CERN Web Services – Discourse Forum Automation

August 2019

AUTHOR(S):
Rajula Vineet Reddy

SUPERVISOR(S):
Ismael Posada Trobo
Andreas Wagner

CERN openlab

PROJECT SPECIFICATION

Discourse at CERN is a service developed by the IT department in the last year. The main aim of the service is to provision discourse instances for the people at CERN, upon request, using a given deployment template. Currently, at the time of writing this report, 24 instances are being managed and this number is expected to increase rapidly. Some of the examples of the discourse instances are CERN Marketplace (https://marketplace.web.cern.ch/), Root forum (https://root-forum.cern.ch/), EOS community (https://eos-community.web.cern.ch/), etc. The Discourse communities are created with the aim of linking people on a specific area so that discussions on the area can happen on the latest software such as Discourse. The current implementation of this service is done using managed Openshift cluster, creating an ecosystem based on containers (micro-services). Given that the current steps to provision a Discourse service are manual because of the use of Openshift templates and are not flexible enough. This project seeks to build a Discourse Operator for the Openshift cluster that can run & manage all the CERN's discourse instances by the Openshift cluster itself.

ABSTRACT

The Discourse service which started last year at CERN currently manages 24 instances and the number of instances is expected to increase soon. Currently, the instances are created manually using Openshift templates. This method is not flexible and requires a lot of manual uploading, backup, etc. When a template is used to create an instance, the administrator has no control over the instance as a whole. Any modification needs to be done manually on each of the required components. In order to overcome these, the provisioning of instances needs to be automated, thereby also allowing easier update & recovery. A new concept of Kubernetes, called Operators which are based on the **Custom Resource Definition** feature of Kubernetes comes to the rescue. Operators are custom controllers in Kuberentes, using which application can be deployed as well as managed by the Kuberentes itself. This report discusses how to develop an Openshift operator for the Discourse platform and how it can be deployed. The report also covers using a Postgres operator, which in turns provisions instances for each of the Discourse instance provisioned by the Discourse operator.

TABLE OF CONTENTS

1. INTRODUCTION

The Discourse service is one of the different IT services, started last year at CERN. The aim of the service is to provision Discourse forum instances upon request. The service currently manages 24 instances, among them, are the CERN Marketplace (https://marketplace.web.cern.ch/), Root forum (https://root-forum.cern.ch/) and so on. To give an idea about the instances, the CERN Marketplace instance has about 10,000 users and the ROOT forum instance has about 9,567 users. On top of this, the amount of traffic served by the ROOT instance is almost similar to that of the https://home.cern, the CERN main home page. The number of instances currently managed is expected to grow soon. Given all this, the instances are currently created manually using an Openshift template. Provisioning is not scalable and requires a lot of manual uploading, backup, etc.

With the advent of the Custom Resource Definitions feature in Kubernetes, it is now possible to write a custom controller, or sometimes called as an operator which runs as a Kubernetes pod and can, in turn, create or manage applications running on the cluster. Also, as beneficial are templates in OKD 3.x, RedHat is moving away from templates in OKD 4.x towards operators. Although there were plans to consider Ansible Playbooks (APB) as a replacement of templates, it wasn't pursued. Therefore in order to automate and make it easier to deploy Discourse, we plan to build a Discourse operator based on the Custom Resource Definition feature. The procedure to build such an operator will be discussed in the following chapters.

2. OPERATORS

Kubernetes has made it relatively easy to manage web apps, API services, etc. Kubernetes also enables scaling, rolling updates, services, etc. But most of these apps are usually stateless which makes scaling, deploying, recovery from failures simpler without any additional knowledge. The problem comes into picture when handling stateful application that involves databases, caches, etc. Stateful applications require knowledge of the domain in order to scale, upgrade or recover, all while protecting against loss of data. An operator is a piece of software in Kubernetes, that encodes the domain knowledge of the application, and also extends the Kubernetes API through the Third Party Resources mechanism. Using the extended API users can create, configure and manage the applications. Operators, in general, manage multiple instances across the cluster.

Operators are built on two core concepts of Kubernetes called Resources and Controllers. There are simpler to other controllers such as deployments or daemon sets, but with the additional set of knowledge and configuration. For example, in order to scale an etcd cluster manually, the following steps have to be followed

- Create a new DNS entry for the new member
- Deploy the new member
- Update the number of members using the etcd administrative tools

Whereas with an operator, the user can simply scale by increasing the size field by 1. Operators tasks can also be extended to backup configurations, application upgrades, TLS certificate configuration, service discovery, disaster recovery, etc.

Operators make it easy to manage complex stateful applications on top of Kubernetes.

