Kube Docs

Release 0.0.1

Shaun Cullen

Sep 29, 2020

CONTENTS:

CHAPTER

ONE

DOCKER

Containers are the basic building block for machine independent cloud based computing. Kubernetes is a container orchestrator, allowing your application and its dependencies to be deployed independent of any given machine or architecture. Docker is the most common method for creating containers. We will go over the most useful methods for creating a Docker container for your application, which we will eventually be deploying to a Kubernetes cluster.

Contents

1.1 Dockerfile: Build Your Container Image

Dockerfiles are used to declaratively create images for Docker containers, which can be integrated into source control. This is a common theme among the dev ops topics we will be covering: finding ways to integrate configuration and deployment into the source control of your projects, so that they can be tracked and modified over time.

Example Dockerfile

```
FROM registry.cullen.io/ubuntu-base:v0.0.8

RUN apt update && apt install -y nginx

ARG GITHUB_PAT

```

(continues on next page)

(continued from previous page)

```
COPY ./requirements.txt /code/requirements.txt

WORKDIR /code

RUN python3.7 -m pip install -r /code/requirements.txt

RUN python3.7 -m pip install git+https://sscullen:$GITHUB_PAT@\
github.com/sscullen/landsat_downloader.git@v0.0.4#egg=landsat_downloader
RUN python3.7 -m pip install git+https://sscullen:$GITHUB_PAT@\
github.com/sscullen/sentinel_downloader.git#egg=sentinel_downloader
RUN python3.7 -m pip install git+https://sscullen:$GITHUB_PAT@\
github.com/sscullen/spatial_ops.git@v0.0.2#egg=spatial_ops

COPY . /code

RUN cp -r /code/common/grid_files/ /usr/local/lib/python3.7/dist-packages/spatial_ops␣
↪&& \
    cp -r /code/common/data/ /usr/local/lib/python3.7/dist-packages/spatial_ops

RUN python3.7 /code/manage.py collectstatic --no-input

COPY nginx_site.txt /etc/nginx/sites-available/jobmanager

RUN ln -s /etc/nginx/sites-available/jobmanager /etc/nginx/sites-enabled/jobmanager

EXPOSE 80

EXPOSE 443

EXPOSE 5000

CMD ["gunicorn", "--bind", "0.0.0.0:5000", "--log-level=debug", "jobmanager.wsgi"]
```

Dockerfile Keywords

FROM Source image to use as a base. The foundation for your image. Often is images of popular linux distros but can also be images you have customized with other Dockerfiles, allowing for images with common requirements to be shared among many docker files. In the example above, `ubuntu-base` has Python3 and gdal already installed.

Note: Notice the URL for the docker image: this is how images are referenced and tagged for access from different docker registries, including the main docker hub. This is discussed more below in the *private registry* section.

RUN Run a command inside the container. Each `RUN` command will add another "layer" (discrete change to the image that is tracked and transferred individually), and many layers will make the image large. `RUN` commands will often be chained with `&&` to reduce the image size.

Note: You can use the experimental `--squash` feature which reduces the amount of layers in an image regardless of how many `RUN` commands there are.

ARG Get environment variables from your machine into the image you are building. In the example this is used for Github authentication so no login or password prompts are needed. In the *build your image* section we go over the different syntaxes for Linux and Windows for using your environment variables in the image build command.

COPY Move your source files and data from your local machine into the docker container image.

WORKDIR Set the `root` directory for the Dockerfile, where all the `RUN` commands will be executed and where the `COPY` commands will put the files they copy from your machine.

EXPOSE Expose the ports your application will be accessible on. For example, for an nginx docker image that is hosting web services, you will expose 443 and 80. For a Django application in development, you could use `django manage.py runserver 0.0.0.0:4000` to start your django instance, and in this case you would expose port 4000 to access your Django application.

CMD The command your docker container will run when it starts. For a simple Python webservice, your command would be starting the webserver application such as `nginx` or `gunicorn`. If you don't want your container to run a command here, you can use `['sleep', 'infinity']` so the container will wait for you to interact with it before shutting down. Once the `CMD` command finishes, the container shuts down.

1.1.1 Installing `pip` Packages Directly from Github

In the example Dockerfile above, we have lines that look like this:

```
RUN python3.7 -m pip install git+https://sscullen:$GITHUB_PAT@\
github.com/sscullen/landsat_downloader.git@v0.0.4#egg=landsat_downloader
```

This allows us to install pip packages directly from Github. We can specify a specific branch and tag, allowing us to make sure we are using the right package. To avoid credential issues, we use a Github Personal Access token, which can be generated from the Github general settings page for your account. This is not required for a public repo.

To be a valid pip package, the repo should be structured like so:

```
repo_root/
    landsat_downloader # <-- actual python module source code
    .gitignore
    Pipfile # <-- requirements.txt equivalent for Pipenv virtual env manager
    Readme.md
    requirements.txt
    setup.py # <-- critical file for the pip package
```

The `setup.py` file is required, and contains metadata and package information. The version number in `setup.py` should match the latest release tag for your repo.

Example setup.py

```
from setuptools import setup

setup(name='landsat_downloader',
    version='0.0.4',
    description='Utilities for downloading Landsat and Sentinel products from USGS',
    url='https://github.com/sscullen/landsat_downloader.git',
    author='Shaun Cullen',
    author_email='ss.cullen@uleth.ca',
    license='MIT',
    packages=['landsat_downloader'],
    zip_safe=False)
```

Pip will use `setup.py` to build the `.whl` package from your repo directly from your source code in the repo. This is important because it makes sure the version of the code we are using is up to date and correct. If we aren't doing active development on a project and only installing it as a dependency, then it is best to install in this way, rather than copying source code using a `COPY` command. If we are actively changing the code on our local machine, the `COPY` command makes more sense.

1.1.2 Exposing Your Service's Ports

Using the `EXPOSE` directive is how you make your docker container service accessible to the outside world. This is also a critical concept in how Kubernetes exposes services from containers to the outside world. While it is possible to mount local volumes and interact directly inside your container, thus bypassing the "service" model employed by exposing ports for APIs and such, it is not a good idea to use Docker containers in this way. Even modest containers should have some sort of network service exposed for interacting with your application.

Live interaction and intervention by a user in your container results in ephemeral changes to your container that are not maintained after the container restarts. In this vein, we should strive to make containers as "stateless" as possible, and move all data "persistence" requirements into dedicated data focused containers and services, which are designed with persistence in mind. More on this will be in the *Kubernetes section* on persistence.

