deploy-commander Documentation

Release 0

Ference van Munster

September 01, 2016

Stuff you'll need to know for deploy commander.

Not familiar with deploy-commander? Read this.

CHAPTER 1

Contents

1.1 What is deploy commander?

A tool mainly used for remotely executing sysadmin tasks. As the name might give you a clue... it's power lies with managing deployments/software builds for continuous integration flows.

Deploy commander is build directly on python/fabric, but can be used for any project.

Main goal of this application is to configure your deployment by using a generic set of predefined actions.

There are other flavors for setting up your dtap... we're focussing on simplicity and transparancy in a centralized application to manage *different deployments* (applications) for *multiple apps* build in *different languages* on *different servers*.

We're building a lot if handy features for sysadmins and developers that will make your life easier.

Stuff like:

- Git checkout and update
- Create/ensure folders dynamicly
- Backup database on deployment
- Import database
- Download files (like assets) for backup.
- Upload files (like assets) for preperation
- Upload templates for dynamic settings

Note: This application is still in it's development phase.

1.1.1 Integration with other systems

We're still in beta mode, but we've released a initial version for integration with bitbucket.

See the integration section for more info.

1.1.2 What it's not!

- Puppet, (can be used for setting up server software but verry handy in your project!)
- Chef (See Puppet)
- Sys ops server management (See puppet/Chef)
- Jenkins (CI Server, we might build an equivelent :))
- Capistrano (Allso used for deployments... but mostly used per project/site...)

1.1.3 Cases

- You would like to setup a project on different ubuntu machines.
- Download assets from server x, and upload them to server y.
- Deploy a new version of your app to a server, and backup the database before running migrations
- More...

1.1.4 Main goals

- Centralized deployment
- Ease of deployments
- Ease of sysadmin tasks
- Simplified deployment by simple configurations
- Unix based environments
- Continuous integration

1.2 deploy-commander installation

Deploy commander is easy to install and configure. Here we'll explain the basic installation.

If you're completely new to python, pip and or virtualenv please follow the tuturials

1.2.1 Prerequisites

- Python 2.7

(others not tested yet)

Other libraries will be installed by the pip installer:

- pycrypto==2.6.1
- ecdsa
- jinja2
- fabric==1.10.0
- simple-crypt==4.0.0

- cython==0.22
- falcon==0.1.10
- gunicorn==19.2.1

1.2.2 Best practices

- In production encrypt the config files.
- Make sure the user home directory (best to use deploy user) is encrypted/secured
- Make sure if you make a backup of copy on your dev, that it's encrypted.
- For production use a dedicated deployment environment.
- Use virtualenv

1.2.3 Install deploy commander

- Install on Ubuntu

Once installed, take a look at our tuturials, key concepts and general usage

1.2.4 Running the webserver

The webserver exposes a rest api which is used by external applications. If you want to integrate with bitbucket, github or other systems you'll need start this server.

1.2.5 Example deployment configuration

We have a github example repo available with all the examples from the tuturials deploy-commander-example

1.3 Key concepts

Although we try to make this application as simple as possible we expect you to understand the basic principles we have for deploy-commander.

1.3.1 Project, Environment and Tasks

Deploy commander is based on 3 main principles.

- The project you want something to do with. (*Can be defined by the config*)
- The environment where this will be executed. (*testing, acceptance, production*)
- The task you want to execute. (*Deploy, Backup DB, Clear cache*)

The projects, environments and tasks must be defined in the settings.

1.3.2 Config

Security

Settings can be encrypted, see the security section.

Configuration is stores in .json files. You can find them in the config folder. The configuration holds all information related to deploy-commander and your project.

More information about the configuration can be found in our config section.

1.3.3 Tasks

Tasks define a set of actions to execute. This is one of the core elements of the system. The key defined will be usable in the command line execute. (deploy-commander go run:<task>) You can define as many tasks as you want.

```
{
  "tasks": {
    "tasks-1": {
      "description": "Test task 1",
      "actions": {}
    },
    "tasks-2": {
      "description": "Test task 2",
      "actions": {}
    }
  }
}
```

Key options:

- input_params (optional, input params from command line)
- description
- actions

Full example

```
{
  "tasks": {
    "tasks-1": {
      "input_params": {
        "unique-key": {
          "param": "git_branch",
          "prompt": "Enter branch"
        }
      },
      "description": "Test task 1",
      "actions": {}
    },
    "tasks-2": {
      "description": "Test task 2",
      "actions": {}
    }
  }
}
```

1.3.4 Actions

Actions define the command to execute and extra params that the command needs.

