hubploy Documentation

Release 0.1

Yuvi Panda

Dec 30, 2020

Contents

`hubploy` is a suite of commandline tools and an opinionated repository structure for continuously deploying JupyterHubs on Kubernetes (with Zero to JupyterHub). Find the `hubploy` repository on GitHub.

CHAPTER 1

Hubploy workflow

Every change to your hub configuration must be made via a pull request to your git repository. Guided by principles of continuous delivery, this informs hubploy's design.

1.1 Components

The following components make up a hubploy based deployment workflow:

1. A deployment *git repository*, containing *all* the configuration for your JupyterHubs. This includes image configuration, zero-to-jupyterhub configuration, and any secrets if necessary. hubploy is designed to support many different hubs deploying to different cloud providers from the same repository.
2. A *staging hub* for each JupyterHub in the git repo. End users rarely use this hub, and it is primarily used for testing by devs. The `staging` branch in the git repo contains the config for these hubs.
3. A *prod(uction) hub* for each JupyterHub in the git repo. End users actively use this hub, and we try to have minimal downtime here. The `prod` branch in the git repo contains the config for these hubs. However, since we want prod and staging to be as close as possible, the *prod branch match the staging branch completely* under normal circumstances. The only commits that can be in prod but not in staging are merge commits.

1.2 Deploying a change

CHAPTER 2

How-To Guides

These how-to guides are intended to walk you through the basics of particular tasks that you might do with Hubploy.

2.1 How-To Guides Home

2.1.1 How to Setup a Repository to Deploy a JupyterHub with Hubploy

This is a guide on how to deploy a JupyterHub with Hubploy.

General Procedure:

- *Step 0: Setup Prerequisites*
- *Step 1: Get the hubploy-template Repository*
- *Step 2: Install Hubploy*
- *Step 3: Configure the Hub*
- *Step 4: Build and Push the Image*
- *Step 5: Deploy the Staging Hub*
- *Step 6: Deploy the Production Hub*
- *Step 7: Setup git-crypt for Secrets*
- *Step 8: GitHub Workflows*

Step 0: Setup Prerequisites

Hubploy does not manage your cloud resources - only your *Kubernetes* resources. You should use some other means to create your cloud resources. Example infrastructure deployments can be found at the terraform-deploy repository. At a minimum, Hubloy expects a Kubernetes cluster. Many installations want to use a shared file system for home directories, so in those cases you want to hvae that managed outside Hubploy as well.

You also need the following tools installed:

1. Your cloud vendor's commandline tool.
   - Google Cloud SDK for Google Cloud
   - AWS CLI for AWS
   - Azure CLI for Azure
2. A local install of helm 3. Helm 2 is also supported, but requires the same version of Helm to be present locally and on the cluster. If you are sing Helm 2, you can find both versions with `helm version`.
3. A docker environment that you can use. This is only needed when building images.

Step 1: Get the `hubploy-template` Repository

There are a couple different options for acquiring the content in this repository.

- Use the repository as a template. Click the "Use this template" button on the GitHub repository's page, then input your own repo name. You can then use `git clone` as normal to get your repository onto your local machine.
- Fork the repository.
- Clone it directly with `git clone https://github.com/yuvipanda/hubploy-template.git`. The disadvantage here is that you probably won't have permissions to push changes and will have to only develop locally. Not recommended.

Step 2: Install Hubploy

```
python3 -m venv .
source bin/activate
python3 -m pip install -r requirements.txt
```

This installs hubploy and its dependencies.

Step 3: Configure the Hub

Rename the Hub

Each directory inside `deployments/` represents an installation of JupyterHub. The default is called `myhub`, but *please* rename it to something more descriptive. `git commit` the result as well.

```
git mv deployments/myhub deployments/<your-hub-name>
git commit
```

Fill in the Minimum Config Details

You need to find all things marked TODO and fill them in. In particular,

1. `hubploy.yaml` needs information about where your docker registry & kubernetes cluster is, and paths to access keys as well. These access key files should be in the deployment's `secret/` folder.
2. `secrets/prod.yaml` and `secrets/staging.yaml` require secure random keys you can generate and fill in.

If you are deploying onto AWS infrastructure, your access key file should look like the aws credentials file (usually found at `~/.aws/credentials`). However, the profile you use *must* be named `default`.