1.2 Build Your Image

Once you have your Dockerfile ready, you need to build the image. You do this by using the Docker CLI `build` command.

```
docker build -f Dockerfile -t ubuntu_base:ver8 .
```

The above command specifies the Dockerfile with `-f` and the tag for the image with `-t`. Lastly, the `.` represents the directory to run the command in, so in that case it is your present working directory. A more complex build command where you passing command line args to be used by the `ARG` directive in your Dockerfile would look like this:

```
docker build --build-arg GITHUB_PAT=${GITHUB_PAT} -f Dockerfile.django -t jobmanager-
↪api:ver1 . --squash --no-cache
```

This command specifies the GITHUB_PAT `ARG` using BASH var substitution, on Windows with Powershell you would do `GITHUB_PAT=$env:GITHUB_PAT`. In either case, you are transferring env vars from your local machine into the container to be used during the build process. `--squash` collapses the layers of the docker image to keep the image small, and `--no-cache` will prevent the build process from using previous built layers. Caching will speed up the build process but it is sometimes nice to build the entire Dockerfile to make sure no issues have occurred since your last build.

> **Warning:** `--squash` is an experimental feature of Docker. To enable experimental features, edit the `/etc/docker/daemon.json` file and add `"experimental": true`. On Windows, edit the Docker Desktop settings, go to Docker Engine, and edit the json there.

If the build process completes successfully, you can list the current images on your machine with `docker images`.

```
$ docker images
REPOSITORY                          TAG                 IMAGE ID
↪CREATED              SIZE
example                             v0.0.1              e22f3bdb392b        9␣
↪hours ago          1.09GB
```

You can see the image id, tag, and size of the images. Image IDs are useful for having concrete references to your images.

1.3 Local Development Inside the Container

Once we have built the image, we want to work with it, test it locally, and make sure it is working properly. We run images with the `docker run` command:

```
docker run -td -p 5001:5000 -e DEVELOPMENT=True -v /home/common/Development/job_
↪manager/:/code -v /mnt/drobos/zeus/:/mnt/zeusdrobo zeus684440.agr.gc.ca/jobmanager-
↪api:v0.0.9 bash
```

If the run command is successful, you will see a random string of characters.

`-td` is telling Docker to run the image in a detached terminal, so that you can access it and disconnect from the container and it will stay running.

`-p` defines port mapping from the container to your local machine, so for `-p 5001:5000`, 5000 is the container port, and 5001 is the local machine port. You can then access the service running on your docker container at `localhost:5001` on your machine. Remember that we `EXPOSE` port 5000 in the Dockerfile previously.

`-e` is for defining environment variables, here we are setting `DEVELOPMENT` to be `True`

`-v` is for mapping local directories on your machine to directories inside the container. Here we map the source code directory so we can make changes to the code on our machine and those changes will appear inside the container. We also mount another volume for data. Similar to the port mapping, the first entry before the `:` is the local machine, and the second entry is the container's directory.

The URL with the image tag is the full URL for the image. We will go over this in the *private registry section* but just remember we don't need the registry URL in the image tag unless we are pushing and pulling from the registry.

Finally we have the command we want to run when we connect to the container, in this case we are running a `bash` shell to interact with the container, but in other more stripped down images you might only have access to the `sh` shell.

1.3.1 Accessing the Container

We can verify the container is running with the `docker ps` command:

```
docker ps
CONTAINER ID        IMAGE                     COMMAND             CREATED
↪STATUS              PORTS                                      NAMES
c8075244d8a9        jobmanager-api:v0.0.24    "bash"              5 days ago
↪Up 5 days           80/tcp, 443/tcp, 0.0.0.0:5001->5000/tcp    sad_thompson
```

The container short name is `sad_thompson` in this example, it is randomly generated each time you run the container. Once you know the name of the container, you can access it with an `docker exec` command:

```
docker exec -it sad_thompson bash
```

Here we are saying give me an interactive terminal for the container and run the command `bash`, this will leave us at a terminal inside the container. From there we can run commands for our application. `exit` will return us to our host machine.

When we are finished with the container, we can stop it from running with `docker kill sad_thompson`, and we can check if any containers are still running again by using the `docker ps` command.

1.3.2 Mounting Volumes Inside the Container

Mounting volumes inside the container is an important concept, useful for local development and getting data to and from the container, but also for Kubernetes, as that is the primary mechanism for getting configuration files into the the containers, in addition to adding persistence to the containers through volume mounts.

Once you are happy with the image created by the Dockerfile, and you have updated your code while developing inside the Docker container, and you are happy with those code changes, you will need to build your image one more time. After you do this build, you must tag the image with the full URL of the docker registry you want to use.

```
docker images
REPOSITORY                                  TAG                 IMAGE ID
↪CREATED              SIZE
example                                     v0.0.1              e22f3bdb392b        9␣
↪hours ago          1.09GB
docker tag e22f3bdb392b registry.cullen.io/example_image:v0.0.1
```

The format for the image tag is <docker registry url>/<image name>:<semantic version tag>, the docker registry URL is used to differentiate which registry the image should be pushed to.

Once you have tagged the image, you push it with:

```
docker push registry.cullen.io/example_image:v0.0.1
```

You'll also need to make sure that you login to the registry first using the command `docker login registry.cullen.io`.

Note: If you are setting up your own docker registry and it is not secured with HTTPS, you will need to add it as an insecure registry in the `/etc/docker/daemon.json` file by adding `"insecure-registries" : ["myregistrydomain.com:5000"]`, in a similar way that you enabled support for experimental features.

1.4 Docker Private Registry

If you have a lot of your own images and don't want to pay for an account on the Docker hub, you can set up your own registry. Digital Ocean has a great tutorial, and you can also set up a docker registry on your Kubernetes cluster using Helm and an Ingress secured with a Let's Encrypt cert.

You pull images with the `docker pull <full image url>` command and push images with `docker push <full image url>` command.

The current private registry on the bare metal cluster is located at `registry.kub-eo.agr.gc.ca`.

1.4.1 Useful Docker Private Registry URL Endpoints

`/v2/catalog` View available images.

`/v2/<image_name>/tags/list` View available tags for a given image.

1.5 Other Useful Docker CLI Commands

`docker rmi <image_id>` Remove an image.

CHAPTER

TWO

KUBERNETES

Kubernetes is a container orchestration tool for deploying containers across many different physical machines organized into a cluster. We will cover only the very basics of getting a container you created up and running on Kubernetes. It is assumed that a cluster is already set up and can be accessed with the `kubectl` utility from your local machine. The official Kubernetes documentation has good tutorials for setting up a cluster using `kubeadm`, and for local development and learning Minikube is easy to set up and get going with. It is assumed that the cluster is already set up or a managed Kubernetes service is available to you such as EKS or AKS.