<Add more about how operators work>

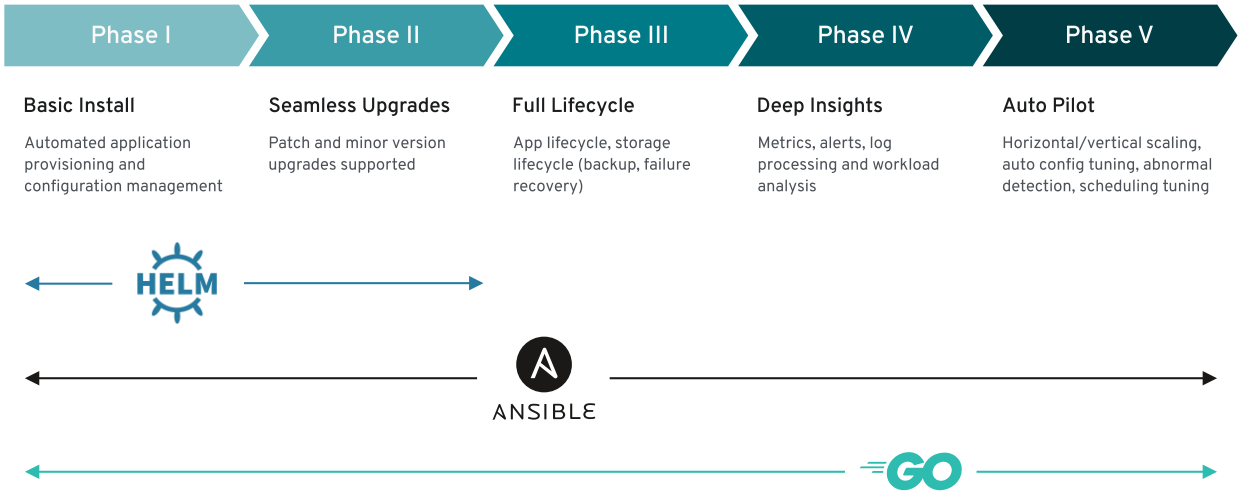
Since operators are basically custom controllers, they can built-in quite a different number of options available to build an operator. Some of them include *kooper* (https://github.com/spotahome/kooper), operator SDK, etc. Operator SDK is an official tool by Openshift to develop operators. Hence, for the scope of this project, we will focus on building operators using O*perator SDK*.

Operator SDK

Operator SDK project is a part of the **Operator Framework**, which is an open-source toolkit to manage operators or so-called Kubernetes native applications in an effective, scalable and automated manner. Operator SDK is a framework that uses the Kubernetes *controller-runtime* library in order to make writing operators easier. Operator SDK provides the following features (https://github.com/operator-framework/operator-sdk)

- High-level APIs and abstractions to write the operational logic more intuitively
- Tools for scaffolding and code generation to bootstrap a new project fast
- Extensions to cover common operator use cases

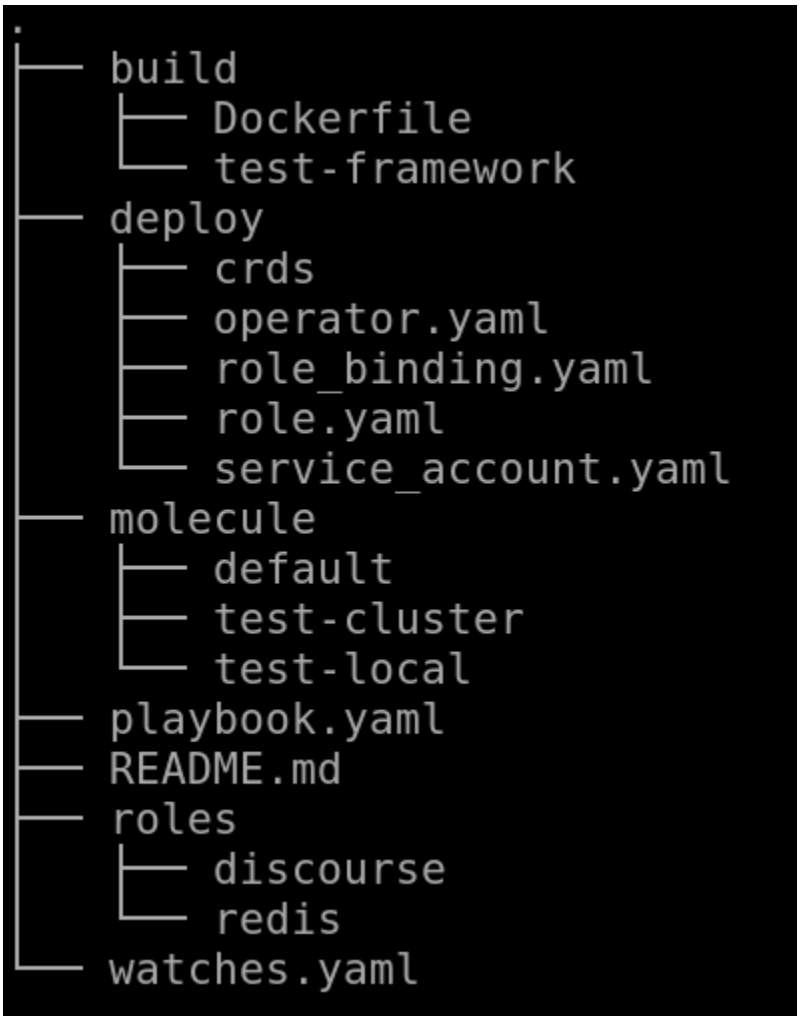
The operator SDK enables developers to build operator in either Go, Ansible or Helm. The capability level of each of these options is as shown in the figure below



For the scope of this project, we will focus on building an operator in Ansible using the operator SDK.