If you want to try deploying to staging now, that is fine! Hub Customization can come later as you try things out.

Hub Customizations

You can customize your hub in two major ways:

1. Customize the hub image. repo2docker is used to build the image, so you can put any of the supported configuration files under `deployments/<hub-image>/image`. You *must* make a git commit after modifying this for `hubploy build <hub-name> --push --check-registry` to work, since it uses the commit hash as the image tag.
2. Customize hub configuration with various YAML files.
   - `hub/values.yaml` is common to *all* hubs that exist in this repo (multiple hubs can live under `deployments/`).
   - `deployments/<hub-name>/config/common.yaml` is where most of the config specific to each hub should go. Examples include memory / cpu limits, home directory definitions, etc
   - `deployments/<hub-name>/config/staging.yaml` and `deployments/<hub-name>/config/prod.yaml` are files specific to the staging & prod versions of the hub. These should be *as minimal as possible*. Ideally, only DNS entries, IP addresses, should be here.
   - `deployments/<hub-name>/secrets/staging.yaml` and `deployments/<hub-name>/secrets/prod.yaml` should contain information that mustn't be public. This would be proxy / hub secret tokens, any authentication tokens you have, etc. These files *must* be protected by something like git-crypt or sops.

You can customize the staging hub, deploy it with `hubploy deploy <hub-name> hub staging`, and iterate until you like how it behaves.

Step 4: Build and Push the Image

1. Make sure tha appropriate docker credential helper is installed, so hubploy can push to the registry you need.

   For AWS, you need docker-ecr-credential-helper For Google Cloud, you need the gcloud commandline tool
2. Make sure you are in your repo's root directory, so hubploy can find the directory structure it expects.
3. Build and push the image to the registry

   ```
   hubploy build <hub-name> --push --check-registry
   ```

   This should check if the user image for your hub needs to be rebuilt, and if so, it'll build and push it.

Step 5: Deploy the Staging Hub

Each hub will always have two versions - a *staging* hub that isn't used by actual users, and a * production* hub that is. These two should be kept as similar as possible, so you can fearlessly test stuff on the staging hub without feaer that it is going to crash & burn when deployed to production.

To deploy to the staging hub,

```
hubploy deploy <hub-name> hub staging
```

This should take a while, but eventually return successfully. You can then find the public IP of your hub with:

```
kubectl -n <hub-name>-staging get svc public-proxy
```

If you access that, you should be able to get in with any username & password.

The defaults provision each user their own EBS / Persistent Disk, so this can get expensive quickly :) Watch out!

If you didn't do more *Hub Customizations*, you can do so now!

Step 6: Deploy the Production Hub

You can then do a production deployment with: `hubploy deploy <hub-name> hub prod`, and test it out!

Step 7: Setup git-crypt for Secrets

git crypt is used to keep encrypted secrets in the git repository. We would eventually like to use something like sops but for now...

1. Install git-crypt. You can get it from brew or your package manager.
2. In your repo, initialize it.

   ```
   git crypt init
   ```

3. In `.gitattributes` have the following contents:

   ```
   deployments/*/secrets/** filter=git-crypt diff=git-crypt
   deployments/**/secrets/** filter=git-crypt diff=git-crypt
   support/secrets.yaml filter=git-crypt diff=git-crypt
   ```

4. Make a copy of your encryption key. This will be used to decrypt the secrets. You will need to share it with your CD provider, and anyone else.

   ```
   git crypt export-key key
   ```

   This puts the key in a file called 'key'

Step 8: GitHub Workflows

1. Get a base64 copy of your key

   ```
   cat key | base64
   ```

2. Put it as a secret named GIT_CRYPT_KEY in github secrets.
3. Make sure you change the *myhub* to your deployment name in the workflows under *.github/workflows*.
4. Push to the staging branch, and check out GitHub actions, to see if your action goes to completion.
5. If the staging action succeeds, make a PR from staging to prod, and merge this PR. This should also trigger an action - see if this works out.

Note: *Always* make a PR from staging to prod, never push directly to prod. We want to keep the staging and prod branches as close to each other as possible, and this is the only long term guaranteed way to do that.