So in a normal deploy/build you'll make a task *deploy* and define actions something like:

- Backup database (mysql.backup_db)
- Checkout/update git repo (git.deploy)

```
{
  "tasks": {
    "deploy": {
      "description": "Deploy app example",
      "actions": {
        "mysql-backup-db": {
          "command": "mysql.backup_db",
          "sequence": 1
        },
        "git-clone": {
          "command": "git.deploy",
          "sequence": 2
        }
      }
    }
  }
}
```

Possible options:

- sequence : Numeric value in which order the commands will be executed (Required)
- command : Main command to execute <package>.<action> (Required)
- params : Set of key/values for the command. (Default: none)
- confirm : Asks if you want to execute this command.. you can enter an own question as it's value. (Default: none)
- description : Can be used to describe the command (Default: none)
- enabled : Is this action enabled in the sequence? (Default: true)

Every command can have it's own params... see the commands for their options.

So a full example would look like:

```
{
  "tasks": {
    "deploy": {
      "description": "Deploy app example",
      "actions": {
        "mysql-backup-db": {
          "description": "Make a backup from the mysql database",
          "enabled": "True",
          "command": "mysql.backup_db",
          "sequence": 1,
          "confirm": "Lets backup the database?",
          "params": {
            "host": "%(db_host)s",
            "user": "root",
            "password": "root",
            "database": "your-database",
            "backup_file": "/full/path/to/%(user)s/backup/path/file.sql",
            "download_tar_to_local_file": "./local/path/db/backup.tar.gz"
          }
        },
        "git-clone": {
          "description": "We're executing a git clone action!",
          "enabled": "True",
          "command": "git.deploy",
          "sequence": 2,
          "confirm": "Sure you want to clone ?",
          "params": {
            "git_repo_path": "/full/path/to/repo",
            "git_source_path": "/full/path/to/source/%(tag)s",
            "git_branch": "develop"
          }
        }
      }
    }
  }
}
```

You can use generic params in the command params to prevent repetition. So if you define a key/value in the 'main' params you can use this like '%(param)s'

1.3.5 Commands

Commands can best be defined as an isolated predefined functions with certain functionality. We try to put as much functionality for each command to keep the config simple.

See the commands section for a full list

1.3.6 Params

Params can be used for creating dynamic variables in your settings.

Params can be (re)used in post param values and command param values. This is an easy way to manage central 'constants' that can be used by different commands

You can use the params in all the other tasks/actions based on your input.

```
{
  "params": {
    "name": "Minime",
    "yourvar": "what you want"
  }
}
```

Much of the params for actions/commands can be defined in the global params without defining them in the action itself. (See the difference in the "action" examples.) You can find the global params in the commands section

Next to that we have system params. (that can be overwritten)

- timestamp (unix timestamp)
- user (current unix user logged in to remote server)
- project (initial set from the command line)
- environment (initial set from the command line)

1.3.7 Post params

Post params are build/formatted with the param values. This is an easy way to generate generic reusable params for your commands. In the example we used a path, this is a good way to manage base path's for your commands

```
{
  "params": {
    "name": "Minime",
    "yourvar": "what you want"
  },
  "post_params": {
    "some_path": "/home/%(user)s/%(name)s/",
    "base_path": "/home/%(user)s/%(project)s/%(environment)s/"
  }
}
```

Now *some_path* will evaluate as : */home/root/Minime/*

1.4 Usage

Usage is very simple, by executing a command from the terminal you can initiate *tasks*

If this is the first time you're using this, you might want to start with a tuturial and understand the key concepts.

1.4.1 Command line execute

There are 2 options when executing a *deploy-command* from the command line.

```
$ deploy-commander go run:<task>
```

The *go* part will ask you for the project and environment you want to use. The *run* part will execute several actions (*Like git clone, mysql import*) combined. We defined this as *the task*.