3. DISCOURSE OPERATOR

a. Operator Directory Structure



As shown in the picture above, the operator directory is divided into different sub-directories & files. Each of them is explained below

i. *build* – This folder contains the scripts for building the docker image of the operator
ii. *deploy* – This folder contains the generic set of deployment manifests, which are used to deploy the operator on a Kubernetes cluster
iii. *molecule* – This folder contains ansible files used for testing the operator
iv. *playbook.yaml* – This file is the main *playbook* file for the operator to deploy an instance. It contains the information about what all *roles* to run when requested for a specific instance
v. *roles* – All the roles listed in the *playbook.yaml* file are included in this with their respective sub-directories
vi. watches.yaml – This contains the information related to Group, Version, Kind, Ansible invocation method and Ansible configuration

b. Translating from a template to Operator

The usual Openshift template is a file that contains all the components required for an application. To build an operator, these individual components have to be consolidated into different folders as to how Ansible perceives it. The following steps will go through 'how to build an operator using a template'

i. Installing *operator-sdk*
The first step to building an operator in Ansible using operator SDK is to install the *operator-sdk* CLI (command-line tool). The tool can be further used to create a skeleton, build a docker image and others. The *operator-sdk* CLI can be installed using the following commands

```
# Set the release version variable
$ RELEASE_VERSION=v0.10.0
# Linux
$ curl -OJL https://github.com/operator-framework/operator-sdk/releases/download/${RELEASE_VERSION}/operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
$ cp operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu /usr/bin/
```

The installation can be verified by running the following command with executable *./operator-sdk –version*

ii. Initializing the operator

The operator can be initialized by running the following command

```
operator-sdk new discourse-operator --api-version=cache.example.com/v1alpha1 --kind=Discourse --type=ansible
```

The flag *kind* specifies the name of the CRD, while *type* specifies the option chosen to write the operator in.

iii. Creating the required files & directories

Now that we have the directory structure ready. The next step is to add manifest files and roles. For the Discourse operator, we have 2 sets of roles namely 'Discourse' and 'Redis'. In order to create the directory structure for each of these roles, I have used *ansible-galaxy*. *ansible-galaxy* is a command-line tool for installing, creating & managing roles. This comes installed with the ansible package. Although the directories & sub-directories can be created manually without the *ansible-galaxy*, this is a clean and efficient way of doing it.

In order to create the role directory structure, navigate to the *roles* in the root directory & run the following commands

```
ansible-galaxy init discourse
ansible-galaxy init redis
```

These commands will create directories *discourse* and *redis*, with many other sub-directories inside.

The next step is to add roles & change the configuration files.

a. Adding variables to *var/main.yml*

Add the following content to the file '*discourse-operator/roles/discourse/vars/main.yml*'.

```
db_host_value: "{{ lookup('env','db_host_value') }}"
db_port_value: "{{ lookup('env','db_port_value') }}"
db_name_value: "{{ lookup('env','db_name_value') }}"
db_username_value: "{{ lookup('env','db_username_value') }}"
db_password_value: "{{ lookup('env','db_password_value') }}"
developers_email_value: "{{ lookup('env','developer_emails_value') }}"
```

These can be considered as variables within the scope of all the ansible roles. In other words, these are the parameters to be given as input by the user to the operator in order to create an instance.

b. Adding tasks to *discourse/tasks*

From the openshift-template file, each of the discourse components namely configmaps, deploymentconfigs, persistent volumes, routes, and services are added as a separate file. Each of the file's contents is as follows

