip_file, make=False, use_sudo=True)

>>> # Enable Nginx
>>> ubuntu_server.create_symbolic_link(remote_config_file, remote_enable_file)

>>> # Reload new nginx config
>>> ubuntu_server.service_reload('nginx')
```

1.8 Complete!

This has been a minimal walk through of the fab-objects API, for a more complete list of methods see the API docs for more information.

CHAPTER 2

Built-in Applications

Installing and configuring applications are sometimes complex and hard, but they also require the developer to login via SSH to do so manual labor. The application class helps limit this and often just require the developer to call a class method to perform the remote task. Application classes are simple and flexible to use or extend.

2.1 Managing Redis

For this example we will install and configure redis on our server:

```
>>> from fabobjects import RedisServer
>>> from os import environ

>>> ubuntu_server = Ubuntu(**config_dict)

>>> # Redis configuration
>>> redis_config = {'maxmemory':'256mb', 'public':False}

>>> # Create redis instance
>>> redis = ubuntu_server.create_app(RedisServer, redis_config)

>>> # Install and configure redis
>>> redis.deploy()

>>> # To send commands to redis server
>>> redis.redis_cli("PING")
"PONG"
```

Note: You have to pass the application class and the application config to the server instance `create_app` method to instantiate an application. This gives the application all the server settings, along with all the server methods too.

Example: **redis = ubuntu_server.create_app(RedisServer, redis_config)**

The sample above is simple and often perfect for situations where the application storing and retrieving data from and to the redis-server lives on the same host as the redis-server. But when your user base increases you will want to place your redis-server on a different host, and have your applications connect to it **securely** over ssl with password authentication. In our next example we will set up a redis-server that can accepts ssl connection with ssl cert and password authentication:

```
>>> from fabobjects import Debian, Ubuntu
>>> from myapp.conf import config_dict, app_config_dict

>>> # Create our servers
>>> cache_server = Ubuntu(**config_dict)  # private '10.10.10.1'
>>> flask_app_server = Debian(**app_config_dict) # private '10.10.10.2'
```

The code above shows our two servers we created, one will hold our flask app , the other will hold our redis-server. Next we configure the redis-server to listen and accept connection on its internal ip('10.10.10.1') over ssl but with password authentication.:

```
>>> # Redis configuration
>>> redis_config = {"maxmemory": "256mb",
...                 "exposed_ip": "10.10.10.1",
...                 "redis_password": environ.get("REDIS_PASSWORD"),
...                 "allowed_ip": "10.10.10.2",
...                 "public": False}

>>> # instantiate a redis-server
>>> redis = cache_server.create_app(RedisServer, redis_config)

>>> # configure and install redis-server
>>> redis.deploy()

>>> # Redis ssl configuration used to generate self signed ssl cert
>>> ssl_conf = {"domain": "example.com", "country_iso": "US",}
...             "state": "California", "city": "San Francisco",
...             "company_name":"Example Inc")

>>> # Configure ssl for redis
>>> redis.enable_ssl(**ssl_conf)

>>> # To send commands to redis server
>>> redis.redis_cli("PING")
"PONG"
```

Next we setup our redis client on our other server, so it can communicate with the redis-server over ssl connecting using our internal next work ip addresses for added security by not totaly exposing redis to public internet. Then we will open our firewalls to allow connection from the client to the server:

```
>>> # Download the private.pm from the redis-server to our local workstation
>>> # The private.pem is downloaded to our /tmp/private.pem
>>> # This will be used to encrypt communication by both servers

>>> cache_server.get("/etc/redis/ssl/certs/private.pem",
...                  local_path="/tmp/", use_sudo=True)

>>> redis_client_config = {"redis_password": environ.get("REDIS_PASSWORD"),
...                        "server_ip": "10.10.10.1",
...                        "server_cert": "/tmp/private.pem"
...                        }
```

(continues on next page)

(continued from previous page)

```
>>> redis_client = flask_app_server.create_app(RedisSslClient, redis_client_config)

>>> # Upload the private.pem to our flask server now then set it all up
>>> redis_client.deploy()

>>> # set key from client
>>> redis_client.redis_cli("SET greetings 'Hello'")
"OK"