You can bypass the prompt for the project and environment by setting them in your command.

```
$ deploy-commander go:<project>,<enviroment> run:<task>
```

1.4.2 Show config

Show full config

```
$ deploy-commander go:<project>,<enviroment> show_config
```

1.4.3 Show available tasks

```
$ deploy-commander go:<project>,<enviroment> show_tasks
```

1.4.4 Show task

Will show the task for a certain project/environment

```
$ deploy-commander go:<project>,<enviroment> show_task:<task>
```

1.4.5 Run webserver

The webserver exposes a rest webserver with some extra functionality for post hooks. (bitbucket)

See the config section for possible configurations.

```
$ deploy-commander runserver
```

Stop any deploy-commander processes

```
$ deploy-commander stopserver
```

1.5 Tuturials

We're working on more tuturials we're *still in development mode*

1.5.1 Install on Ubuntu

Go to home root, or the folder where you want to install.

```
$ cd ~
```

Install python pip and dev

```
$ sudo apt-get -y install python-pip python-dev
```

Install virtualenv, and activate

```
$ virtualenv environment
$ . environment/bin/activate
```

Install deploy commander, and all it's dependencies

```
$ pip install deploy-commander
```

This will install the python libraries.

Run the server

```
$ deploy-commander runserver
```

1.5.2 Example Configurations

We have a ready to go repo in github that contains a basic setup with the following:

- Vagrant file (for your local dev machine)
- Setup server with puppet
- Encrypted configs
- Configs with different examples
- Main config.dist for main configuration

See the github repo : https://github.com/munstermedia/deploy-commander-example.

Git checkout

In this example we'll use virtualbox and vagrant.

We'll assume you have worked with then...

We only tested this system on unix like machines, like Ubuntu and MacOS. Currently we don't support other flavors... sorry... (allthough it must work on centos too...) So for now, this quick demo can't be run if you are using a windows machine.

Clone an example

```
$ git clone https://github.com/munstermedia/deploy-commander-example.git
```

Go into repo

```
$ cd deploy-commander-example
```

Setup main config

```
$ mv config.json.dist config.json
```

Load development server, a ubuntu trusty box with ip 192.168.56.135

```
$ vagrant up
```

Executing tasks

We're gonna start with a small example project and inspect some configuration files.

Install app

We're gonna install a new app *php-info* on a development environment.

```
$ deploy-commander go run:install-app
```

it will prompt you for the project and environment.

A shortcut to do the same:

```
$ deploy-commander go:phpinfo,development run:install-app
```

Different environments

Try to enter different environments, in this example they all point to the vagrant box, but for your production it can point to different servers.

What just happened?

- This will create base folders and clone the repo into a development enviroment
- We've cloned a repo into */home/<user>/<env>/repo*
- We've created a database
- We've installed the default install.sql from repo

Deploy app

Now we're gonna deploy the source code and use the master branch to do so.

```
$ deploy-commander go run:deploy-app
```

This wil prompt you with the same like install but it will ask for a tag. The default tag is the latest from the list.

What just happened?

- We've updated */home/<user>/<env>/repo*
- We've created */home/<user>/<env>/source/<tag>* from the repo
- We've backupped the database in */home/<user>/<env>/db_backup*
- We've created a symlink */home/<user>/<env>/current* to */home/<user>/<env>/source/<tag>*

Rollback app

Now we're gonna rollback the app... .. euhm rollback? impossible in continuous integration right?

```
$ deploy-commander go run:rollback-app
```

What just happened?

- We've removed the old symlink */home/<user>/<env>/current* and linked it to the new tag you've entered.
- If we answered yes to import database, we could rollback the database to another version.

1.6 Config

This page contains some example configuration that will give you some more insight of the possible parameters.

In your config you can setup stuff like ssh credentials, symlinks, mysql backup and much more...

It's important to understand how settings are loaded and which settings are possible. We've implemented a nice feature to inherit settings based on environments and projects.

We'll start by explaining each config file by it's location on the filesystem.

1.6.1 Root config

The main configuration file must be located in the root folder and named "config.json" This config file contains json structured params, like your master password.

Note: Don't commit this file to your repo because the master password should be stored elsewhere.

/config.json

```
{
  "master_password": "abc1234",
  "env": {
    "debug": "False",
    "warning_only": "True",
    "running": "False",
    "stdout": "False"
  }
}
```

1.6.2 Mail config

When using automated deployments and executing tasks it can send out notifications and other mails to keep you and your team up to date.

Note: If you'll leave the mail configuration empty it won't send out any mails.

