abaco Documentation

Joe Stubbs,KeDarius Whitley

Jul 31, 2018

Introduction:

CHAPTER 1

Deployer

1.1 What is Deployer

The Deployer project provides a command-line tool for automating the deployment of distributed systems developed in the Cloud and Interactive Computing group (CIC) at the Texas Advanced Computing Center. Currently, support for the following projects is planned:

- TACC's Integrated JupyterHub
- Abaco API
- Agave API

The CIC group uses Deployer internally to manage deployments of its hosted offerings of the above systems, but external groups can also use Deployer to manage their own on-premise deployments of these systems.

1.2 Organization of Documentation

The Deployer documentation is organized into three sections: *Getting Started*, which covers installation and basic concepts; the User Guide, intended for operators who will use Deployer to manage deployments of one or more of the supported projects and the Developers Guide, which is intended for individuals wishing to develop Deployer itself.

CHAPTER 2

Getting Started

This Getting Started guide will conver installing the Deployer CLI as well as some initial basic concepts needed before moving to the User Guide.

2.1 Installation

To install the Deployer CLI, the only dependency is Docker. If you do not have Docker installed, see the Official Docs for installation instructions for your platform.

Once Docker is installed, change into a temporary directory and run:

```
$ docker run -v $(pwd):/conf tacc/deployer --setup && mv deployer /usr/local/bin/
↪deployer
```

The first step pulls down the latest stable version of the Deployer Docker image and installs a small alias script, `deployer`, in the current working directory. The second stop is optional but recommended so that the `deployer` script is on `$PATH` and not left in the temporary directory.

Note: Different versions of Deployer can be installed by specifying a TAG on the `tacc/deployer` image.

With the alias installed, simply issue commands directly to the script. For example, validate your installation by executing:

```
$ deployer --help
```

which should display the help.

2.2 Basic Usage

The general format for executing commands via the the Deployer CLI is:

```
$ deployer <command> -p <project> -i <instance> -t <tenant> -a <action> -d
↪<deployment_file>
```

For any kind of deployment activity, one will use the `execute` command to the Deployer CLI, which is its default value and can be omitted. However, the Deployer CLI recognizes a few other informational commands. For example, we can get the version of the installed Deployer using the `version` command which requires no other arguments:

```
$ deployer version
TACC Deployer
Version: 0.1-rc1
```

The arguments `project`, `instance`, etc, are defined in the following table and are covered in more detail in the *Basic Concepts* section below.

CLI Arguments:

-h, –help	show help message and exit
-p PROJECT, –project PROJECT	Software project to deploy such as JupyterHub or Abaco. Overrides that specified in the deployment file.
-i INSTANCE, –instance INSTANCE	Instance to deploy, such as 'prod' or 'dev'. Overrides that specified in the deployment file.
-t TENANT, –tenant TENANT	Tenant to deploy, such as 'SD2E' or 'designsafe'. Overrides that specified in the deployment file.
-a ACTION, –action ACTION	Action to take, such as 'deploy' or 'update' Must be a valid action for the project specified.
-s SERVERS, –servers SERVERS	Relative path to servers file, in the YAML format, of servers to target. This file should be in the current working directory or a sub directory therein.
-c CONFIGS, –configs CONFIGS	Relative path to config file, in the YAML format, of config to use. This file should be in the current working directory or a sub directory therein.
-z PASSWORDS, –passwords PASSWORDS	Relative path to passwords file, in the YAML format, of passwords to use. This file should be in the current working directory or a sub directory therein.
-e EXTRAS, –extras EXTRAS	Relative path to a directory of extra files needed for the action. This path should be in the current working directory or a sub directory therein.
-o OVERRIDES, –overrides OVERRIDES	String of config overrides; should have the form key1=value1&key2=value2.. These values will override values supplied in the config file at the project/instance/tenant level specified through those corresponding command line arguments or the deployment file
-v, –verbose	The value of the actor's state at the start of the execution.
-k, –keep_tempfiles	Whether to keep the temporary Ansible files generated to execute playbooks
-vv, –very_verbose	Display very verbose output.