configmaps.yml

```
###################
# CONFIGMAPS      #
###################
- name: Create env configmap
  k8s:
    definition:
      kind: ConfigMap
      apiVersion: v1
      metadata:
        name: env-configmap
        namespace: '{{ namespace }}'
      data:
        DISCOURSE_DB_HOST_KEY: "{{ db_host_value }}"
        DISCOURSE_DB_PORT_KEY: "{{ db_port_value }}"
        DISCOURSE_DB_NAME_KEY: "{{ db_name_value }}"
        DISCOURSE_DB_USERNAME_KEY: "{{ db_username_value }}"
        DISCOURSE_DB_PASSWORD_KEY: "{{ db_password_value }}"
        DISCOURSE_DEVELOPER_EMAILS_KEY: "{{ developers_email_value }}"
        DISCOURSE_DB_POOL_KEY: "8"
        LANG: "en_US.UTF-8"
        SIDEKIQ_CONCURRENCY_KEY: "5"
        UNICORN_PID_PATH: "/var/run/unicorn.pid"
        UNICORN_PORT: "3000"
        UNICORN_SIDEKIQS: "1"
        UNICORN_WORKERS: "1"
        RUBY_GC_HEAP_GROWTH_MAX_SLOTS: "40000"
        RUBY_GC_HEAP_INIT_SLOTS: "400000"
        RUBY_GC_HEAP_OLDOBJECT_LIMIT_FACTOR: "1.5"

- name: Create nginx configmap
  k8s:
    namespace: '{{ namespace }}'
    resource_definition: "{{ lookup('file', 'nginx-configmap.yml') }}"

- name: Create discourse configmap
  k8s:
    namespace: '{{ namespace }}'
    resource_definition: "{{ lookup('file', 'discourse-configmap.yml') }}"
```

deploymentconfig.yml

```
###################
# DEPLOYMENTS     #
###################
  - name: Create discourse deployment config
    k8s:
      definition:
        kind: DeploymentConfig
        apiVersion: apps.openshift.io/v1
        metadata:
          labels:
            app: discourse-cern
          name: webapp
          namespace: '{{ namespace }}'
        spec:
          replicas: 1
          selector:
            app: discourse-cern
            deploymentconfig: webapp
          strategy:
            rollingParams:
              timeoutSeconds: 1200
            type: Rolling
          template:
            metadata:
              labels:
                app: discourse-cern
                deploymentconfig: webapp
            spec:
              containers:
              -
                name: nginx
                command:
                  - ./run-nginx.sh
                image: discourse-cern:v2.3.0
                imagePullPolicy: IfNotPresent
                ports:
                - containerPort: 8080
                  protocol: TCP
                resources:
                  limits:
                    memory: 400Mi
                    cpu: 200m
                  requests:
                    memory: 20Mi
                    cpu: 50m
                terminationMessagePath: /dev/termination-log
                volumeMounts:
                - mountPath: /discourse/public/uploads
                  name: discourse-uploads
                - mountPath: /discourse/public/assets
                  name: discourse-public-assets
                - mountPath: /discourse/public/backups
                  name: discourse-backups
                - mountPath: /tmp/nginx/
                  name: nginx-configmap
                - mountPath: /var/cache/nginx
                  name: var-cache-nginx
                readinessProbe:
                  failureThreshold: 3
                  initialDelaySeconds: 30
                  periodSeconds: 10
                  successThreshold: 1
                  tcpSocket:
                    port: 8080
                  timeoutSeconds: 10
                livenessProbe:
                  failureThreshold: 3
                  initialDelaySeconds: 10
```

pvc.yml

For this operator, I have used a local OKD cluster which comes with default PV's. Therefore, for this reason, *pvc* is not configured for now. When deploying to production, these roles need to be configured and added.

route.yml

```
###################
# ROUTES          #
###################
- name: Create route
  k8s:
    definition:
      kind: Route
      apiVersion: route.openshift.io/v1
      metadata:
        labels:
          app: discourse-cern
        name: nginx
        namespace: '{{ namespace }}'
      spec:
        port:
          targetPort: 8080-tcp
        to:
          kind: Service
          name: nginx
          weight: 100
        tls:
          termination: "edge"
          insecureEdgeTerminationPolicy: Redirect
```

services.yml

```
###################
# SERVICES      #
###################

- name: Create a nginx service
  k8s:
    definition:
      kind: Service
      apiVersion: v1
      metadata:
        labels:
          app: discourse-cern
        name: nginx
        namespace: '{{ namespace }}'
      spec:
        ports:
        - name: 8080-tcp
          port: 8080
          protocol: TCP
          targetPort: 8080
        selector:
          app: discourse-cern
          deploymentconfig: webapp
        sessionAffinity: None
        type: ClusterIP

- name: Create a webapp service
  k8s:
    definition:
      kind: Service
      apiVersion: v1
      metadata:
        labels:
          app: discourse-cern
        name: webapp
        namespace: '{{ namespace }}'
      spec:
        ports:
        - name: 3000-tcp
          port: 3000
          protocol: TCP
          targetPort: 3000
        selector:
          app: discourse-cern
          deploymentconfig: webapp
        sessionAffinity: None
        type: ClusterIP
```