Don't have a smtp server? Try the free mandrillapp

/config.json

```
{
  "mail":{
    "host":"smtp.host.com",
    "port":"587",
    "from":"your@email.com",
    "user":"smtpuser",
    "password":"smtppassword",
    "to":["devops@yourdomain.com"]
  }
}
```

1.6.3 Slack hook

Send a message to slack!

/config.json

```
{
  "hook":{
        "slack_hook_url":"https://thehookyouvegeneratedwithslack"
  }
}
```

1.6.4 Hook branch mapping

Listen to which branch to trigger an action

You can use regex for the key.

Example : if you'll merge a branch into testing-1234 it will deploy.

/config.json

```
{
  "hook":{
        "environment_mapping":{
    "^(testing)":"testing",
    "^(release)":"staging"
}
```

```
  }
}
```

1.6.5 Webserver config

Default path

The runserver command will autodetect the current path, application path and virtualenv path to generate the right command for executing the gunicorn webserver.
By default it expects the virtualenv folder 'environment' in the root of the home path where your configuration lives.

The webserver runs by default on port 8687, and ip 0.0.0.0.

This config will overwrite the default config.

dc_application_path : the deploy commander path. dc_virtualenv_path : path to virtualenv dc_home_path : path to configuration

/config.json

```
{
  "webserver":{
        "ip":"0.0.0.0",
"port":"8687",
"dc_application_path":"/deploy-commander/code/path",
"dc_virtualenv_path":"/default/home/path/environment",
"dc_home_path":"/default/home/path",
"workers":1
  }
}
```

1.6.6 Initial project config

This config file is located in the /config/<project> folder. Here you can define settings related to deploy commander.

This file is optional, and not required to be present. The application autodetects the presence of this file.

/<project>/config.json

```
{
  "config_load_strategy": [
    "config/default.json",
    "config/%(environment)s.json",
    "config/%(project)s/default.json",
    "Your own config here"
  ]
}
```

Note: config_load_strategy is optional, leave it empty by default

1.6.7 Global project config

Config Examples

For more information about the configuration options please see the examples in ./config in the example github repo.

The basic foundation of this system are tasks and actions. You can initiate a task, and these have actions to execute.

The default config strategy (if not set in /<project>/config.json) will be:

1. **/config/default.json**

This is the base config. Everything will be extended from this config.

2. **/config/%(environment)s.json**

Main config for development environments. This will overwrite the ./config/default.json

3. **/config/%(project)s/default.json**

Main Config for the development project. This will overwrite the ./config/default.json, and development.json

4. **/config/%(project)s/%(environment)s.json**

Config for the development project. This will overwrite the ./config/default.json, default.json and development.json

To view the configuration from the command line you can run:

```
$ deploy-commander go show_config
```

When running a task it will load in sequence (if available) and combine the configuration of:

1. Generic config (default.json)
2. Generic environment config (<environment>.json)
3. Project config (<project>/default.json)
4. Project environment config (<project>/<environment>.json)

Note: If you want to load a different config strategy for your project, you can create a file '<project>/config.json' (See *Initial project config*)

So in this case, the 2'nd config will append and overwrite the 1'st. The 3'rd the 2'nd etc..

This is an example structure of a project config:

1. **/config/default.json**

```
{
  "params": {
    "some_param": "param/value"
  },
  "post_params": {
    "dynamic_post_param": "/some/%(some_param)s/path"
  },
  "tasks": {
    "deploy": {
      "description": "Deploy project",
      "actions": {
        "your-own-description": {
          "sequence": 1,
          "execute": "command.action",
          "params": {
            "dynamic_param": "%(dynamic_param)s/repo",
            "dynamic_post_param": "%(dynamic_post_param)s/source/%(tag)s"
          }
        }
      }
    }
  }
}
```

2. **/config/testing.json**

```
{
  "params": {
    "environment": "testing"
  }
}
```

3. **/config/test/default.json**

```
{
  "params": {
    "project_name": "test"
  },
  "post_params": {
    "post_test": "%(project_name)s-%(environment)-%(password)s"
  },
  "tasks": {
    "deploy": {
      "description": "Lets change the title"
    }
  }
}
```

4. **/config/test/testing.json**

```
{
  "params": {
    "password": "1234"
  }
}
```

When executing : $ deploy-commander go:test,testing show_config it will output as:

```
{
  "params": {
    "some_param": "param/value",
    "password": "1234",
    "project_name": "test",
    "environment": "testing"
  },
  "post_params": {
    "dynamic_post_param": "/some/%(some_param)s/path",
    "post_test": "%(project_name)s-%(environment)-%(password)s"
  },
  "tasks": {
    "deploy": {
      "description": "Lets change the title",
      "actions": {
        "your-own-description": {
          "sequence": 1,
          "execute": "command.action",
          "params": {
            "dynamic_param": "%(dynamic_param)s/repo",
            "dynamic_post_param": "%(dynamic_post_param)s/source/%(tag)s"
          }
        }
      }
    }
  }
}
```

1.7 Commands

Commands can best be defined as an isolated predefined functions with certain functionality.