>>> # test redis server if key exist
>>> redis.redis_cli("GET greetings")
"Hello"
```

2.2 Managing PostgreSQL

PostgreSQL, is an object-relational database management system often used in many python applications. Configuring and managing postgres is relatively simple with fab-objects:

```
>>> from fabobjects import PostgresServer

>>> # Note if gis_version is not set to None PostGis will we installed along with␣
↪Postgres
>>> db_config = {"db_pass":"password1", "db_name":"testdb1", "db_user":"test_db_user"
...               "gis_version": None}
>>>
>>> postgres = ubuntu_server.create_app(PostgresServer, db_config)

>>> # install and configure postgres
>>> postgres.deploy()
>>> # Please note that the db_user will be granted ``All`` permissions on db
```

Note: Please note that postgres 9.5 and postgis 2.2 are installed by default, and you can change this passing the version you want to the constructor.

Example: db_config = {..., 'gis_version': '2.3', 'db_version':'9.6'}

Now that we have postgres up and running lets run some commands and see how things work:

```
>>> db_name = "testdb2"
>>> db_user = "test_db_user2"
>>> passwd = "password2"

>>> # Create a database
>>> postgres.create_db(dbname=db_name)

>>> # Create a new db user
>>> postgres.create_db_user(user=db_user, passwd=passwd)

>>> # Grant user permission
>>> postgres.grant_permission(permission_type='SELECT', db=db_name,
...                           role_name=db_user)
```

(continues on next page)

(continued from previous page)

```
>>> # Run SQL commands
>>> postgres.psql("SELECT * FROM books WHERE book_id >= 100 ORDER BY book_id ASC;")

>>> # Run Shell Command with user postgres
>>> postgres.postgres_run('touch /tmp/postgres.txt')

>>> # Clone / backup / restore settings
>>> remote_host = "155.155.155.55"
>>> remote_host_user = "db_user"
>>> remote_db_name = "test1"
>>> local_host_user = "am_local"
>>> backup_filename = "backup_filename.sql"

>>> # Clone a remote db
>>> postgres.clone_db(remote_host, remote_host_user, remote_db_name, local_host_user)

>>> # Backup a db
>>> postgres.backup(remote_db_name, backup_filename)

>>> # Restore a db
>>> postgres.restore(remote_db_name, filename)

>>> # Set Up daily backups
>>> password = "somepasswords"
>>> postgres.set_daily_backup(password)
```

Installing PostGIS and PostgreSQL

PostGis is installed by default, except if you turn it off when initializing your app by setting `gis_version = None`:

```
>>> from fabobjects import Ubuntu, PostgresServer
>>> from myapp import server_config

>>> ubuntu_server = Ubuntu(**server_config)

>>> db_config = {"db_pass":"password1", "db_name":"testdb1", "db_user":"test_db_user"}

>>> # By default postgres-9.5 gets installed with postgis-2.2
>>> postgres_n_gis = ubuntu_server.create_app(PostgresServer, db_config)

>>> # To install a specific gis version
>>> db_config = {"db_pass":"password1", "db_name":"testdb1", "db_user":"test_db_user",
...              "db_version": "9.6", "gis_version": "2.3"}

>>> # This will install postgres-9.6 gets installed with postgis-2.3
>>> postgres_n_gis = ubuntu_server.create_app(PostgresServer, db_config)
```

Now postGIS is installed along with postgreSQL and enabled. You can begin to create your geographic objects and run location queries in SQL.

Postgresql Replication:

```
>>> from fabobjects import PostgresServer, PostgresServerReplica

>>> # server config
```

(continues on next page)

(continued from previous page)