Contents

2.1 YAML Manifests

Resources on Kubernetes are defined by `.yaml` format manifest files. Manifest files can be applied to the cluster using `kubectl`, the Kubernetes command line interface, using `kubectl apply -f manifest.yaml`. For each resource type below an example `.yaml` file will be provided.

2.2 Pods and Containers

Pods are the smallest building block of Kubernetes that can be queried directly. The can contain one or more containers, and the containers can expose ports in order to be accessed individually from the overall pod. Volumes can be mounted and assigned to individual containers. Pods have labels which allow services to direct traffic to specific containers within a pod. Think of a pod as a grouping of containers. Different pods can communicate with each other in different ways, including a shared message queue such as Rabbit MQ. In this way, related containers do not have to be in the same pod. In fact, it is common for pods to have only a single container.

2.2.1 Example Pod YAML

```
apiVersion: v1
kind: Pod
metadata:
name: ubuntu-worker
labels:
    name: ubuntu-worker
spec:
containers:
    - name: ubuntu-worker-container
    image: registry.kub-eo.agr.gc.ca/ubuntu-base:v0.0.8
    ports:
        - name: web
        containerPort: 80
    volumeMounts:
        - name: nfsvol
        mountPath: /mnt/zeusdrobo
    command:
        - "sleep"
    args:
        - "infinity"
imagePullSecrets:
    - name: regcred
volumes:
```

(continues on next page)

(continued from previous page)

```
  - name: nfsvol
  persistentVolumeClaim:
      claimName: nfs-pvc-miniostorage
```

2.2.2 Pod Definition Key Aspects

name Used for querying the status of the pod.

labels Used to identify the pod so that services can correctly send traffic to the right pod. In the service definition `selectors` are used to match the `labels` defined for the pod.

containers:image Specifies the image for the container.

containers:ports Specifies the ports exposed by the container, and therefore the pod.

containers:volumeMounts Specifies which of the volumes defined for the pod will be mounted in this container and to which path.

containers:command and args The command the container will run on start up with which variables.

imagePullSecrets The credentials for your private registry so that Kubernetes is able to pull the image. This requires some extra steps to set up which will be covered in the *secret section*.

volumes The persistent volume claim that allows the pod to take a share of a persistent volume and provide that persistence to its containers as a mount point.

2.2.3 Common Commands for Pods

View all pods for the default namespace:

```
kubectl get pods
```

View pods for a specific namespace:

```
kubectl get pods --namespace namespace_name
```

View pods for all namespaces:

```
kubectl get pods -A
```

View details for a pod:

```
kubectl describe pod pod_name
```

```
kubectl get pod pod_name -o yaml
```

View logs for a pod with one container:

```
kubectl logs pod_name
```

View logs for a pod with many containers:

```
kubectl logs pod_name -c container_name
```

Interact inside a container:

```
kubectl exec -it pod_name -- bash
```

Similarly to logs, if you have a pod with more then one container, use `-c` to specify the container.

Delete a pod:

```
kubectl delete pod pod_name
```

When you delete a pod, it doesn't get recreated automatically; that is where a *deployment* comes in handy. Often to re-apply a new configuration or if there are problems with a pod, you will need to delete it and re-apply the manifest to recreate the pod.

Pods are where your code runs, and services are how your code communicates inside the cluster and out.

2.3 Services

Services provide an interface to communicate with the applications running in your pods. The two major types of services are `ClusterIP` and `NodePort`. ClusterIPs are used for inter-cluster communication, while NodePorts enable communication from outside the cluster. For pod to pod communication it is common to use ClusterIP services to do this. NodePorts require access to the IP or hostname of the nodes (the physical hardware providing the resources required by your pods) running your cluster, and the service is differentiated by the port number. A nicer way of allowing access from outside your cluster is with subdomains through an Ingress and ingress manager like a load balancer. That will be covered in the *ingress section*.

So if you want to access a pod running a service on port 6379, and one of your node's IP address is 10.117.206.94, you would create a `NodePort` service that has a nodePort of 30637, and selectors that are equivalent to the pod's labels. The result is that your pod's service is accessible outside the cluster at `10.117.206.94:30637`. If your node had a hostname assigned to it, you could also use that, so it would be `node_hostname.local:30637`.

If you don't specify your nodePort, one will be automatically assigned in the range of port numbers between 30000-32000. Keep this range in mind when manually specifying node ports. Ports in this range will not conflict with the default ports in use on your node.

ClusterIP services don't have a nodePort and are not accessible by machines outside the cluster.

2.3.1 Example Service YAML

```
apiVersion: v1
kind: Service
metadata:
name: redis-service
labels:
    name: redis-service
spec:
type: NodePort
ports:
    - port: 6379
    nodePort: 30637
    targetPort: 6379
selector:
    app: redis
    release: redis-dev
    role: master
```

2.3.2 Service Definition Key Aspects

type Can be `NodePort` or `ClusterIP`.

ports Defines the port that the service will send traffic to on the pod (port and targetPort), while for `NodePort` the nodePort defines what port on your node that requests will be coming from.

selector Corresponds to the `labels` defined on the pod you are wanting to send traffic to. For example, if you pod has a label `name: ubuntu-worker`, then your selector will have `name: ubuntu-worker` as well.

2.3.3 Commands for Services

See the *common commands for Pods section* for examples on how to fetch services using `kubectl get` and `kubectl describe` with the resource type `svc`.

For example:

```
kubectl get svc
```

The above will show all the services for the default namespace.

Deleting services works the same as pods as well.

2.4 Ingress

Ingress is a special kind of service that differentiates the destination of the traffic based on the domain the request came from. This requires a load balancer which will be a assigned an IP address that is accessible outside the cluster. This load balancer IP can be assigned an A record (a primary DNS entry that points to machine IPs), such as `ingress.kub-eo.agr.gc.ca`. Once that `A record` is in place, `CNAME records` (alias DNS records that point domain names to `A record`s) can be created that point to `ingress.kub-eo.agr.gc.ca`. So `subdomain.testing.kub-eo.agr.gc.ca` will point to `ingress.kub-eo.agr.gc.ca`. When a request to `subdomain.testing.kub-eo.agr.gc.ca` is received, the load balancer can see which domain the request was for, and will forward it to the correct pod based on that instead of the port like NodePort services. This is the primary way that web services are built because using domains and subdomains instead of port numbers is much more user friendly.

The caveat to using ingress services is that a load balancer must be available. Cloud based providers have their own load balancers for you to use. If you are running your own bare metal cluster, you must use something like metal lb which provides a load balancer implementation for you to use.

