





Операторы, CustomResourceDefiniti on

Что с нами будет?

- Ветка для работы: `kubernetes-operators`
- В ходе работы мы:
 - Напишем CustomResource и CustomResourceDefinition для mysql оператора
 -  Напишем часть логики mysql оператора при помощи python KOPF
 - Сделаем соберем образ и сделаем деплой оператора.
- Если делаете часть с , нужно поставить *label*  на Pull request

В данной работе есть задачи в которых необходимо будет программировать на python, они необязательные (в заголовке слайда отражены знаком ).

Подготовка

- Запустите kubernetes кластер в `minikube`
- Создадим директорию `kubernetes-operators/deploy`:

```
mkdir -p kubernetes-operators/deploy && cd kubernetes-operators
```



В ходе работы понадобится python и различные зависимости

Что должно быть в описании MySQL

Для создания pod с MySQL оператору понадобится знать:

1. Какой образ с MySQL использовать
2. Какую db создать
3. Какой пароль задать для доступа к MySQL

То есть мы бы хотели, чтобы описание MySQL выглядело примерно так:

```
apiVersion: otus.homework/v1
kind: MySQL
metadata:
  name: mysql-instance
spec:
  image: mysql:5.7
  database: otus-database
  password: otuspassword # Так делать не нужно, следует использовать secret
  storage_size: 1Gi
```

CustomResource

Создадим CustomResource `deploy/cr.yml` со следующим содержимым:

```
apiVersion: otus.homework/v1
kind: MySQL
metadata:
  name: mysql-instance
spec:
  image: mysql:5.7
  database: otus-database
  password: otuspassword # Так делать не нужно, следует использовать secret
  storage_size: 1Gi
  useless_data: "useless info"
```

 gist

CustomResource

Пробуем применить его:

```
kubectl apply -f deploy/cr.yml
```

Видим ошибку:

```
error: unable to recognize "deploy/cr.yml": no matches for kind "MySQL" in version  
"otus.homework/v1"
```

Ошибка связана с отсутствием объектов типа MySQL в API kubernetes.
Исправим это недоразумение.

CustomResourceDefinition

`CustomResourceDefinition` - это `ресурс` для определения других `ресурсов` (далее CRD)

Создадим CRD `deploy/crd.yml` [gist](#):

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: mysqls.otus.homework # имя CRD должно иметь формат plural.group
spec:
  scope: Namespaced          # Данный CRD будер работать в рамках namespace
  group: otus.homework       # Группа, отражается в поле apiVersion CR
  versions:                  # Список версий
    - name: v1
      served: true            # Будет ли обслуживаться API-сервером данная версия
      storage: true           # Версия описания, которая будет сохраняться в etcd
  names:                     # различные форматы имени объекта CR
    kind: MySQL              # kind CR
    plural: mysqls
    singular: mysql
    shortNames:
      - ms
```

Создаем CRD и CR

Создадим CRD:

```
kubectl apply -f deploy/crd.yml
```

```
customresourcedefinition.apiextensions.k8s.io/mysqls.otus.homework created
```

Создаем CR:

```
kubectl apply -f deploy/cr.yml
```

```
mysql.otus.homework/mysql-instance created
```


Взаимодействие с объектами CR CRD

С созданными объектами можно взаимодействовать через kubectl:

```
kubectl get crd  
kubectl get mysqls.otus.homework  
kubectl describe mysqls.otus.homework mysql-instance  
...
```

Validation

На данный момент мы никак не описали схему нашего CustomResource. Объекты типа `mysql` могут иметь абсолютно произвольные поля, нам бы хотелось этого избежать, для этого будем использовать `validation`. Для начала удалим CR `mysql-instance`:

```
kubectl delete mysqls.otus.homework mysql-instance
```

Добавим в спецификацию CRD (`spec`) параметры `validation` [gist](#)

Validation

```
validation:
  openAPIV3Schema:
    type: object
    properties:
      apiVersion:
        type: string # Тип данных поля ApiVersion
      kind:
        type: string # Тип данных поля kind
      metadata:
        type: object # Тип поля metadata
        properties: # Доступные параметры и их тип данных поля metadata (словарь)
          name:
            type: string
      spec:
        type: object
        properties:
          image:
            type: string
          database:
            type: string
          password:
            type: string
          storage_size:
            type: string
```

Пробуем применить CRD и CR

```
kubectl apply -f deploy/crd.yml
```

```
kubectl apply -f deploy/cr.yml
```

```
error: error validating "deploy/cr.yml": error validating data:  
ValidationError(MySQL): unknown field "usless_data" in homework.otus.v1.MySQL; if  
you choose to ignore these errors, turn validation off with --validate=false
```

Убираем из cr.yml:

```
usless_data: "useless info"
```

Применяем:

```
kubectl apply -f deploy/cr.yml
```


Ошибки больше нет

Задание по CRD:

Если сейчас из описания mysql убрать строчку из спецификации, то манифест будет принят API сервером. Для того, чтобы этого избежать, добавьте описание обязательных полей в CustomResourceDefinition

| Подсказка. Пример есть в лекции.

Операторы

- **Оператор** включает в себя **CustomResourceDefinition** и **custom controller**
 - CRD содержит описание объектов CR
 - Контроллер следит за объектами определенного типа, и осуществляет всю логику работы оператора
- CRD мы уже создали далее будем писать свой контроллер (все задания по написанию контроллера дополнительными)
- Далее развернем custom controller:
 - Если вы делаете задания с , то ваш
 - Если нет, то используем готовый контроллер

Описание контроллера

Используемый/написанный нами контроллер будет обрабатывать два типа событий:

1) При создании объекта типа (`kind: mySQL`), он будет:

- * Создавать `PersistentVolume`, `PersistentVolumeClaim`, `Deployment`, `Service` для `mysql`
- * Создавать `PersistentVolume`, `PersistentVolumeClaim` для бэкапов базы данных, если их еще нет.
- * Пытаться восстановиться из бэкапа

2) При удалении объекта типа (`kind: mySQL`), он будет:

- * Удалять все успешно завершённые `backup-job` и `restore-job`
- * Удалять `PersistentVolume`, `PersistentVolumeClaim`, `Deployment`, `Service` для `mysql`

MySQL контроллер

В папке `kubernetes-operators/build` создайте файл `mysql-operator.py`. Для написания контроллера будем использовать kopf.

Добавим в него импорт необходимых библиотек