2.1.2 How to Setup a Hubploy Development Environment

This is a guide on how to setup a development environment for Hubploy. Use cases would be for making a custom Hubploy image for your own use or contributing to the Hubploy repository.

- *Prerequisites*
- *Modifying Hubploy Files*
- *Using a Custom Hubploy Locally*
- *Building a Custom Hubploy on DockerHub*
- *Contributing to Hubploy*

Prerequisites

To start, fork the main Hubploy repository and then clone your fork. This will enable easier setup for pull requests and independent development. Methodology for testing Hubploy is limited right now but it is recommendation that you have a working JupyterHub configuration so you can try to build and deploy.

If you don't have such a configuration set up, we recommend setting one up using the hubploy template repository and following the How-To guide Deploying a JupyterHub with Hubploy.

Modifying Hubploy Files

The code for Hubploy is contained in the `hubploy/hubploy` folder. All of it is in Python, so there is no compiling necessary to use it locally. As long as the files are saved, their changes should be reflected the next time you run a `hubploy` command.

Using a Custom Hubploy Locally

Hubploy can be installed via `pip install hubploy`, but this version is very out-of-date. Using a custom version of Hubploy will require different installation methods.

If you are just using your custom Hubploy locally, you can link it with `pip`. Go to the top folder of your `hubploy-template` or JupyterHub deployment repo and run:

```
pip install -e ~/<absolute-path-to>/hubploy
```

You can then make changes to your local Hubploy files and rerun Hubploy commands in the other folder for quick development.

hubploy can also be installed at any specific commit with the following line in a *requirements.txt* file:

```
git+https://github.com/yuvipanda/hubploy@<commit-hash>
```

Building a Custom Hubploy on DockerHub

Another way to use Hubploy is by building a Docker image and pushing it to DockerHub. For this, you will need to have forked the Hubploy repository to your personal GitHub account. You will also need a personal DockerHub account.

Modify the file `hubploy/.github/workflows/docker-push.yaml`. Change `name: yuvipanda/hubploy`
to `name: <your-dockerhub-name>/hubploy`. You will need to input your DockerHub credentials as secrets in your personal Hubploy GitHub repository as `DOCKER_USERNAME` and `DOCKER_PASSWORD`. Also in the GitHub repository, go to the Actions tab and allow the repo to run workflows by clicking "I understand my workflows, go ahead and run them."

Once you have made the changes you want for your custom Hubploy, you can `git push` your local changes. The file mentioned above will automatically attempt to push your Hubploy to DockerHub! If it fails, there will be output in the Actions tab that should have some insights.

Now that you have a publicly-hosted image for your custom Hubploy, you can reference it anywhere you want! In `hubploy-template`, these references are in the `hubploy/.github/workflows/` files

```
jobs:
  build:
    name:
    # This job runs on Linux
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v1
      - uses: docker://yuvipanda/hubploy:20191210215236cfab2d
```

You will need to change the docker link everywhere you see it in these files to the link of your image on DockerHub.

Contributing to Hubploy

If you have your own fork of Hubploy, and have a feature that would be generally useful, feel free to join the dicussions in the Issues section or contribute a PR!

For more details, see the full contribution guide.

2.1.3 How to Build a JupyterHub Image

CHAPTER 3

Topic Guides

These topic guides are meant as informative reference documents about various pieces of Hubploy.

3.1 Topics Home

3.1.1 Hubploy's Expected Directory Structure

Hubploy expects the directory structure shown in the hubploy template repository. The folders must be set up in this fashion:

```
hubploy-template/
├── .github
│   └── workflows
│       ├── deploy.yaml
│       └── image-build.yaml
├── deployments
│   └── hub
│       ├── config
│       │   ├── common.yaml
│       │   ├── prod.yaml
│       │   └── staging.yaml
│       ├── hubploy.yaml
│       ├── image
│       │   ├── ipython_config.py
│       │   ├── postBuild
│       │   └── requirements.txt
│       └── secrets
│           ├── aws-ecr-config.cfg
│           ├── aws-eks-config.cfg
│           ├── prod.yaml
│           └── staging.yaml
├── hub
```

(continues on next page)

(continued from previous page)

```
│    ├── Chart.yaml
│    ├── requirements.lock
│    ├── requirements.yaml
│    ├── templates
│    │   ├── jupyter-notebook-config.yaml
│    │   └── nfs-pvc.yaml
│    └── values.yaml
├── LICENSE
├── README.rst
└── requirements.txt
```

.github Folder

This folder houses the GitHub Workflow files that you can use for Continuous Integration with Hubploy. `deploy.yaml` will attempt to build the staging or production JupyterHub upon updates to the respective GitHub branch. `image-build.yaml` will attempt to build the JupyterHub image upon updates to only the production branch.