2.4.1 Example Ingress YAML

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
name: jobmanager-ingress
annotations:
    nginx.ingress.kubernetes.io/proxy-body-size: "4000m"
    nginx.ingress.kubernetes.io/proxy-send-timeout: "600"
    nginx.ingress.kubernetes.io/proxy-read-timeout: "600"
    kubernetes.io/ingress.class: nginx
spec:
```

(continues on next page)

(continued from previous page)

```
rules:
- host: s2d2.kub-eo.agr.gc.ca
    http:
    paths:
    - backend:
        serviceName: jobmanager-api
        servicePort: 8080
```

2.4.2 Ingress Definition Key Aspects

annotations Most load balancers use nginx as a reverse proxy, therefore there are some nginx settings that we can change for our ingress. If our ingress is sending big files back and forth, we can up the `proxy-body-size` to 4 GB, and increase the timeouts to 5 minutes if transfers take a long time.

rules Here we set the domain that the ingress relates to, where the request should be sent to. In this case we are using a ClusterIP service that is listening on port 8080.

2.4.3 Ingress Commands

You use similar commands to pods and services with ingresses, only with the `ingress` resource type:

```
kubectl get ingress
```

2.5 Deployments

Deployments are a way of bundling together manifests for pods, services, persistent volume claims, and more, into a single `yaml` manifest. In addition, the deployment will automatically restart deleted pods, which makes it ideal for applications where the configuration changes a lot. Deployments get auto-generated names, so you will see deployments with names like `deployment-name-adsfas-asdfas` with the last 2 suffixes being auto generated IDs for the deployment resource. In most cases, deployments with pods and services should be used instead of pod and service yamls individually.

In the *Helm and Helm Chart section* we go over the use of Helm to manage deployments, but it brings even more power such as variable templating, subcharts, and release versioning and rollbacks. Think of Helm and helm charts as deployments on steroids. You don't always need Helm, but you almost always should be using a deployment.

2.5.1 Example Deployment YAML

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: jobmanager-api
spec:
selector:
    matchLabels:
    app: jobmanager-api
replicas: 1
template:
    metadata:
```

(continues on next page)

(continued from previous page)

```
    labels:
        app: jobmanager-api
    spec:
    containers:
        - name: jobmanager-api
        image: registry.cullen.io/jobmanager-api:v0.0.24
        command: ["gunicorn"]
        args:
            [
            "--bind",
            "0.0.0.0:5000",
            "--log-level=debug",
            "jobmanager.wsgi",
            "--timeout",
            "300",
            ]
        ports:
            - containerPort: 5000
            protocol: TCP
        imagePullPolicy: Always
        volumeMounts:
            - name: nfs-media-vol
            mountPath: /code/media
            - mountPath: /code/config.yaml
            subPath: config.yaml
            name: jobmanager-config-volume
        - name: jobmanager-api-nginx
        image: registry.cullen.io/jobmanager-api:v0.0.24
        command: ["nginx"]
        args: ["-g", "daemon off;"]
        ports:
            - containerPort: 80
            protocol: TCP
            - containerPort: 443
            protocol: TCP
        imagePullPolicy: Always
        volumeMounts:
            - name: nfs-media-vol
            mountPath: /code/media
            - mountPath: /code/config.yaml
            subPath: config.yaml
            name: jobmanager-config-volume
            - mountPath: /etc/nginx/sites-available/jobmanager
            subPath: jobmanager
            name: jobmanager-api-nginx-siteconf
            - mountPath: /etc/nginx/nginx.conf
            subPath: nginx.conf
            name: jobmanager-api-nginx-conf
    imagePullSecrets:
        - name: regcred
    volumes:
        - name: nfs-media-vol
        persistentVolumeClaim:
            claimName: nfs-media-data
        - configMap:
            name: jobmanager-config
            items:
```

(continues on next page)

(continued from previous page)

```
            - key: config.yaml
              path: config.yaml
        name: jobmanager-config-volume
        - configMap:
            name: jobmanager-api-nginx-siteconf
            items:
            - key: jobmanager
              path: jobmanager
        name: jobmanager-api-nginx-siteconf
        - configMap:
            name: jobmanager-api-nginx-conf
            items:
            - key: nginx.conf
              path: nginx.conf
        name: jobmanager-api-nginx-conf
```

It may seem like there is a lot going on, but keep in mind that there is a lot of overlap with the previously covered resources we have seen before, pods and services.

Manifests can be bundled together in one file, seperated by `---`. In this way we can include a service that the deployment will use at the top of the deployment manifest.

```
apiVersion: v1
kind: Service
metadata:
name: jobmanager-api
spec:
ports:
- port: 5000
    protocol: TCP
    name: gunicorn
- port: 8080
    protocol: TCP
    name: static
selector:
    app: jobmanager-api
---
apiVersion: apps/v1
kind: Deployment
metadata:
name: jobmanager-api
spec:
selector:
    matchLabels:
    app: jobmanager-api
replicas: 1
template:
    metadata:
    labels:
        app: jobmanager-api
...
```

2.5.2 Deployment Commands

You use similar commands to pods and services with deployments, only with the `deployment` resource type:

```
kubectl get deployment
```

2.6 Configmaps

Configmaps are the resource to get configuration into your pods and deployments. Any configuration files or environment variables can be set with configmaps. You can set environment variables in a pod definition, but the recommended way to get credentials and config files into your pods is to use configmaps, as it will let you save whole files of config as yaml or json and have that yaml or json be mounted inside your container in a similar way to how data volume mounts work. That is why configmaps are accessed and added to to a pod in the same way that volume mounts from persistent volume claims are.

In the above *example deployment YAML* there are configmaps being loaded and mounted as files inside the pods under the `volumes` section. In this section we will go over how to create configmaps from config files, and more importantly how to update the configmap from a changed config file as your configurations change.

Regular volume mounts using a persistent volume claim name and a config map created using a `.yaml` config file:

```
template:
    ...
    container:
    ...
    volumeMounts:
        - name: nfs-media-vol
        mountPath: /code/media
        - mountPath: /code/config.yaml # Path to mount the configmap
        subPath: config.yaml # Specified so the configmap file mount doesn't replace␣
↪the whole volume
        name: jobmanager-config-volume
volumes:
    - name: nfs-media-vol
    persistentVolumeClaim:
        claimName: nfs-media-data
    - configMap:
        name: jobmanager-config
        items:
        - key: config.yaml # <-- key that points to the YAML file
            path: config.yaml # <-- name of the YAML file when it is created in the␣
↪pod
```

2.6.1 Example Creating A Configmap

Assume you have a configuration file named `config.yaml` that looks like this:

```
S3_URL: http://s3.kub-eo.agr.gc.ca
S3_ACCESS_KEY: <s3 access key here>
S3_SECRET_KEY: <s3 secret key here>
S3_REGION: us-east-1
PSQL_DB: jobmanagerapp
PSQL_DB_USER: jobmanagerapp
PSQL_DB_PASS: <db password here>
```