Configuring the *main.yml* in *roles/discourse/tasks*

Now that we have added all the ansible roles, all these roles have to be connected in the *main.yml* file. The contents of the file are as follows

```
#############################################################################
###
## Provision Discourse site
#############################################################################
###

- name: Provision Discourse site
  block:

    - name: Configure Configmaps
      include_tasks: configmaps.yml

    # - name: Configure PVCs
    #   include_tasks: pvc.yml

    - name: Configure DeploymentConfig
      include_tasks: deploymentconfig.yml

    - name: Configure Services
      include_tasks: services.yml

    - name: Configure Routes
      include_tasks: route.yml

  rescue:
    #- include_role:
    #    name: mail-handler
    #    tasks_from: failure

    - name: Writing Termination Message '/dev/termination-log'
      shell: >
        echo "Failed task -> {{ ansible_failed_task.name }}.
        Error was -> {{ ansible_failed_result.msg }}"
        > /dev/termination-log

    - fail:
        msg: "Error in task {{ ansible_failed_task.name }}: {{
ansible_failed_result.msg }}"
```

c. Configmaps of *discourse* roles

Since the configmaps are huge to put in a single file, for the purpose of brevity they are separated into two files *discourse-configmap.yml* and *nginx-configmap.yml* in the directory *discourse-operator/roles/discourse/files*. And these files are configured respectively the configmaps roles i.e *discourse-operator/roles/discourse/tasks/configmaps.yml*

There are no changes required besides these in the discourse roles.

d. Like the discourse roles, config files have to be added to the Redis roles directory. The tasks in the Redis roles are as follows

deploymentconfig.yml

```
- name: Create a redis deployment config
  k8s:
    definition:
      - kind: DeploymentConfig
        apiVersion: apps.openshift.io/v1
        metadata:
          labels:
            app: discourse-cern
          name: redis
          namespace: '{{ namespace }}'
        spec:
          replicas: 1
          selector:
            app: discourse-cern
            deploymentconfig: redis
          strategy:
            type: Rolling
          template:
            metadata:
              labels:
                app: discourse-cern
                deploymentconfig: redis
            spec:
              containers:
              - image: ' '
                imagePullPolicy: Always
                name: redis
                ports:
                - containerPort: 6379
                  protocol: TCP
                resources: {}
                terminationMessagePath: /dev/termination-log
                volumeMounts:
                - mountPath: /var/lib/redis/data
                  name: redis-1
                readinessProbe:
                  failureThreshold: 3
                  initialDelaySeconds: 5
                  periodSeconds: 10
                  successThreshold: 1
                  tcpSocket:
                    port: 6379
                  timeoutSeconds: 10
                livenessProbe:
                  failureThreshold: 3
                  initialDelaySeconds: 30
                  periodSeconds: 10
                  successThreshold: 1
                  tcpSocket:
                    port: 6379
                  timeoutSeconds: 10
              dnsPolicy: ClusterFirst
              restartPolicy: Always
              securityContext: {}
              volumes:
              - emptyDir: {}
                name: redis-1
          test: false
          triggers:
          - type: ConfigChange
          - type: ImageChange
```

service.yml

```
- name: Create a redis service
  k8s:
    definition:
      kind: Service
      apiVersion: v1
      metadata:
        labels:
          app: discourse-cern
          service: redis
        name: redis
        namespace: '{{ namespace }}'
      spec:
        type: ClusterIP
        sessionAffinity: None
        ports:
        - name: 6379-tcp
          port: 6379
          protocol: TCP
          targetPort: 6379
        selector:
          app: discourse-cern
          deploymentconfig: redis
```

The *main.yml* also needs to be configured specifying the tasks to run

```
############################################################################
###
## Provision Redis
############################################################################
###

- name: Provision Redis
  block:

    - name: Provision Redis DeploymentConfig
      include_tasks: deploymentconfig.yml

    - name: Provision Redis Service
      include_tasks: service.yml

  rescue:
    #- include_role:
    #    name: mail-handler
    #    tasks_from: failure

    - name: Writing Termination Message '/dev/termination-log'
      shell: >
        echo "Failed task -> {{ ansible_failed_task.name }}.
        Error was -> {{ ansible_failed_result.msg }}"
        > /dev/termination-log

    - fail:
        msg: "Error in task {{ ansible_failed_task.name }}: {{
ansible_failed_result.msg }}"
```

e. Integrating all the roles in the *watches.yaml* file & *playbook.yaml* file

The *playbook.yaml* file should contain all roles that are to be run by the operator. The file simply lists all the roles, which in our case are *discourse* and *redis*.
The contents of the file are follows

```
- hosts: localhost
  tasks:
  - debug: msg="Running Discourse Operator Playbook"
  - import_role:
      name: "redis"
  - import_role:
      name: "discourse"
```

The *watches.yaml*, besides Ansible configuration, should also include the path of the *playbook.yaml* file. The *playbook.yaml* file path should be relative to the docker image, i.e the path to which the file is copied to when building the Docker image. The contents of the file are as follows

```
---
- version: v1alpha1
  group: discourse.cern
  kind: Discourse
  playbook: /opt/ansible/playbook.yaml
  # reconcilePeriod: 5m
  # manageStatus: false
  watchDependentResources: False
  # role: /opt/ansible/roles/discourse
```

f. Modifying the Dockerfile to include all the roles & tasks

The *Dockerfile* present in the directory *discourse-operator/build/* should look as follows

```
FROM quay.io/operator-framework/ansible-operator:v0.9.0

COPY watches.yaml ${HOME}/watches.yaml

COPY roles/ ${HOME}/roles/

COPY playbook.yaml ${HOME}/playbook.yaml
```

g. Adding manifest files for the operator deployment

Now that we have added logic of what to do when a Discourse instance if requested, we have to now add Kubernetes manifest files that will deploy the Discourse operator on the cluster, which will then manage all the instances for us. These files are auto-generated by the *operator-sdk* in the *deploy* directory. The files that are created are as follows