1.7.1 System commands

system.symlink

```
{
  "your-own-description": {
    "sequence": 1,
    "command": "system.symlink",
    "params": {
      "source": "/path/where/to/create/symlink",
      "target": "/path/where/the/symlink/should/link"
    }
  }
}
```

Functionality:

- Creates symlink, if symlink allready exists it will remove the existing one.

system.command

```
{
  "list-source": {
    "sequence": 1,
    "command": "system.command",
    "params": {
      "command": "ls -las /home",
      ("secure":"False")
    }
  }
}
```

Functionality:

- Run command on server

- If secure param is set to true, then it won't output the command with params. Default value is False

system.multi_command

```
{
  "list-source": {
    "sequence": 1,
    "command": "system.multi_command",
        "params": {
    "command":"print %(domain)s",
    "list_config_file": "config/%(project)s/list/%(environment)s.json",
    ("secure":"False")
}
  }
}
```

Functionality:

- Run command on server and use a source json file to format params.
- It will format the command based on the list_config_file
- The list_config_file wil overwrite any previous set params
- If secure param is set to true, then it won't output the command with params. Default value is False

Example list

```
[
  {
    "domain": "domain1.com"
  },
  {
    "domain": "domain2.com"
  }
]
```

system.multi_local_command

```
{
  "your-own-description": {
    "sequence": 1,
    "command": "system.multi_local_command",
    "params": {
    "command":"print %(domain)s",
    "list_config_file": "config/%(project)s/list/%(environment)s.json",
    ("secure":"False")
    }
  }
}
```

Functionality: (see multi_command)

- Run command on local server and use a source json file to format params.
- It will format the command based on the list_config_file
- The list_config_file wil overwrite any previous set params
- If secure param is set to true, then it won't output the command with params. Default value is False

system.upload_template

```
{
  "upload-environment-config": {
    "sequence": 1,
    "command": "system.upload_template",
    "params": {
      "source": "some/file/in/the/template/path.ini",
      "target": "/path/where/to/copy/on/the/server.ini",
      "yourvar_1": "whatever",
      "yourvar_2": "you-want"
    }
  }
}
```

Functionality:

- Uploads template from .template folder/file to server.
- Renders the template with params.. you can use {{ param_name }} in the template. In this example the path.ini could contain the param {{ yourvar_1 }}.
- Unlimited own params.. source and target are required

system.ensure_path

```
{
  "your-own-description": {
    "sequence": 1,
    "command": "system.ensure_path",
    "params": {
      "path": "/the/full/path/you/need"
    }
  }
}
```

Functionality:

- Checks if folders exists, if not it will try to create the path

system.download_from_remote

```
{
  "your-own-description": {
    "sequence": 1,
    "command": "system.download_from_remote",
    "params": {
      "remote_path": "/some/remote/path/*.jpg",
      "local_path": "./templates/tmp"
    }
  }
}
```

Functionality:

- Will download file(s)
- Can use wildcards for files.

- Can download one or more files/folders

system.upload_to_remote

```
{
  "your-own-description": {
    "sequence": 1,
    "command": "system.upload_to_remote",
    "params": {
      "local_source": "/some/remote/path/*.jpg",
      "target_source": "./templates/tmp"
    }
  }
}
```

Functionality:

- Will upload file(s)
- Can use wildcards for files.
- Can upload one or more files/folders

system.filesystem_remove_old

```
{
  "delete-old-files": {
    "sequence": 1,
    "command": "system.cleanup_old_files",
    "params": {
      "minutes": "86400",
      "path": "/some/path"
    }
  }
}
```

Params: - minutes : Remove files older then x minutes (optional, default 86400) - path : path to folder.(required)

Functionality:

- Deletes old files and (sub)folders

1.7.2 Git commands

git.clone

```
{
  "your-own-description": {
    "sequence": 1,
    "command": "git.clone",
    "params": {
      "git_repo_path": "/full/path/to/repo",
      "git_repo_url": "https://github.com/munstermedia/demo.git"
    }
  }
}
```

Functionality:

- Checks if repo path exists.. if not it will ask to reinstall and it will reset/remove all existing code in the path.
- Clones the repository to the path

git.deploy

```
{
  "your-own-description": {
    "sequence": 1,
    "command": "git.deploy",
    "params": {
      "git_repo_url": "http://www.somegit.repo",
      "git_repo_path": "/full/path/to/repo",
      "git_source_path": "/full/path/to/source/%(tag)s",
      "git_branch": "deploy-0.0.1"
    }
  }
}
```

Functionality:

It will use the code in the repo path to go to a certain branch/tag. This will be copied to a tag path so you'll have versioned codebases living next to each other.

- Branch in params is required, you can use input_params to make this dynamic.
- If repo path is not existing it will exit. You'll need a valid cloned repo path
- If target path is allready existing it will remove it and all it's content. And deploy a completely new version.
- Allow updates submodules by running 'git submodule update'

1.7.3 MySql commands

mysql.backup_db

```
{
  "your-own-description": {
    "sequence": 1,
    "command": "mysql.backup_db",
    "params": {
      "host": "localhost",
      "user": "root",
      "password": "root",
      "database": "your-database",
      "backup_file": "/full/path/to/database/backup/path/file.sql",
      "download_tar_to_local_file": "./local/path/db/backup.tar.gz"
    }
  }
}
```

Functionality:

- Runs mysqldump and creates a mysql sql that will be compressed to tar.gz.
- The generated sql file will be removed.

- Tries to create the path on remote if it doesn't exist
- If download_tar_to_local_file is given it will download the tar.gz for local backup

mysql.cleanup_db_backups

(Gonna be deprecated, this is something we should implement with the backup_db)

```
{
  "your-own-description": {
    "sequence": 1,
    "command": "mysql.cleanup_db_dumps",
    "params": {
      "path": "/full/path/to/database/backup/path",
      "max_backup_history": "5"
    }
  }
}
```

Functionality:

- Reads path for *.tar.gz files... and removes the oldest files. (by filesystem)
- max_backup_history is optional, defaults to 5

mysql.query

```
{
  "your-own-description": {
    "sequence": 1,
    "command": "mysql.query",
    "params": {
      "host": "localhost",
      "user": "root",
      "password": "root",
      "query": "CREATE DATABASE IF NOT EXISTS your-db-name"
    }
  }
}
```

Functionality:

- Execute a raw query thru the command line

mysql.import_file

```
{
  "your-own-description": {
    "sequence": 3,
    "command": "mysql.import_file",
    "params": {
      "host": "localhost",
      "user": "root",
      "password": "root",
      "database": "your-database",
      "import_file": "/full/path/to/repo/.data/install.sql"
    }
  }
}
```

Functionality:

- Executes : 'mysql -h %(host)s -u %(user)s –password='%(password)s' %(database)s < %(import_file)s'
- If import_file does not exist it will show a warning and will continue to the next command

mysql.restore_db

```
{
  "your-own-description": {
    "sequence": 2,
    "command": "mysql.restore_db",
    "params": {
      "host": "localhost",
      "user": "root",
      "password": "password",
      "database": "your-database",
      "backup_path": "/full/path/to/database/backup/path",
      "version": "sql-version"
    }
  }
}
```

Functionality:

- By default the version is left empty, but you can force this.
- It will list the versions and prompt for a version to restore when params is empty.
- Requires valid backup version

1.8 Security

Because your config files contain critical credentials for your database/ssh you should consider some security measures.

1.8.1 Environment

This application will probably run on a dedicated server.

- Be sure to use long passwords, better to generate them... like 15 characters and mixed symbols, digits, uppercase, lowercase.
- Use an encrypted home... so if the hardware is stolen it can't be used.
- Setup a firewall with minimal access.
- Setup a firewall on the server where you'll deploy to!

1.8.2 Encrypted config's

Main password

There is a main password set in the .config file. With this password the configuration files are encrypted/decrypted...
Please set this password with a strong password generator (and make sure it's escaped for json)

The config files can be encrypted before you want to push them to some repo. This way you can maintain your deploy setup in git without exposing login credentials and other critical information.