These files have references to a Docker image that uses Hubploy. You can change this image. Some options are listed in *How to Setup a Hubploy Development Environment*.

Deployments Folder

The deployments folder can hold multiple subfolders, but each must have the same structure as the hub folder. Renaming the hub folder is part of the recommended workflow for deploying a JupyterHub. Each subfolder directly under deployments needs a different name so that Hubploy can distinguish between them in Hubploy commands.

Each JupyterHub is deployed with YAML files. The YAML files listed under deployments must have these names.

Hubploy takes in secrets for credentialing via the `.cfg` files. You can rename these freely, just be sure to put the proper names into `hubploy.yaml`.

The image folder can have additional files depending on how you are building the image. See more in the image building how-to. If you are not specifying `images` in your `hubploy.yaml` file, the `images/` folder can be deleted.

Hub Folder

The hub folder houses a local Helm Chart. This chart and folder can be renamed, but the name needs to be present in Hubploy commands, the files in the `.github` folder, and in `Chart.yaml` . Modification of the files in here should be done as you would change a Helm Chart.

3.1.2 YAML File Value Overriding in Hubploy

There are several `.yaml` files present in the hubploy-template repository. It can be unclear which settings go in which files. This topic hopes to clear that up a bit. As a reminder, here is the directory structure that Hubploy expects (minimized for focus on the yaml files):

```
hubploy-template/
├── .github
│    └── workflows
│        ├── deploy.yaml
│        └── image-build.yaml
```

(continues on next page)

(continued from previous page)

```
├── deployments
│   └── hub
│       ├── config
│       │   ├── common.yaml
│       │   ├── prod.yaml
│       │   └── staging.yaml
│       ├── hubploy.yaml
│       └── secrets
│           ├── aws-ecr-config.cfg
│           ├── aws-eks-config.cfg
│           ├── prod.yaml
│           └── staging.yaml
└── hub
    ├── Chart.yaml
    ├── requirements.lock
    ├── requirements.yaml
    ├── templates
    │   ├── jupyter-notebook-config.yaml
    │   └── nfs-pvc.yaml
    └── values.yaml
```

GitHub Action Files

The two files under `.github/workflows/` manage individual GitHub Actions for Hubploy. They are independent of most of the rest of Hubploy.

JupyterHub Deployment Files

The main value files are related to the JupyterHub Helm release. The lowest level of these are specified in `hub/values.yaml` via:

```
jupyterhub: {}
```

The braces can be removed once there are yaml values in the file, but they are needed if this block is empty. Appropriate values to put in this file are those that will span both versions of all JupyterHubs that you will deploy with Hubploy, as this file will be used for all of them.

The next file in the heirarchy is `deployments/hub/config/common.yaml`. This file covers deployment values that are common to both the staging and production hubs that Hubploy named "hub," or what you had changed that folder name to. If there are multiple JupyterHubs being managed , each one will have a `common.yaml`. Values in this file will overwrite `hub/values.yaml`.

The next two files in the heirarchy are also in the config folder: `staging.yaml` and `prod.yaml`. These contain values for the staging and production hubs, respectively. Values in these files will override the previous two. These two files do not override each other ever since they are for two different hubs.

The last files in the heirarchy are under the `secrets` directory. These are set in a folder that we tell git-crypt to encrypt when pushing code to GitHub. In general, there shouldn't be anything in these files that overwrites the other `staging.yaml` and `prod.yaml`. It is more expected that values in these files will overwrite default credentials or paths present in the first two files.