```
>>> pry_db_config = {}
>>> replica_config = {}
```

2.3 Managing Nginx

2.4 Managing Git Remote Server

2.5 Managing Python Applications

2.6 Bring it all together

2.7 Creating your own Application!

This has been a minimal walk through of the fab-objects API, for a more complete list of methods see the API docs for more information.

CHAPTER 3

The API Documentation

If you are looking for information about the API details on a specific class, method or function, checkout our API documentation:

3.1 distros

class fabobjects.distros.**BSD**(**args*, ***kwargs*)

get_package_manager()
This will return the name of package manager on given os, example ubuntu should return apt or apt-get. :return: string

class fabobjects.distros.**BaseServer**(**args*, ***kwargs*)
An SSH daemon connection. **Basics** This class performs useful high level operations over ssh.

add_app(*app*)
Add and app to list of apps installed on server. :param app: App instance :return: None

compress(*input_file*, *output_file=None*, *file_type='tar'*)
Compress tar or zip files or folder :param input_file: The input file :param output_file: The output file :param file_type: type if its zip or tar :return: None

create_symbolic_link(*link_from*, *link_to*)
Create a symbolic link :param str link_from: The initial link :param link_to: File/fold linked to :return: None

deploy_all()
Deploy all app added to server :return: None

get_package_manager()
This will return the name of package manager on given os, example ubuntu should return apt or apt-get. :return: string

getattribute(*func*, **args*, ***kwargs*)
This method helps you call other instance methods and can accept args and kwargs mostly good for call private variables and methods. :param func: function :param args: :param kwargs: :return: function or None

list_apps()
List apps deployed on server. :return:

static local(*command*, *capture=True*, *shell=None*)
Run a command on the local system.

remove_app(*app*)
Remove an app from list of apps deployed on server. :param app: App instance :return: None

run_as_app_user(**args*, ***kwargs*)
This should be implemented by apps inheriting this server class.

uncompress(*file_path*, *output_dir=None*, *file_type='tar'*)
Uncompress tar or zip files or folder :param file_path: The input file :param output_dir: The output file :param file_type: type if its zip or tar :return: None

class fabobjects.distros.**CentOS**(**args*, ***kwargs*)

class fabobjects.distros.**Debian**(**args*, ***kwargs*)

get_package_manager()
This will return the name of package manager on given os, example ubuntu should return apt or apt-get. :return: string

class fabobjects.distros.**FreeBsd**(**args*, ***kwargs*)

class fabobjects.distros.**RedHat**(**args*, ***kwargs*)

get_package_manager()
This will return the name of package manager on given os, example ubuntu should return apt or apt-get. :return: string

class fabobjects.distros.**Ubuntu**(**args*, ***kwargs*)

fabobjects.distros.**shell_safe**(*path*)
Makes sure that the given path/string is escaped and safe for shell :param string path: :return: string

3.2 redis

class apps.redis.**RedisApp**(**args*, ***kwargs*)
A Redis class that defines a set of methods that's used by both redis server and redis client class.

redis_cli(*command*)
Run redis command :param str command: The command you want to pass to redis :return:

class apps.redis.**RedisServer**(**args*, ***kwargs*)
A Redis server class, this class installs and sets up a redis server on a host server.

enable_ssl(*domain=None*, *country_iso=None*, *state=None*, *city=None*, *company_name=None*)
Enable ssl connection on redis server :param str domain: The website domain name :param str country_iso: :param str state: :param str city: :param str company_name: :return: None

reload()
A method for reload the redis. :return: None

restart()
A method for restarting redis. :return: None

start()
A method for starting redis. :return: None

status()
A method for restarting the application. :return: None

stop()
A method for stopping redis. :return: None

class apps.redis.**RedisSslClient**(**args*, ***kwargs*)
A Redis ssl client, this class installs and sets up a redis client to talk to a remote redis server over an ssl connection.

deploy()
Install and set up a redis client on a host server. :return:

3.3 utils

class fabobjects.utils.**ServerHostManager**(*instance_method*)
A descriptor decorator that decorates an object and exposes the instance that is decorated.

fabobjects.utils.**log_call**(*func*)
Logs any callable

fabobjects.utils.**random_password**(*bit=12*)
Generates a random password which include numbers, letters and special characters.

fabobjects.utils.**return_distinct_servers**(*servers*)
Returns only one instance of a server or application. This will not guarantee that the server you want is the one you will get, it will only get them in order.

fabobjects.utils.**server_host_manager**
alias of *fabobjects.utils.ServerHostManager*

fabobjects.utils.**timing**(*func*)
Times function call.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

f

Index