2.3 Basic Concepts

The following concepts are important to understand before using Deployer.

2.3.1 Projects, Instances and Tenants

The notions of `project`, `instance` and `tenant` are fundamental to Deployer's approach to managing deployments. A `project` is one of a set of systems Deployer knows how to manage, and will eventually include the TACC JupyterHub, Abaco and Agave projects. When working with Deployer CLI, projects are referenced by a project id. To see information about what projects are supported in an existing Deployer installation, including their id's, use the `list_projects` command:

```
$ deployer list_projects
  Available Projects:

  TACC Integrated JupyterHub
  **************************
  id: jupyterhub
  Description: Customized JupyterHub enabling deeper integration and
  ease of use of TACC resources from within running notebooks.
  Docs: http://cic-deployer.readthedocs.io/en/latest/users/projects.html#jupyterhub
```

The values for `instance` and `tenant` can be chosen by the operations team to best organize their infrastructure and configuration. One approach is to use `instance` values to distinguish physically isolated systems such as "development" and "production" and to use `tenant` values to distinguish logically separated aspects of systems (such as the DesignSafe tenant for JupyterHub or Abaco).

2.3.2 Actions

Actions define what procedure should be taken on the deployment. Actions are defined on a project by project basis, though some standard actions such as `deploy` are available for all projects.

To see which actions are available for a given project, use the `list_actions` command, specifying a project; e.g.:

```
$ deployer -p jupyterhub list_actions
  Available actions: ['deploy']
```

Note that the value to the project argument must be a project id, as output by the `list_projects` command.

2.3.3 Deployment Files

In order to use Deployer, the operator will need to supply some deployment files describing the infrastructure to deploy onto and the configuration for the projects to deploy. At a minimum, this includes configuration file(s) and server file(s). Details on how to write these files are provided in the User Guide. We encourage teams to keep these files in a version control system and check them out on each machine that will run Deployer. For example, the CIC team stores its own deployment files in a bitbucket repository.

2.3.4 Hierarchical Organization of Properties

The goal of the Deployer design is to minimize the time needed to write deployment files by eliminating the need to ever duplicate a property definition for a server or a project configuration. To achieve this goal, Deployer uses a hierarchical organization of properties for both servers and configuration, organized by `project`, `instance` and `tenant`.

In general, each `instance` belongs to exactly one `project`, and each `tenant` belongs to exactly one `instance`. Properties can be defined at a `project`, `instance` or `tenant` level, and property values defined at a more "local" level override those defined at a more "global" level. For example. if the `jupyter_user_image` property is defined for the "prod" instance but also for the "DesignSafe" tenant within the "prod" instance, then the value defined for DesignSafe would be used for all deployment actions taken against that tenant.

More details are given in the User Guide.

2.3.5 Ansible

The Deployer contains scripts that can be launched from the command line to manage deployments on remote servers. It does so by first reading configuration files, server files, (optionally) extra files and command line arguments provided by the operator to generate temporary Ansible playbooks and then execute these playbooks on the remote servers specified. In general, the operator should not need to know anything about the generated Ansible scripts, and by default, Deployer removes these files after each command. For debugging purposes, Deployer can be instructed to keep these files using the `-k` flag.

CHAPTER 3

Overview

This section of the documentation is intended for individuals planning to use Deployer to mange their own deployments of CIC projects. Before starting this section, make sure to go through the Getting Started section and install the Deployer CLI.

CHAPTER 4

Terminology

Deployer makes use of the following concepts to manage remote deployments:

Project A CIC software project that will be managed. Support is planned for JupyterHub, Abaco, and Agave.

Instance A physical manifestation of a project, often times for a specific environment such as "dev", "staging", "production", etc.

Tenant A logical isolation of a (typically, subset) of resources within an instance.