I. crds
    a. discourse_v1alpha1_discourse_cr.yaml
    b. discourse_v1alpha1_discourse_crd.yaml
II. imagestream.yaml
III. operator.yaml
IV. role.yaml
V. role-binding.yaml
VI. service_account.yaml

The files that are to be modified are *operator.yml* and *crds/discourse_v1alpha1_discourse_cr.yaml*.

In the *operator.yml*, only the name of the image and imagePullPolicy have to be modified. These parameters will be discussed in the next sub-section

The *crds/discourse_v1alpha1_discourse_cr.yaml* has to be modified to include the input parameters that are to be sent to the operator when creating an instance. The contents of this file are follows

```
apiVersion: discourse.cern/v1alpha1
kind: Discourse
metadata:
  name: example-discourse
spec:
  # Add fields here
  size: 3
  namespace: discourse-operator
  version: latest
  category: personal
  db_host_value: <host>
  db_name_value: <db_name>
  db_username_value: <username>
  db_password_value: <password>
  developers_email_value: rajula.vineet.reddy@cern.ch
```

iv. Building the operator image

Once all the files are put in the respective directories, the docker image can be built by running the command *operator-sdk build <image_name>*. This will use the *Dockerfile* from the *deploy* folder in order to build the image. In order to automate this, I have integrated GitLab CI/CD along with SVC to build the image. The setup is in such a way that, all my code for the operator is hosted on GitLab, and whenever I make a commit with a **tag**, GitLab triggers it's CI, which will then build the Docker image in its runner and stores the image in its **Registry Server**. I can then use the image URL from the registry and configure it in the *deploy/operator.yml* file.

The configuration of *gitlab-ci.yml*, The GitLab CI files is as follows

```
stages:
  - build

build docker image with host daemon:
  tags:
  - docker-privileged
  stage: build
  image: docker:latest
  script:
    - docker info
    - docker login -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD"
$CI_REGISTRY
    - docker build --pull -f $CI_PROJECT_DIR/build/Dockerfile -t
"$CI_REGISTRY_IMAGE" .
    - docker push "$CI_REGISTRY_IMAGE"
  only:
    - tags
```

v. Running the operator

Once the operator image is built & stored in a local/ remote registry, using the image URL and the manifest files, the operator can be deployed. Detailed steps to deploy an operator are as follows

1. Clone the repository containing the operator files

```
git clone https://gitlab.cern.ch/rvineetr/discourse-operator.git
cd discourse-operator
```

2. Create the CRD (Custom Resource Definition)

```
oc create -f deploy/crds/discourse_v1alpha1_discourse_crd.yaml
```

3. Create all the components of the operator in a new project

```
oc new-project discourse-operator
oc create -f deploy/
```

4. Give admin permissions to the operator user

```
oc adm policy add-cluster-role-to-user admin -z discourse-operator
```

5. Verify if the operator is running

```
oc get pods
oc logs <pod_name> -c operator -f
```

vi. Provisioning an instance

Now that the operator is running, an instance can be created using the operator. To create an instance, the CR (Custom Resource) file *deploy/crds/discourse_v1alpha1_discourse_cr.yaml* has to be executed. Before running it, the parameters have to be configured. For now, since there is no way to get a database server, I have started a PostgreSQL server locally and have configured it with the CR. Once I have the CR ready, it can be created by running the command `oc create -f deploy/crds/discource_v1alpha1_discource_cr.yaml` This will trigger the operator, which will then create an instance using the ansible roles provided by us. It is important to configure the namespace parameter in the CR. The CR will be deployed in the namespace specified.

4. DATABASE OPERATOR

Although I was able to deploy the Discourse operator with a local PostgreSQL server, it is not the right way of doing this. Upon discussion with my supervisor, we found some issues with the database services. In most of the cases, it is either to manually create a PostgreSQL server locally or in the Openshift infra, and then configure all the backends with the address of the database server. This comes with many issues. All the database servers are isolated from the actual webservers, they require maintenance, backups, upgrade, etc. Since they are deployed independent of the actual webservers, upon deletion of the webservers the database servers must be manually deleted. There is a lot of manual work going into this. To automate this, we came up with trying to deploy a database operator which will provision database instances and use the *finalizers* option in *operator-sdk*, which acts like a webhook and upon deletion of an instance *operator-sdk*

will run the roles specified in the *finalizers* field. The roles in the *finalizers* field in an operator can be in such a way that they delete the database by talking to the database operator upon termination request to itself.