A quick summary of the heirarchy follows in descending priority (lower overwrites higher) but ascending generality (higher applies to more hubs):

```
hub/values.yaml
        deployments/hub/config/common.yaml
                deployments/hub/config/staging.yaml
                deployments/hub/config/prod.yaml
                deployments/hub/secrets/staging.yaml
                deployments/hub/secrets/prod.yaml
```

Local Hub Helm Chart Files

Everything under the hub folder is related to the Helm Chart. In `Chart.yaml`, the main specification is what the Chart is named and what version you are on. In `requirements.yaml`, the JupyterHub Helm chart is listed as the only dependency and you can pick a specific version. `values.yaml` is used to provide the lowest level of values for JupyterHub configuration and other deployment pieces that are present in the `templates/` folder or other dependencies you choose to add to the Helm chart.

3.1.3 Helm Versions in Hubploy

- *Helm Versions Present by Default*
- *Using a Custom Version of Helm*
- *Local Usage*
- *GitHub Action Usage*

Helm Versions Present by Default

The `hubploy` Docker image has Helm v2.16.9 and v3.2.4 installed by default. This may depend on the specific version of `hubploy` that is installed. Versions can be found in the Dockerfile present in the base folder of the hubploy repository. There isn't a version matrix to help find which versions of `helm` ship with certain versions of `hubploy`. You can look at the `Dockerfile`'s commit history or just use the most recent version of `hubploy`, which has the versions listed above.

Using a Custom Version of Helm

To use your own installed version of `helm`, set the environment variable `HELM_EXECUTABLE`. `hubploy` will pick up the value from this environment variable to use when running `helm` commands. It will default to `helm`, ie. v2.16.9, if nothing else is installed. You can find the line of code that does this here.

Local Usage

You can use several versions of `helm` in local usage of `hubploy` This does require that you have installed `helm` or are using the `hubploy` Docker image on your local machine.

To use this environment variable on a local installation of `hubploy`, use the following command from your terminal:

```
export HELM_EXECUTABLE=~/absolute/path/to/helm/binary
```

For example, if you wanted to use `helm` v3 locally and had installed and moved it to `/usr/local/bin/helm3`, you would run the following from your terminal:

```
export HELM_EXECUTABLE=/usr/local/bin/helm3
```

If you already have `helm` v2 installed, no extra steps are necessary.

GitHub Action Usage

To use this environment variable in a GitHub Action, use the following lines in your workflow file:

```
env:
  HELM_EXECUTABLE: /absolute/path/to/helm/binary
```

More information on this second option can be found on the Environment variables page on GitHub Docs.

CHAPTER 4

Reference Documentation

These reference documents are here to describe the configuration values of various files in Hubploy .

4.1 Reference Docs Home

4.1.1 Hubploy Configuration Values Reference

This reference doc will detail the various configuration values present in `hubploy.yaml`. Here is the `hubploy.yaml` file that comes with cloning hubploy-template:

```
images:
  image_name: # TODO: Full path to your docker image, based on the following pattern
  # On AWS: <account-id>.dkr.ecr.<zone>.amazonaws.com/<your-hub-name>-user-image
  # On Google Cloud: gcr.io/<project-name>/<your-hub-name>-user-image
  registry:
    provider: # TODO: Pick 'gcloud' or 'aws', and fill up other config accordingly
    gcloud:
      # Pushes to Google Container Registry.
      project: # TODO: GCloud Project Name
      # Make a service account with GCR push permissions, put it in secrets/gcr-key.
↪json
      service_key: gcr-key.json
    aws:
      # Pushes to Amazon ECR
      account_id: # TODO: AWS account id
      region: # TODO: Zone in which your container image should live. Match your␣
↪cluster's zone
      # TODO: Get AWS credentials that can push to ECR, in same format as ~/.aws/
↪credentials
      # then put them in secrets/aws-ecr-config.cfg
      service_key: aws-ecr-config.cfg
```

(continues on next page)

(continued from previous page)

```
cluster:
  provider: # TODO: gcloud or aws
  gcloud:
    project: # TODO: Name of your Google Cloud project with the cluster in it
    cluster: # TODO: Name of your Kubernetes cluster
    zone: # TODO: Zone or region your cluster is in
    # Make a service key with permissions to talk to your cluster, put it in secrets/
↪gkee-key.json
    service_key: gke-key.json
  aws:
    account_id: # TODO: AWS account id
    region: # TODO: Zone or region in which your cluster is set up
    cluster: # TODO: The name of your EKS cluster
    # TODO: Get AWS credentials that can access your EKS cluster, in same format as ~/
↪.aws credentials
    # then put them in secrets/aws-eks-config.cfg
    service_key: aws-eks-config.cfg
```