Action The devops action to take against the project such as "deploy", "update", etc. Represents a single, atomic action to take on a set of servers; typically implemented as one or more playbooks. Actions may invoke other actions as part of execution.

Note that a given instance is only defined within the context of a project; for example, the `prod` instance of the `JupyterHub` project and the `prod` instance of the `Abaco` project are different. In the same way, a tenant is only defined within a given instance.

CHAPTER 5

Configs

Operations teams put desired project configuration in YML files ending in the `.configs` extension. The config file to be used can be specified at the command line (`-c`) or in the deployment.yml file. Otherwise, the first file with a `.configs` extension in the current working directory will be used.

Each config must specify a project and can be optionally assigned to an instance or tenant. These are specified by defining a YML key in the format `<project>.<instance>.<tenant>` and then defining the properties using valid YML within that stanza. Configs can be complex YML objects.

Consider the following example config file:

```
---
jupyterhub:
    jupyterhub_image: taccsciapps/jupyterhub:0.8.0
    jupyter_user_image: taccsciapps/jupyteruser-ds:1.1.0
    use_stock_nfs: false
    jupyterhub_config_py_template: jupyterhub_config_0.8.py.j2
    worker_tasks_file: designsafe.yml

jupyterhub.prod.designsafe:
    use_stock_nfs: true
    # prod didn't define images, etc., so it should inherit from the above.

    # here's an example of a config leveraging a complex YML object:
    volume_mounts:
      - /corral-repl/tacc/NHERI/shared/{username}:/home/jupyter/MyData:rw
      - /corral-repl/tacc/NHERI/published:/home/jupyter/NHERI-Published:ro

jupyterhub.staging.designsafe:
    jupyterhub_image: taccsciapps/jupyterhub:0.8.1
    jupyter_user_image: taccsciapps/jupyteruser-ds:1.2.0rc9
    # these image properties ^^ override the global onces defined above, but just for␣
↪Designsafe staging
    cull_idle_notebook_timeout: 259200
    notebook_memory_limit: 3G
    notebook_offer_lab: True
```

In the first stanza, with key `jupyterhub`, a set of configuration properties and values are specified. These values will provide a default for all actions taken on the "jupyterhub" project if the property isn't otherwise specified. In the next two stanzas, properties values for the "designsafe" tenant within the "prod" and "staging" instances, respectively, are defined. These values override the values defined in the "jupyterhub" stanza when deployment actions are taken against the particular instances.

So, when running Deployer actions against the prod instance of the DesignSafe tenant, the `jupyterhub_image` would have value `taccsciapps/jupyterhub:0.8.0` (inherited from the `jupyterhub` stanza) and `use_stock_nfs` would have value `true`. For the staging instance, `jupyterhub_image` would have value `taccsciapps/jupyterhub:0.8.1` and `use_stock_nfs` would have value `false`.

CHAPTER 6

Servers

Servers are the remote hosts where projects will be deployed. Deployer will ultimately execute commands over SSH on these servers to manage the deployments on behalf of the operator. The servers an operator wishes to have Deployer operate on are described in one or more server files. Here is a small example of a potential servers file:

```yaml
---
ssh_user: root
ssh_key: my_ssh.key

jupyterhub.prod.sd2e:
    cloud: roundup
    hosts:
        - host_id: manager1
          ssh_host: 129.114.97.247
          private_ip: 129.114.97.247
          roles:
            - hub
        - host_id: worker1
          ssh_host: 129.114.97.248
          private_ip: 129.114.97.248
          roles:
            - worker
        - host_id: worker2
          ssh_host: 129.114.97.249
          private_ip: 129.114.97.249
          roles:
            - worker

jupyterhub.staging:
    cloud: jetstream
    hosts:
        - host_id: worker2
          ssh_host: 129.114.17.47
          private_ip: 10.10.10.6
          roles:
```

(continues on next page)

(continued from previous page)

```
            - worker
```

In the above example, we define two global properties, `ssh_user` and `ssh_key`, and two groups of servers defined in the `jupyterhub.prod.sd2e` and `jupyterhub.staging` stanzas. Property values are organized hierarchically just like configuration values, and more "local" values override more "global" values.