(continues on next page)

(continued from previous page)

```
PSQL_DB_URL: "10.96.107.240"
PSQL_DB_PORT: "5432"
USGS_EE_USER: <username here>
USGS_EE_PASS: <password here>
ESA_SCIHUB_USER: <username here>
ESA_SCIHUB_PASS: <password here>
DJANGO_SETTINGS_MODULE: jobmanager.settings
RABBIT_MQ_USER: user
RABBIT_MQ_PASS: <rabbit password here>
RABBIT_MQ_URL: "10.96.181.64"
RABBIT_MQ_PORT: "5672"
DJANGO_SECRET: <django secret here>
REDIS_PASS: <redis password here>
REDIS_PORT: "6379"
REDIS_HOST: "10.96.12.117"
MAILGUN_API_URL: https://api.mailgun.net/v3/mg.satdat.space/messages
MAILGUN_API_KEY: <api key here>
```

To create a config map from this file, you would use this command:

```
kubectl create configmap jobmanager-config --from-file config.yaml
```

If you have several versions of your config, but inside the pod the config needs a standardized name, you can rename the config for access inside the pod.

```
kubectl create configmap jobmanager-config --from-file=config.yaml= config_with_
↪nonstandardname.yaml
```

So the config_with_nonstandardname.yaml will be renamed to config.yaml inside the pod. In this way, all of your config files do not have to be named the same.

To view your configmaps:

```
kubectl get configmaps

kubectl get configmap jobmanager-config -o yaml
```

The above command will let you see the structure of the files and values that make up your configmap. In the above case, the key `config.yaml` will be assigned to a valid YAML serialized string, so that when the configmap is mounted, that key will be used to recreate the file and make it accessible inside the pod. Other keys and values can be stored inside a configmap.

To update your configmap:

```
kubectl create configmap jobmanager-config --from-file config.yaml -o yaml --dry-run␣
↪| kubectl replace -f -
```

After you update your configmap, its a good idea to restart your pods with `kubectl delete pod <pod_name>`.

Warning: If you are not using a deployment, your pod will not be recreated automatically, so don't forget to recreate the pod with `kubectl apply -f pod_manifest.yaml`

If you have user credentials that are common to many pods, it might be better to use a *secret*, similar to how we use a secret for the docker private registry credentials in order to pull docker images. But keep in mind, secrets provide

no security beyond configmaps by default. It is just another resource type for credentials that are common across your cluster. Don't assume secrets are secret.

2.7 Secrets

Secrets are base64 encoded configuration and credential storage. Most useful for common config across many pods, or to standardize how you store credentials in your cluster.

An example of creating a secret is the docker private registry secret, where the docker login credentials stored in a .json file are saved in a secret and used by pods to pull the docker image they need.

```
  kubectl create secret generic do-registry \
--from-file=.dockerconfigjson=docker-config.json \
--type=kubernetes.io/dockerconfigjson
```

Secrets are very similar to configmaps, except they are base64 encoded. So to get a secret you would retrieve the secret as a yaml:

```
kubectl get secret secret-name -o yaml

apiVersion: v1
data:
redis-password: Q2p3QlR6cmg5Wg== # base64 encoded credential we are looking for
kind: Secret
metadata:
...
```

To view the credential, we need to decode it like so:

```
echo Q2p3QlR6cmg5Wg== | base64 --decode

CjwBTzrh9Z
```

2.8 Persistent Volumes and Claims

Persistent volumes allow you to you store data across pod restarts, and share data among pods in the case of NFS shares. Persistent volumes are the general storage resource, and persistent volume claims are specific claims to chunks of those persistent volumes which are dedicated to specific pods.

In addition to manually specified persistent volumes and persistent volume claims, there are storage classes which enable auto provisioning. This means that when a pod requests a certain amount of storage from a storage class that supports provisioning, the provisioner automatically creates the claim and allows the pod to access it. You don't have to create the PV and PVC, they are created for you by the provisioner when you create the pod.

2.8.1 Example Persistent Volume and Claim YAML

Persistent volume:

```
apiVersion: v1
kind: PersistentVolume
metadata:
name: nfs-pv-miniostorage
spec:
capacity:
    storage: 10Ti
accessModes:
    - ReadWriteMany
persistentVolumeReclaimPolicy: Retain
nfs:
    path: /mnt/md0/minio_storage
    server: 10.117.206.94
    readOnly: false
```

Persistent volume claim:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
name: nfs-pvc-miniostorage
spec:
accessModes:
- ReadWriteMany
resources:
    requests:
    storage: 10Ti
```

The persistent volume references a specific type of storage, whether that is block storage provided by a cloud provider, an nfs share specified by export and IP address of the server, or others. It is the actual storage that persistence volume claims can try to access and provide to the pods that have references to those claims.

The persistent volume claim sets the requirements, and then Kubernetes tries to find a matching persistent volume that can meet those requirements. Usually if you only have one persistent volume and claim being created at a time, the correct persistent volume will be matched to your persistent volume claim, if it is able to meet the requirements of the persistent volume claim. If you want to explicitly match PV and PVC, you can use a `claimRef` in the PV and `volumeName` in the PVC.

2.8.2 NFS

NFS shares as persistent volumes allows ReadWriteMany data access, which allows multiple pods to read and write from the same volume at the same time. If we create an NFS server outside of the cluster, we can access the data in the persistent volume as a normal network share. The trade off is that auto provisioning is not possible.

NFS shares can be mounted ephemerally directly in the pod manifest, or through persistent volumes and claims, which are then referenced in the pod manifest. The primary difference is you should use persistent volumes if you want the NFS share to managed by Kubernetes and documented in your source control. This should be used if the shares are used across many different pods, as you would only have to change persistent volume or claim once, instead of all of the pods where the NFS settings are defined.

See this article for more info on NFS and the different ways to access the network shares in your pods.

2.8.3 Auto Provisioning

Auto provisioning is when a persistent volume claim is specified with a storage class that with a supporting provisioner. When the claim is specified, the corresponding volume is created automatically. There is an NFS provisioner helm chart, which will create NFS persistence volumes automatically. The trade off is that these NFS volumes are not accessible outside the cluster, and mounting them as regular network shares is not possible (or at least not easy).

On bare metal clusters, where block device volume storage from cloud providers is not available, dynamic provisioning is possible through the use of OpenEBS. A cstor storage pool of block devices can be created using block devices on each node, which creates a pool of storage that you can use to create persistent volumes from. After you create the pool and name the storage class, you can use that storage class in helm charts and deployments to create the persistent volume from the storage pool automatically.

2.9 Helm: Standardized Deployment Format

Helm is a tool that allows you to deploy a set of resources to Kubernetes in a standardized and customizable way. You can bundle complex configurations, dependencies in a format called a helm chart that can be transferred and customized through templating that allows users to install your application with minimal effort. In addition, helm will track the version of the application, allowing for easy rollbacks if things go wrong. Helm charts can be simple or very complex, but in any case it is a good idea to become comforatable with Helm and Helm charts, as you will often be using them to install 3rd party dependencies for your application, such as Postgres, RabbitMQ, Redis, Minio, and others.

Follow the directions in the official Helm docs to install Helm and begin using it.

2.9.1 Example of Using Helm

Here we are going to use Helm to install Postgres on our cluster.

First we find the postgres helm chart we want to use. In this case we will be using the bitnami helm chart.

Then we make sure we have the bitnami repo added to helm:

```
helm repo add bitnami https://charts.bitnami.com/bitnami

helm repo update
```

Then we install postgres, setting the release name to `psql`, using all the default settings:

```
helm install psql bitnami/postgresql
```

More likely, we would want to change a few settings, so we would use the `--set` arg to change some configuration settings:

```
helm install psql bitnami/postgresql --set persistence.storageClass=openebs-hdd-
↪storageclass,persistence.size=100Gi,postgresqlPassword=sBwqls0FpQ
```

You can view all the available configuration options on the chart repo ReadMe.

To see the current helm installs:

```
helm list
```

To remove a helm install:

```
helm uninstall psql
```

2.9.2 Helm Charts

If you have a complex application, you can make it easier to install for you and others by creating a helm chart to bundle it together.

The general format of a helm chart has this structure:

```
root/
    .helmignore
    Chart.yaml
    values.yaml
    templates/
    charts/
    configfiles/
```

The `templates` directory contains your yaml manifests for the different resources for your application. Inside the yaml files you can add templating directives that pulls values from the `values.yaml` file.

```
apiVersion: v1
kind: PersistentVolume
metadata:
name: nfs-pv-{{ include "jobmanager-api.fullname" . }} # templating directive
spec:
capacity:
    storage: 14Ti
accessModes:
    - ReadWriteMany
persistentVolumeReclaimPolicy: Retain
nfs:
    path: /mnt/drobo
    server: zeus684440.agr.gc.ca
    readOnly: false
```

The `values.yaml` file is where you setup your default values for your helm chart, and these are also the values that can be overridden by the user with the `--set` args when installing the helm chart.

`Chart.yaml` contains general info about your chart, such as version, dependencies, and other packaging info.

The `charts/` directory is where you would put subcharts that your chart depends on. Subcharts are full helm charts that your main chart requires to function. So here you could add subcharts for postgres and rabbitmq if your application required those applications to function. Subcharts, and full charts, can be stored as compressed tar archives.

A helm "repo" is a simple web accessible directory with helm charts stored as compressed tar archives with the correct naming convention (`postgresql-8.2.1.tgz`) and an `index.yaml` file that summarizes the available helm charts in the repo.

CHAPTER

THREE

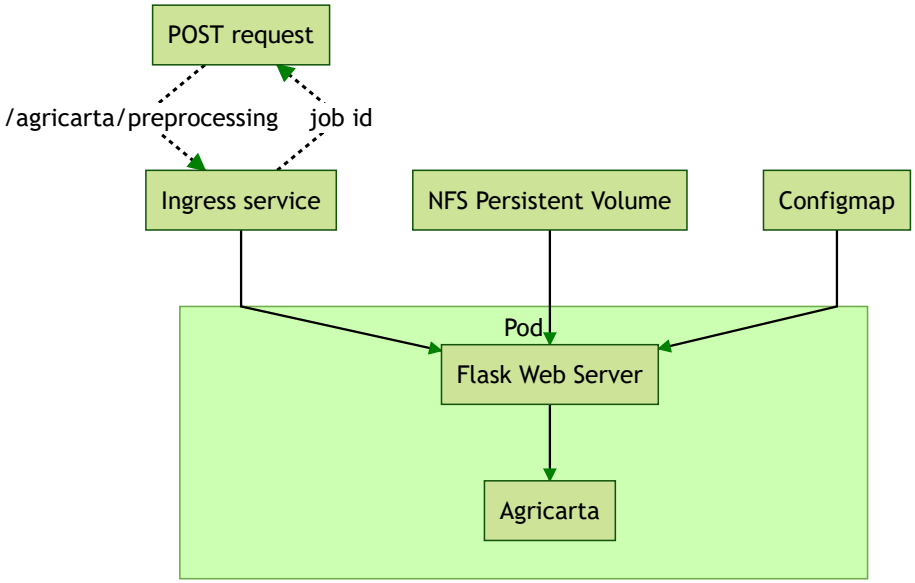
EXAMPLE 1 - SIMPLE - AGRICARTA PREPROCESSING

Contents

3.1 Summary

The first example is as simple as you can get for running an application on Kubernetes. We will have *one pod with one container*, *one service*, *one configmap*, with a simple Flask web server wrapper around the application. Data will be stored on NFS *persistent volumes* with a simple file to indicate processing status. Our input data is already stored on an NFS share, so we can mount that inside our pod. For future improvements, you could look to storing the data as S3 objects to make deployment to the cloud easier, in addition to leveraging GDAL's ability to load from S3 data sources to make incremental networked file loads possible.

3.2 Architecture



3.3 Prerequisites

Imagery data stored on an NFS network share that is accessible to our cluster. For this example we are going to create 2 NFS exports that will be used to store the data inputs and outputs. On the NFS server, you will need to create the directory, edit the /etc/exports file, and restart the nfs-kernel-server. There will also be datasets required for processing that we will mount from a NFS share. All of these shares will be mounted using persistent volume claims inside the container.

You can verify the new mounts are accessible by running `showmount -e <NFS server IP>` from the cluster node machines.

3.4 Build the Container

Here's the Dockerfile we will use:

```
FROM ubuntu:18.04

COPY ./requirements.txt /code/requirements.txt

WORKDIR /code

RUN apt update

RUN apt install -y software-properties-common build-essential

RUN apt-add-repository -y ppa:deadsnakes/ppa

RUN apt update

RUN apt install -y python3.7 python3.7-dev python3.7-venv python3-pip zlib1g zlib1g-
↪dev libpng-dev libjpeg-dev

RUN python3.7 -m pip install --upgrade pip
RUN python3.7 -m pip install numpy
RUN apt install -y gdal-bin gdal-data libgdal-dev

RUN python3.7 -m pip install gdal==`gdal-config --version` --global-option=build_ext --
↪global-option="-I/usr/include/gdal/"

RUN python3.7 -m pip install -r /code/requirements.txt

COPY . /code

EXPOSE 5000

#CMD ["gunicorn", "--bind", "0.0.0.0:5000", "--log-level=debug", "jobmanager.wsgi"]
```

3.5 Service

For the service we will use a Nodeport, but if we had easy access to creating DNS entries, an Ingress would be preferable.