The config files are encrypted with a AES256 encryption. For more technical info see the simple-crypt library. Remember to setup this password on your production server manually, and do not commit this in any repo!

There are 2 commands available to encrypt and decrypt all the config files. It won't encrypt the config files if it is named as config.json. Only config.crypt.json will be encrypted/decrypted

To encrypt all crypt.json config files:

```
$ deploy-commander encrypt_config
```

or process only path...

:: $ deploy-commander encrypt_config:<./config/path>

To decrypt the .json.encrypt config files:

```
$ deploy-commander decrypt_config
```

or process only path...

```
$ deploy-commander decrypt_config:<./config/path>
```

- It will only encrypt or decrypt files that needs to be encrypted/decrypted. It might take a while if your configuration is large.

1.9 Cheatsheet

Quick overview of the command line possibilities.

```
// Run a command
$ deploy-commander go:<project>,<environment> run:<action>

// List config
$ deploy-commander go:<project>,<environment> show_config

// Encrypt/Decrypt config
$ deploy-commander encrypt_config
$ deploy-commander decrypt_config

$ deploy-commander encrypt_config:<path>
$ deploy-commander decrypt_config:<path>

// Show available tasks
$ deploy-commander go:<project>,<enviroment> show_tasks

// Show task info
$ deploy-commander go:<project>,<enviroment> show_task:<task>
```

```
// Run the api server
$ deploy-commander runserver
```

1.10 Integration

Deploy commander can be integrated with different systems.

1.10.1 Bitbucket integration

It's possible to connect bitbucket to deploy-commander and automate some deployment tasks.

We use the git flow by default for automated builds. See a successful git branching model

How it works

Bitbucket has a post pull request hook that will post information when pull and merge requests are executed. We have exposed a rest endpoint that can catch this post request.

Setup and start the webserver (`See the usage`, and `config`)

```
$ deploy-commander runserver
```

By default it will listen to the merge action from bitbucket. So when somebody merges code in the develop branch it will execute :

```
$ deploy-commander go:<project>,testing run:deploy-app
```

When somebody merges code into the release branch it will execute:

```
$ deploy-commander go:<project>,staging run:deploy-app
```

Note: Still in beta now.. we'll update as much as possible.

Setup Bitbucket hook

Go to your webadmin https://bitbucket.org repo. Under settings -> webhooks you'll find add webhook.

Select a trigger (pull request->merged) and enter your deploy-commander rest endpoint:

- http://<yourip/domain>:8086/api/v1/bitbucket/webhook

1.11 News

1.11.1 1 Sept 2016

Added possibility to configure on which environment to deploy and which branch to listen when a hook is triggered.

1.11.2 8 Dec 2015

I’ve decided that most of the commands are just ‘linux commands’ and can be executed by the system.command. So in the next release expect a deprecated notification for a lot of commands.

1.11.3 24 March 2015

So.. i’ve created an easy way to start a webserver that listens to the bitbucket post hook requests. This nice feature will deploy automaticly to the testing and staging environment.

In the next release i will work on a ‘deployment strategy configuration’ which can be used to configure how deploy-commander will react to bitbucket and github post request.

1.11.4 17 Feb 2015

Started testing a clean setup on ubuntu working with a Vagrant box. See the Vagrantfile and puppet folder for more info.

Goal of this test is to look for possible problems with the installation. It seemed that cython needs some dependencies that can be installed by the apt-get package python-dev.

Added an initial folder for unit tests.

Allso added a new variable *environment* to the runserver command so we can select a virtualenv. I think it might be a better idea to start the webserver based on certain configuration in de root config.json. So i’ve added my first issue: #1

Released a new version 0.1.4 with these changes

1.11.5 15 Feb 2015

So we’ve finally created a version that works and our documentation is getting better. I was thinking *now* is the time to start with a historical news log.

In the current release (0.1.3) we’ve integrated a rest webserver to handle task request and integrate a bitbucket hook to automate the deployment proces.

We think the rest api will be the entry point to communicate with deploy-commander. We’re thinking about developing a web interface that ‘talks’ to your deploy-commander instance. Maybe a iphone/android app to track deployments would be a nice addition!

So in the next releases we might move certain command line actions to the rest api. Expect more functionality in the upcoming months!