In general, server files should conform to the following requirements:

- Written in the YML format ending in the `.hosts` extension.
- Each entry in the YML file should be either a global property to apply to all servers in the file (e.g., `ssh_key`

for the SSH key Deployer should use to access to the servers, as in the above example) or a stanza containing descriptions of one or more servers. * Each host must be assigned to a project and can be optionally assigned to an instance or tenant. The project, instance and tenant are specified through the YML key in dot notation, `<project>.<instance>.<tenant>` * Each host must define `ssh_host`, the IP address or domain of the host and `host_id`, a unique name for the host. * The `ssh_key` and `ssh_user` properties are also required for all hosts, though they can be inherited. The `ssh_key` parameter must be a relative path from the directory where Deployer is run to a key file that can be used to access the server via SSH, and the `ssh_user` parameters must be a string representing the user account to SSH as. * Each host can optionally provide a list of `roles`. Project scripts use `roles` to distinguish which services should run on which servers. The roles attribute must be a list. * Each host can have a additional key:value pairs defined by the operators used for further filtering (e.g. "cloud: jetstream")

The servers file to be used can be specified at the command line (-s) or in the deployment.yml file. Otherwise, the first file with a .hosts extension in the current working directory will be used.

CHAPTER 7

Passwords

In Deployer, password files are used to specify sensitive properties such as database passwords or the contents of an SSL certificate. The passwords file to be used can be specified at the command line (-z) or in the deployment.yml file. Otherwise, the first file with a .passwords extension in the current working directory will be used. Password files are the same as config files in all regards; for example, each password is either a global proerty or (optionally) is assigned to a project, instance or tenant and password properties can be arbitrary YML objects.

CHAPTER 8

extras_dir

Path relative to the current working directory to a directory containing extra files needed for configuring the actions. These files could include SSL certificates, the Jupyter volume_mounts config, the Abaco service.conf, etc. When possible, actions are strongly encouraged to generate these files using configuration strings and templates instead of requiring physical files in the extras_dir.

CHAPTER 9

Projects

This section covers documentation for specific projects supported by Deployer.

9.1 JupyterHub

TACC's integrated JupyterHub project extends the community supported JupyterHub with customizations that enable deeper integration and ease of use of TACC resources from directly within running notebook servers.

The main features provided by the TACC JupyterHub are:

- Individual Jupyterhub instance for each project with customized notebook server image based on the research interests of the community.
- Dedicated notebook server running in a Docker container for each user.
- Persistent storage volumes backed by large-scale storage at TACC and mounted directly to the "permanent" directory within each notebook container.
- Additional volume mounts, customizable through configuration, to provide POSIX interfaces to additional file collections stored on TACC systems.
- Integrated authentication with the TACC OAuth server or another tenant OAuth server.
- TACC CLI, agavepy, and other libraries pre-installed in the user's notebook server for integrating with other TACC Cloud services.

The CIC group at TACC maintains the core TACC JupyterHub code and automated deployment management in Deployer. The notebook server images are developed by a number of individuals across multiple groups at TACC as well as through contributions by external partners.

9.1.1 Primary Server Requirements and Options

The TACC JupyterHub uses Docker Swarm 1.11 to deploy notebook servers as Docker containers on a cluster. It requires a minimum of two servers meeting the following requirements:

- One server with the `hub` role. This server will run the main JupyterHub application and therefore will require ports 80 and 443 open to the public internet.
- One or more servers with the `worker` role. These servers will run the user notebook containers.
- Ports must be opened between manager and workers to enable communication. For each server, the optional `private_ip` attribute may be specified. If is it, communication will happen on this IP. Otherwise, it will happen on `ssh_host`. If firewalld and/or iptables are running on the hosts, make sure that both the local IP and the 172.17.0.0 subnets (the Docker0 bridge) are opened.