```
apiVersion: v1
kind: Service
metadata:
name: agricarta-nodeport
spec:
selector:
    app: agricarta
ports:
    - protocol: "TCP"
    port: 5000
    targetPort: 5000
    nodePort: 30199
type: NodePort
```

3.6 Deployment

The important parts of the deployment are the single container, which starts our Flask server, and the volume mounts, one for the NFS that our imagery is mounted on, and one for the configmap that stores our generic config options.

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: agricarta
spec:
selector:
    matchLabels:
    app: agricarta
replicas: 1
template:
    metadata:
    labels:
        app: agricarta
    spec:
    containers:
        - name: agricarta
        image: registry.kub-eo.agr.gc.ca/agricarta:v0.0.2
        env:
        - name: FLASK_APP
            value: server.py
        - name: FLASK_DEBUG
            value: "1"
        command: ["flask"] # env FLASK_APP=server.py FLASK_DEBUG=1 flask run --host 0.
↪0.0.0
        args:
            [
            "run",
            "--host",
            "0.0.0.0"
            ]
```

(continues on next page)

(continued from previous page)

```
        ports:
            - containerPort: 5000
            protocol: TCP
        imagePullPolicy: Always
        volumeMounts:
            - name: nfs-miniostorage-volume
            mountPath: /code/miniostorage
            - mountPath: /code/config.yaml
            subPath: config.yaml
            name: agricarta-preprocessing-config-volume
    imagePullSecrets:
        - name: regcred
    volumes:
        - name: nfs-miniostorage-volume
        persistentVolumeClaim:
            claimName: nfs-pvc-miniostorage
        - configMap:
            name: agricarta-preprocessing-config
            items:
            - key: config.yaml
                path: config.yaml
        name: agricarta-preprocessing-config-volume
```

3.7 Configmaps

For the configmap, we are using a config.yaml file for the generic options, and the specific job options will be set by a JSON object in the POST request for our API.

```
ROOT_DIR: /code
# WORKING_DIR: /code/workingdir
DEPENDENCIES_DIR: /code/miniostorage/required_datasets
SRTM_DIR: /code/miniostorage/required_datasets/dem/SRTM41
IMAGERY_STORAGE: /code/miniostorage
# RESOLUTION: 10
# CPU_CORES: 6
# LOGGING_DIR: /code/logs
LOGGING_CONFIG:
console_level: DEBUG
file_level: DEBUG
# DELETE_INTERMEDIATE: True

# Projection as PROJ4 string as string
PROJECTION: "+proj=aea +lat_1=44.75 +lat_2=55.75 +lat_0=40 +lon_0=-96 +x_0=0 +y_0=0␣
↪+ellps=WGS84 +datum=WGS84 +units=m +no_defs"

# Imagery System Parameters
# NOTE: Custom parameters can be defined. Bellow is an example of the defaults
# System will use defaults if 'params' NOT defined within this file
#       LC8 'products' --> dn = Digital Number
#                          radiance = Spectral Radiance
#                          sr = Surface Reflectance
#                          toa = Top of Atmosphere Reflectance
#       RCM / RS2 'filter' --> gamma = Gamma Filter
#       RCM / RS2 'modes'  --> W2 = Wide Beam #2
```

(continues on next page)

(continued from previous page)

```
#       RCM / RS2 'units'  --> amp = Amplitude
#                              dB = Decibel
#                              pow = Power
PARAMS:
LC8:
    bands:
    - B2
    - B3
    - B4
    - B5
    - B6
    - B7
    product: sr
    resamp_clip: True
RCM:
    bands:
    - CH
    - CV
    - HH
    - HV
    - VH
    - VV
RS2:
    filter: gamma
    modes:
    - W2
S2:
    bands:
    - B02
    - B03
    - B04
    - B05
    - B8A
    - B11
    - B12
    resamp_clip: True
```

The command we use to create the configmap from the config.yaml file is:

```
kubectl create configmap agricarta-preprocessing-config --from-file config.yaml
```

3.8 Persistent Volume and Claim

The persistent volume is a NFS share that we have access to inside the cluster and out, so we can mount it as a regular network share in addition to using it as a persistent volume.

```
apiVersion: v1
kind: PersistentVolume
metadata:
name: nfs-pv-miniostorage
spec:
capacity:
    storage: 10Ti
accessModes:
```

(continues on next page)

(continued from previous page)

```
    - ReadWriteMany
persistentVolumeReclaimPolicy: Retain
nfs:
    path: /mnt/md0/minio_storage
    server: 10.117.206.94
    readOnly: false
```

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
name: nfs-pvc-miniostorage
spec:
accessModes:
- ReadWriteMany
resources:
    requests:
    storage: 10Ti
```

3.9 API

The Nodeport service means our API is accessible on port `30199` on the node through the nodes' hostnames or IP addresses. The end point is `/agricarta/preprocessing`, it supports the POST HTTP method, and the processing parameters are JSON data that you include with the POST request.

```
{

    "imagery_list": ["LC08_L1TP_015028_20200814_20200822_01_T1", "LC08_L1TP_015029_
↪20200814_20200822_01_T1"],

    "resolution": 10,

    "cores": 2,

    "delete_intermediate": false

}
```

We can use Postman to test our API.

3.10 Data Management and Access

Because we have access to the NFS share with the imagery we are using, we can use the NFS share to host the preprocessing results. Thus the results are acessible on the NFS share and the S3 Minio instance that NFS share is also backing.

3.11 Flask wrapper

`server.py` is a Flask web server instance that we use to control our application. We run the server in our container with the command `flask run --host 0.0.0.0` as seen above in our deployment definition.

```
from flask import Flask, request

import uuid

import subprocess

app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello, World!"

@app.route('/agricarta/preprocessing', methods=['POST'])
def login():
    if request.method == 'POST':
        json = request.get_json()
        print(json)

        job_id = uuid.uuid4()

        # example sub process command
        # python3.7 server_preprocessing.py
        # LC08_L1TP_042025_20200624_20200707_01_T1 LC08_L1TP_042026_20200624_20200707_
↪01_T1 /code/workingdir config.yaml UNIQUEJOBID 10 4 True
        result = subprocess.Popen(["python3.7",
                                   "server_preprocessing.py",
                                   *json["imagery_list"],
                                   "/code/workingdir",
                                   "config.yaml",
                                   str(job_id),
                                   str(json["resolution"]),
                                   str(json["cores"]),
                                   str(json["delete_intermediate"])])

    # the code below is executed if the request method
    # was GET or the credentials were invalid
    return str(job_id)
```

3.12 Preprocessing Wrapper

We need to create a preprocessing wrapper that calls the Agricarta preprocessing module which accepts command line args and can read our general `config.yaml` configuration file.

`preprocessing_server.py` is similar to the 1_preprocessing.py executor.