Instead of writing an operator for the database from scratch, we have decided to explore existing open-source options from *Operatorhub.io*. There were 2-3 options available for PostgreSQL. After trying to deploy one of them namely **Postgres-Operator** by Zalanado, I have realized the architecture of this operator was complex and there were compatibility issues with the OLM (Operator Lifecycle Manager) and the OKD 3.11 version. So, I have moved on to a different option. The other option was **Crunchy PostgreSQL Enterprise** by Crunchy Data. This one had a good amount of documentation and support from the community. After trying to deploy an older version 3.9.x for about a week I ended with a lot of bugs. Later found the documentation & release of a newer version 4.0.0, which worked perfectly. I succeeded in deploying the postgres-operator and then connecting it with the Discourse operator. Although, there were some Openshift issues I encountered during this process, was able to resolve/ bypass them by taking to Alexandre Lossent and Iago Santos Pardo. More about these issues in the Miscellaneous section.

This operator has a CLI client, that comes with the operator, which is mandatory to communicate with the postgres-operator to create/ delete/ manage instances. The steps to deploy this Crunchy PostgreSQL operator are as follows

1. Create a directory to clone files `mkdir -p $HOME/odev/src/github.com/crunchydata $HOME/odev/bin $HOME/odev/pkg`
2. Navigate to the newly created directory `cd $HOME/odev/src/github.com/crunchydata`
3. Clone the operator `git clone https://github.com/CrunchyData/postgres-operator.git`
4. cd into the directory `cd postgres-operator`
5. Checkout to the 4.0.0. branch `git checkout 4.0.0`
6. Setup the environment with the necessary paths to the binaries `cat $HOME/odev/src/github.com/crunchydata/postgres-operator/examples/envs.sh >> $HOME/.bashrc`
7. Activate the new environment `source $HOME/.bashrc`
8. Export the namespace names for the postgres clusters `export NAMESPACE=pgouser1,pgouser2`
9. Export the namespace name in which the postgres-operator is deployed `export PGO_OPERATOR_NAMESPACE=pgo`
10. Create the namespaces using the variables setup `make setupnamespaces`
11. Change the storage options in `conf/postgresql-operator/pgo.yaml` to `hostpathstorage`
    Change from

```
PrimaryStorage: storageos
BackupStorage: storageos
ReplicaStorage: storageos
BackrestStorage: storageos
```

    to

```
PrimaryStorage: hostpathstorage
BackupStorage: hostpathstorage
ReplicaStorage: hostpathstorage
BackrestStorage: hostpathstorage
```

12. Install expenv, which is a dependency for the makefile in the next steps
```
wget
https://github.com/blang/expenv/releases/download/v1.2.0/expenv_amd64.ta
r.gz
tar -xzf expenv_amd64.tar.gz expenv
cp expenv /usr/bin
```
13. Copy the files required to create the postgres instance in the later steps

```
cp ./conf/postgres-operator/pgouser $HOME/.pgouser
```

```
cp ./conf/postgres-operator/pgorole $HOME/.pgorole
```

14. Install the RBACs `make installrbac`
15. Deploy the operator `make deployoperator`
16. Install the CLI client which is required in order to communicate with postgres-operator

    `wget https://github.com/CrunchyData/postgres-operator/releases/download/4.0.0/pgo -O /usr/bin/pgo` - Since the version of postgres operator is 4.0.0

    `chmod 777 /usr/bin/pgo` - Give executable permissions to the downloaded 'pgo' file

17. Export the API Server of the operator using the IP of the POD into the environment

```
export PGO_APISERVER_URL=https://$(oc get svc postgres-operator -n pgo -
o jsonpath='{range.spec}{.clusterIP}'):8443
```

18. Once the API server is running and everything is installed, the version of the client and server can be verified by running

```
pgo version
```

19. Create a new cluster in one the instance namespaces namely *pgouser1*

```
pgo create cluster mycluster -n pgouser1
```

20. View the cluster details

```
pgo show cluster mycluster -n pgouser1
```

21. `pgo create user user1 --selector=name=mycluster --pasword=newpass` - Create a new user with a password
22. `pgo user --change-password=postgres --selector=name=mycluster --password=newpass` - Updated an existing user with a given password
23. `pgo test mycluster -n pgouser1` - Testing the cluster
24. `pgo scale mycluster -n pgouser1` - Scaling the cluster

Once the PostgreSQL instances are provisioned, using the service name of the pod in the *pgouser1* namespace, and the dbname, username and the password, these can be configured in the CR and a new discourse instance can be provisioned. This can be automated by putting all these steps in a single shell script, which itself will provision a PostgreSQL instance, use the details and then provision a Discourse instance. All using **operators**. Or the provisioning of the PostgreSQL instance can be set up inside the Discourse operator, were using Ansible roles PostgreSQL instance are provisioned first and then using it the Discourse instance.