In addition to a Docker 1.11 Swarm cluster, an NFS management cluster will be constructed from the `hub` node (nfs server) to all `worker` nodes (nfs clients). This NFS management cluster is required as it is used to persist login data from the hub to the workers.

9.1.2 Primary Configuration Requirements and Options

The following configuration properties are required.

- `tenant` - The tenant id this JupyterHub belongs to. Determines the OAuth server to use.
- `jupyterhub_tenant_base_url` - The base URL for the OAuth server. Note: this property is only required if not using a standard TACC tenant; for standard TACC tenants, Deployer will automatically derive the base URL.
- `jupyterhub_host` - The domain to serve JupyterHub on (without the protocol); e.g. "jupyter.designsafe-ci.org".
- `jupyterhub_oauth_client_id` - The ID for the OAuth client that JupyterHub will use to authenticate users. The JupyterHub requires a valid OAuth client for the tenant being used, and this OAuth client *must* be registered with the required `callbackUrl` for the hub, which has the form `https://<jupyterhub_host>/hub/oauth_callback`. See *Additional Considerations* below for more details.
- `jupyterhub_oauth_client_secret` - The corresponding client secret for the OAuth client. Note: it is recommended that the `jupyterhub_oauth_client_secret` be stored in a `.passwords` file.

The following configuration properties are optional, though some are strongly encouraged; see below.

- `jupyterhub_cert` and `jupyterhub_key` - As strings, the contents of an SSL certificate and key for the domain specified in `jupyterhub_host`. If these attributes are not specified, *self-signed certificates will be supplied*. This will result in insecure warnings in the browser for users.
- `volume_mounts` - A list of directories to mount into every user notebook container. Each mount should be a string in the format `<host_path>:<container_path>:<mode>` where `host_path` and `container_path` are absolute paths and `mode` is one of `ro` (read only), or `rw` (read-write). Also, the paths variables recognize the following template variables:
  - `{username}` - the username of the logged in user.
  - `{tenant_id}` - the tenant id for the JupyterHub.
  - `{tas_homeDirectory}` - the home directory for the logged in user in TAS (TACC Accounting System). This template variable can only be used in tenants using the TACC identity provider (see *Integrating with StockYard* below).
- `jupyterhub_image` - Docker image to use for central JupyterHub program. Will default to deploying the latest stable version. Image must be available on the public Docker Hub.
- `jupyter_user_image` - Docker image to use for user notebook servers. If not specified, the latest stable version of `taccsciapps/jupyteruser-base` will be used.

- `jupyteruser_uid` and `jupyteruser_gid` - The uid and gid to run the user notebook servers as; if not specified, the uid and gid created in the Docker image specified by `jupyterhub_image` will be used unless integration with TAS is enabled (see *Integrating with TAS* below).

9.1.3 Additional Considerations

We highlight some additional considerations regarding JupyterHub configuration.

- While `volume_mounts` is technically optional, at least one mount is needed to provide persistent user data; we generally recommend mounting a user directory on the host (e.g. `/path/to/nfs/share/{username}` to a path such as `/home/juupyter/my_data` inside the container.
- Every `host_path` in the `volume_mounts` parameter must exist on *all* worker nodes (for example, via an NFS share) or container execution will fail. Unless using the NFS management cluster, Deployer assumes these directories have been created already.
- For security purposes, ensure that the `oauth_client_secret` is in the `.passwords` file
- To generate an OAuth client key and secret for your JupyterHub instance, use a command like the following:

```
$ curl -u <service_account>:<service_account_password> \
-d "clientName=<you_pick>&callbackUrl=https://<jupyterhub_host>/hub/oauth_callback" \
https://<tenant_base_url>/clients/v2
```

9.1.4 Integrating with TAS

If the tenant being used for JupyterHub leverages the TACC identity provider (i.e., ldap.tacc.utexas.edu) then JupyterHub can integrate with TAS (the TACC Accounting System) to enable individual notebook servers to run as the uid and gid of the logged in user. This feature provides the advantages of "vertical single sign-on". i.e., files created and/or updated by the user in the notebook server will have the correct ownership properties. Integrating with TAS is required for integrating with Stockyard, TACC's gloabl file system. (see *Integrating with Stockyard* below).