```
import click
import logging
import yaml
from pathlib import Path
import os
```

(continues on next page)

(continued from previous page)

```
import shutil
import tarfile

import agricarta as ag
from utilities.common import ConfigFileProblem, ConfigValueMissing

REQUIRED_CONFIG_KEYS = [
        "ROOT_DIR",
        "IMAGERY_STORAGE",
        "DEPENDENCIES_DIR",
        "SRTM_DIR",
        "LOGGING_CONFIG",
        "PROJECTION",
        "PARAMS",
    ]

@click.command()
@click.argument('imagery_names', nargs=-1)
@click.argument('working_dir', type=click.Path(exists=True))
@click.argument('config', type=click.Path(exists=True))
@click.argument('job_id')
@click.argument('resolution')
@click.argument('cores')
@click.argument('delete_intermediate')
def start(imagery_names, working_dir, config, resolution, job_id, delete_
↪intermediate=True, cores=4):
    job_result_log = []

    # Load config from config.yaml
    try:
        with open(config, "r") as stream:
            config = yaml.safe_load(stream)
    except yaml.YAMLError as e:
        logging.error("Problem loading config... exiting...")
        job_result_log.append("Problem loading config")
    except FileNotFoundError as e:
        logging.error(f"Missing config file with path {args.config}")
        job_result_log.append("Missing config file")
    except BaseException as e:
        logging.error("Unknown problem occurred while loading config")
        job_result_log.append("Problem occurred while loading config")

    job_result_log.append(f"Imagery names: {','.join(imagery_names)}")
    job_result_log.append(f"Job id: {job_id}")
    job_result_log.append(f"Working dir: {working_dir}")
    job_result_log.append(f"Resolution: {resolution}")
    job_result_log.append(f"Delete intermediate: {delete_intermediate}")
    job_result_log.append(f"Cores: {cores}")

    JOB_DIR = Path(working_dir, job_id)
    IMAGERY_DIR = Path(JOB_DIR, 'imagery')
    OUTPUT_DIR = Path(JOB_DIR, 'output')

    os.mkdir(JOB_DIR)
    os.mkdir(IMAGERY_DIR)
    os.mkdir(OUTPUT_DIR)
```

(continues on next page)

(continued from previous page)

```
    # Before this is called, make sure all the appropriate folders are created based␣
↪on the job id
    # 1. Create 'imagery' dir, 'output_dir', inside 'job_id' dir
    # 2. Copy image from imagery storage based on the image type
    # 3. Once all imagery is found, run the preprocessing module

    for image in imagery_names:
        # Determine image type
        if image.startswith('LC08'):
            print('Landsat image')
            pathrow = image.split('_')[2]
            path = pathrow[0:3]
            row = pathrow[3:6]
            image_path = Path(config['IMAGERY_STORAGE'], "l8-l2a-products", "tiles",␣
↪path, row, image)

            if image_path.exists():
                print('Image exists')
                job_result_log.append(f"Image: {image} FOUND")
                shutil.copytree(image_path, Path(IMAGERY_DIR, image))
                print("Image copied")
                job_result_log.append(f"Image: {image} COPIED")
                with tarfile.open(Path(IMAGERY_DIR, image + '.tar.gz'), "w:gz") as␣
↪tar:
                    tar.add(Path(IMAGERY_DIR, image), arcname='.')
            else:
                print("imagery not found")
                job_result_log.append(f"Image: {image} NOT FOUND")

    try:
        job_result_log.append('Preprocessing started')
        ag.preprocessing.process_multiple_images([str(IMAGERY_DIR)],
                                                 str(Path(OUTPUT_DIR, "preprocess")),
                                                 str(Path(OUTPUT_DIR, "metadata")),
                                                 params=config['PARAMS'],
                                                 output_projection=config['PROJECTION
↪'],
                                                 output_resolution=int(resolution),
                                                 align_origin=True,
                                                 srtm_path=config['SRTM_DIR'],
                                                 required_datasets=config[
↪'DEPENDENCIES_DIR'],
                                                 delete_intermediate=delete_
↪intermediate,
                                                 log_config=None,
                                                 ncores=int(cores)
        )
    except BaseException as e:

        logging.error('|       EXCEPTION      |: Encountered a generic exception at␣
↪preprocessing task '
                      'scheduler. Details: {}'.format(e))
        job_result_log.append("Exception occured, preprocessing failed")
        job_result_log.append(str(e))

    else:
        job_result_log.append("Preprocessing finished")
```

(continues on next page)

(continued from previous page)

```
    with open(Path(JOB_DIR, "job_log.txt"), "w") as outfile:
        outfile.write("\n".join(job_result_log))

    shutil.rmtree(IMAGERY_DIR)

    shutil.copytree(JOB_DIR, Path(config["IMAGERY_STORAGE"], "agricarta",
↪"preprocessing", job_id))

if __name__ == "__main__":
    start()
```

CHAPTER

FOUR

EXAMPLE 2 - SIMPLE - LANDSAT DOWNLOADER SERVICE

The first example is as simple as you can get for running an application on Kubernetes. We will have one container, one service, connected to a simple wrapper Flask HTTP wrapper around our application.

Contents

4.1 Summary

4.2 Architecture

4.3 Prerequisites

4.4 Build the Container

4.5 Service

4.6 Configmaps

4.7 Ingress

4.8 API

4.9 Logs

4.10 Data Management and Access

CHAPTER

FIVE

EXAMPLE 3 - COMPLEX - DATACUBE

This example is more complex as we have multiple containers with a database dependency, so for this example we will also integrate a helm chart.

Contents

5.1 Summary

5.2 Architecture

5.3 Prerequisites

5.4 Container

5.5 Service

5.6 Configmaps

5.7 Ingress

5.8 API

5.9 Logs

5.10 Data Management and Access

5.11 Helm Chart

CHAPTER

SIX

EXAMPLE 4 - COMPLEX - TILE VIEWER API

This example brings together all of the aspects from the previous example and then adds more complexities. There are multiple pods, pods with multiple containers, and multiple service dependencies that our application needs to function.

Contents

6.1 Summary

6.2 Architecture

6.3 Prerequisites

6.4 Container

6.5 Service

6.6 Ingress

6.7 Configmaps

6.8 API

6.9 Logs

6.10 Data Management and Access

6.11 Helm Chart

6.12 Future Improvements

6.12.1 Horizontal Pod Autoscaling

6.12.2 Celery Downloader Pods as Jobs Instead of Deployments

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search