5. MISCELLANEOUS

a. Setting up Openshift cluster

Initially, I started to deploy my own cluster to build the operator. There were quite a few options for deploying the cluster. With the help of one of my teammate Iago, I was able to set up a local cluster on one of the CERN cloud virtual machine using the scripts from here https://github.com/gshipley/installcentos. This setup was simple and was working great and it also comes with a domain name for the cluster console, which can be accessed on the local network. As I started working, I was having persistent volume errors with the Discourse operator initially. After discussing with Ismael, we have decided to not use persistent volumes for now, as it was little complex and I can just deploy it without any persistent data. As the work progressed, I had a similar issue when trying to install the Postgresql operator. Instead of skipping it, this time I tried to debug the issue. After trying for a couple of days, I have concluded that the error was because of the SCC (Security Configuration) of the pod. I asked Alex & Iago for their feedback in this, and Alex confirmed that the PV's work on the CentOS with this method and the problem was with the CentOS. Following his suggestion, I have tried reinstalling the cluster using the usual 'oc cluster up' method, as this method generates a set of persistent volumes by default with the cluster installation. This setup worked successfully. Following this setup I wasn't able to access the cluster console using the IP or domain name in the local network. After trying a few methods like port-forwarding and hostname settings. I ended up doing the SSH tunneling (https://github.com/openshift/origin/issues/19699#issuecomment-434367748) using the command *sudo ssh -L 8443:localhost:8443 -f -N user@host*. Also, following this setup by default using any random text as username and password, a user can login. But in order to login as administrator or to give a user admin rights so as to access all the projects, the following command has to be executed *oc adm policy add-cluster-role-to-user cluster-admin <username>*

b. Building Docker images on Gitlab CI using Kaniko

Kaniko is a build method in Gitlab, which builds docker images inside a Kubernetes pod. To build the operator image on Gitlab, I started with Kaniko as it is a new and clean way of building the images. But, I kept running into errors. The error was related to 'cannot resolve image'. The error logs can be found at https://gitlab.cern.ch/rvineetr/discourse-operator/-/jobs/4774013. This error is because my Dockerfile for the operator is being built on a base image from quay.io. Kaniko was having errors to resolve the image from this registry. Given the time constraint, moved to building the images using the traditional method 'docker image with host daemon'.

c. OLM & OKD version issues

Before finalizing on the CrunchyData postgresql operator, I was trying out this other operator called 'Postgres-Operator' by Zalando SE. Both these were operators are available on https://operatorhub.io/. Besides installing these operators by manifest files, they can also be installed using OLM, also called Operator Lifecycle Manager. OLM is part of the operator-framework and needs to be installed separately on OKD 3.x. While it comes pre-installed on OKD 4.x. Although I did try to install OKD 4.0 using the crc containers method, but couldn't do it. So, I tried installing OLM on OKD 3.x and tried to install these operators. There were weird errors using this method. When I tried to delete using OLM i.e by replacing 'create' with 'delete' in the install command, the commands execute completely but doesn't delete all the components. In order to delete the components, the whole project has to be deleted. Also, if the project is deleted manually, the project gets stuck in a 'terminating state'. This can be resolved by following a procedure similar to this https://stackoverflow.com/a/52412965. From what I understood, this is mostly an issue with the version compatibility of OLM and OKD. Probably, OLM should work better with OKD 4.x.

6. FUTURE WORK

There is a lot that can be improved in this project. Firstly, it would sense to complete the triggering of PostgreSQL operator from inside of the Discourse operator, which will make it automatic. There is also scope to add tests to the Ansible roles & the operator. The tests can be to test the roles or something like a ChaosMonkey test. There is also more that can be improved. The operator can be configured in such a way that, during the upgrading of all the instances, the operator can run database backups and then run the upgrade, everything automatically. Database backup is again like a business logic for the stateful app, which can be put as a functionality inside the operator. Also, Discourse upstream allows hooking our Discourse repository to the test-passed branches. This branch is updated every now and then. With proper tests, the operator can be configured to deploy the last changes on this branch.

7. CONCLUSION

With this project, I have successfully built a Discourse Operator which makes the procedure to deploy and manage discourse instances easier as a part of the Discourse at CERN service. The operator reduces a lot of manual involvement while maintaining the instances. By deploying an operator for the database, it became easier to create a database instance and integrate them with other operators. The documentation for deploying the database operator and the documentation for 'translating a template to an operator' will be of great use to the team to develop and port other services to Operators.

8. ACKNOWLEDGMENTS

I would like to start by thanking my supervisors Ismael and Andreas for their guidance and support throughout the project. Ismael went the extra mile at times in order to help me. He has been an amazing mentor. I would like to the thank other members of the team, Eduardo, Pete, Alex, Iago, Daniel for answering my questions and solving my errors. I would also like to thank the whole team for BBQ's and everything. I had so fun and learned a lot. The whole experience was amazing. That being said, I am grateful to the Cern Openlab Team & Cern in general for this opportunity. Thank you for everything <3.