Integrating with TAS also requires the following configurations:

- `use_tas_uid` and `use_tas_gid` - Setting to `true` instructs JupyterHub to launch the user's notebook with the uid and gid for the user in TAS.
- `tas_role_account` and `tas_role_pass` - an account and password in TAS for Jupyterhub to use to make TAS API calls.

9.1.5 Integrating with StockYard

JupyterHub instances can integrate with TACC's global file system, Stockyard, if TAS integration has been enabled. This option is only available to approved JupyterHub instances deployed within the secured TACC network.

In order to provide file system mounts from Stockyard into user notebook servers, a Lustre mount to Stockyard must be made on all worker nodes.

Once the Lustre mounts have been created on the worker nodes, the only configuration required is to add notebook container mounts to Stockyard using the `volume_mounts` parameter. For example, if Stockyard is mounted at `/work` on all worker nodes, creating a mount with the following config

```
/work/{tas_homeDirectory}:/home/jupyter/tacc-work:rw
```

would mount the user's $WORK directory at `/home/jupyter/tacc-work` in read-write mode within the container.

9.1.6 Support for Project Mounts (DesignSafe)

As an example of custom functionality that be can be added for specific JupyterHub instances, we describe the support for project mounts in the DesignSafe tenant.

Coming soon…

9.2 Abaco

Coming soon…

9.3 Agave

Coming soon…

CHAPTER 10

Overview

This section of the documentation is intended for individuals who want to develop the Deployer project itself. Before starting this section, make sure to go through the getting-started/index section and install the Deployer CLI.

CHAPTER 11

High Level Algorithm Design of Core CLI

1. Read Servers, Configs and Passwords files, and denormalize each of these into "flat" python objects; properties specified at a more "local" level override properties specified at a more "global" level.
2. Filter Servers, Configs and Passwords using the specified project, instance and tenant (if specified).
3. Combine Configs and Passwords into a single Configuration object.
4. **Look up action class by project; each project/action can do the following things:**
    - Define implied roles to apply to the servers list; e.g., the "hub" role implies the "swarm_manager" role.
    - Audit the filtered Servers and Configuration object for required fields.
    - Supply default values for variables within the Configuration object for optional fields. This will be a formal mechanism so that the central script can report when optional field values are supplied.
    - Execute one or more playbooks. A central "execute_playbook()" callable will be provided that can accept the Servers and Configuration objects (the action code could thus further filter either object before calling the central execute_playbook).
    - Execute one or more additional actions. A central "execute_action()" callable will be provided that can accept the Servers and Configuration objects (the action code could thus further filter either object before calling the central execute_action).
5. **execute_playbook**
    - Create a temporary directory for the playbook to run in.
    - Copy playbook source directory (roles, playbook files, etc.) to the temp dir.
    - Copy the extras_dir to a directory called _deployer_extras within the temp dir.
    - Write the Servers object to a file called _deployer_servers.yml in the temp dir.
    - **Determine the unique roles among the servers.**
        - For each role, make a group in the servers file (i.e., [<role_name>]) and write out all servers that are in that role with all variables attached.

    - In general, a given server could appear in multiple groups but that is OK.

- Write the Configuration object to a file called _deployer_configuration.yml in the temp dir.
- Execute the playbook through a subprocess (ansible-playbook) and reference the servers file (-i <servers>).
- The playbook/roles should, by convention, issue a "- include_vars: _deployer_configuration.yml" at the top of the tasks list to pick up the configuration.