The various values are described below.

images

The entire `images` block is optional. If you don't need it, comment it out or delete it.

image_name

Full path to your docker image, based on the following pattern:

- On AWS: <account-id>.dkr.ecr.<zone>.amazonaws.com/<your-hub-name>-user-image
- On Google Cloud: gcr.io/<project-name>/<your-hub-name>-user-image

registry

provider

Either 'aws' or 'gcloud'. More options will be present in the future. Both the `aws` and `gcloud` blocks are uncommented. The one that you do not pick should be commented out.

gcloud

project

GCloud Project Name

service_key

`gcr-key.json` by default.

Make a service account with GCR push permissions and put it in `secrets/gcr-key.json`. You can rename this file, but you will also need put the new filename here.

aws

account_id

AWS account ID

region

The zone in which your ECR image will live. This should match the zone where your cluster will live.

service_key

`aws-ecr-config.cfg` by default.

Get AWS Credentials that can push images to ECR. These credentials should be in the same format as found in `~/.aws/credentials` and put in to the file `secrets/aws-ecr-config.cfg`. You can rename this file, but you will also need put the new filename here.

cluster

provider

Either ‘aws’ or ‘gcloud’. More options will be present in the future. Both the `aws` and `gcloud` blocks are uncommented. The one that you do not pick should be commented out.

gcloud

project

Name of your Google Cloud project with the cluster you will create.

cluster

Name of the Kubernetes cluster you will create.

zone

Zone or region this cluster will sit in.

service_key

`gke-key.json` by default.

Make a service key with permissions to talk to your cluster and put it in `secrets/gke-key.json`. You can rename this file, but you will also need put the new filename here.

aws

account_id

AWS account ID

cluster

The name of the EKS cluster you will create.

region

Zone or region this cluster will sit in.

service_key

`aws-eks-config.cfg` by default.

Get AWS credentials that can access your EKS cluster. These credentials should be in the same format as found in `~/.aws/credentials` and put in to the file `secrets/aws-eks-config.cfg`. You can rename this file, but you will also need put the new filename here.

4.1.2 Contribution Guide

- *Setting up for Documentation Development*
- *Setting up for Hubploy Development*

`hubploy` is open-source and anyone can contribute to it. We welcome the help! Yuvi Panda is the original author and can give GitHub contributor access to those who are committed to making `hubploy` better. You do not have to be a contributor on GitHub to suggest changes in the Issues section or make pull requests. A contributor will have to accept your changes before they become a part of `hubploy`.

If you don't have git already, install it and clone this repository.

```
git clone https://github.com/yuvipanda/hubploy
```

Using a forking workflow is also useful and will make seting up pull requests easier.

Once you have made changes that you are ready to offer to `hubploy`, make a pull request to the main hubploy repository. Someone will get back to you soon on your changes.

If you want to dicuss changes before they get onto GitHub or contact a contributor, try the JupyterHub Gitter channel.

Setting up for Documentation Development

The `hubploy` documentation is automatically built on each commit as configured on ReadTheDocs. Source files are in the `docs/` folder of the main repository.

To set up your local machine for documentation development, install the required packages with:

```
# From the docs/ folder
pip install -r doc-requirements.txt
```

To test your updated documentation, run:

```
# From the docs/ folder
make html
```

Make sure there are no warnings or errors. From there, you can check the `_build/html/` folder and launch the `.html` files locally to check that formatting is as you expect.

Setting up for Hubploy Development

See the How-To guide on setting up a development environment for `hubploy`.

In short, you can install `hubploy` and its dependencies easily with the above guide but you will need a kubernetes cluster to do local deployment tests. Some good resources for deploying a kubernetes cluster are:

1. Zero to JupyterHub K8s
2. AWS Terraform K8s Examples

You will also need to reference the section Using a Custom Hubploy Locally, rather than doing a default `hubploy` installation.

CHAPTER 5

Known Limitations

1. hubploy requires you already have infrastructure set up - Kubernetes cluster, persistent home directories, image repositories, etc. There are ongoing efforts to fix this, however.
2. More documentation and tests, as always!