```
import kopf
import yaml
import kubernetes
import time
from jinja2 import Environment, FileSystemLoader
```


MySQL контроллер

- В директории `kubernetes-operators/build/templates` создайте шаблоны:
 - `mysql-deployment.yml.j2` [gist](#)
 - `mysql-service.yml.j2` [gist](#)
 - `mysql-pv.yml.j2` [gist](#)
 - `mysql-pvc.yml.j2` [gist](#)
 - `backup-pv.yml.j2` [gist](#)
 - `backup-pvc.yml.j2` [gist](#)
 - `backup-job.yml.j2` [gist](#)
 - `restore-job.yml.j2` [gist](#)

MySQL контроллер

Добавим функцию, для обработки Jinja шаблонов и преобразования YAML в JSON:

```
def render_template(filename, vars_dict):  
    env = Environment(loader=FileSystemLoader('./templates'))  
    template = env.get_template(filename)  
    yaml_manifest = template.render(vars_dict)  
    json_manifest = yaml.load(yaml_manifest)  
    return json_manifest
```

MySQL контроллер

Ниже добавим декоратор:

```
@kopf.on.create('otus.homework', 'v1', 'mysqls')
# Функция, которая будет запускаться при создании объектов тип MySQL:
def mysql_on_create(body, spec, **kwargs):
    name = body['metadata']['name']
    image = body['spec']['image'] # сохраняем в переменные содержимое описания
MySQL из CR
    password = body['spec']['password']
    database = body['spec']['database']
    storage_size = body['spec']['storage_size']
```

Функция `mysql_on_create` будет запускаться при создании объектов типа MySQL.

MySQL контроллер

Добавим в декоратор рендер шаблонов:

```
# Генерируем JSON манифесты для деплоя
persistent_volume = render_template('mysql-pv.yml.j2',
                                    {'name': name,
                                     'storage_size': storage_size})
persistent_volume_claim = render_template('mysql-pvc.yml.j2',
                                          {'name': name,
                                           'storage_size': storage_size})
service = render_template('mysql-service.yml.j2', {'name': name})

deployment = render_template('mysql-deployment.yml.j2', {
    'name': name,
    'image': image,
    'password': password,
    'database': database})
```

MySQL контроллер

Для создания объектов пользуемся библиотекой kubernetes:

```
api = kubernetes.client.CoreV1Api()
# Создаем mysql PV:
api.create_persistent_volume(persistent_volume)
# Создаем mysql PVC:
api.create_namespaced_persistent_volume_claim('default',
persistent_volume_claim)
# Создаем mysql SVC:
api.create_namespaced_service('default', service)

# Создаем mysql Deployment:
api = kubernetes.client.AppsV1Api()
api.create_namespaced_deployment('default', deployment)
```

MySQL контроллер

Сейчас должно получиться, что-то похожее на [gist](#)

С такой конфигурации уже должны обрабатываться события при создании `cr.yml`, проверим, для этого из папки `build`:

```
kopf run mysql-operator.py
```

Если `cr.yml` был до этого применен, то вы увидите:

```
[2019-09-16 22:47:33,662] kopf.objects          [INFO    ] [default/mysql-instance]
Handler 'mysql_on_create' succeeded.
[2019-09-16 22:47:33,662] kopf.objects          [INFO    ] [default/mysql-instance]
All handlers succeeded for creation.
```

Вопрос: почему объект создался, хотя мы создали CR, до того, как запустили контроллер?

MySQL контроллер

Если сделать `kubectl delete mysqls.otus.homework mysql-instance`, то CustomResource будет удален, но наш контроллер ничего не сделает т. к обработки событий на удаление у нас нет.

Удалим все ресурсы, созданные контроллером:

```
kubectl delete mysqls.otus.homework mysql-instance
kubectl delete deployments.apps mysql-instance
kubectl delete pvc mysql-instance-pvc
kubectl delete pv mysql-instance-pv
kubectl delete svc mysql-instance
```

MySQL контроллер

Для того, чтобы обработать событие удаления ресурса используется другой декоратор, в нем можно описать удаление ресурсов, аналогично тому, как мы их создавали, но есть более удобный метод.

Для удаления ресурсов, сделаем deployment,svc,pv,pvc дочерними ресурсами к mysql, для этого в тело функции `mysql_on_create`, после генерации json манифестов добавим:

```
# Определяем, что созданные ресурсы являются дочерними к управляемому
CustomResource:
    kopf.append_owner_reference(persistent_volume, owner=body)
    kopf.append_owner_reference(persistent_volume_claim, owner=body) # adopt
    kopf.append_owner_reference(service, owner=body)
    kopf.append_owner_reference(deployment, owner=body)
    # ^ Таким образом при удалении CR удалятся все, связанные с ним pv,pvc,svc,
    deployments
```


MySQL контроллер

В конец файла добавим обработку события удаления ресурса mysql:

```
@kopf.on.delete('otus.homework', 'v1', 'mysqls')
def delete_object_make_backup(body, **kwargs):
    return {'message': "mysql and its children resources deleted"}
```

Перезапустите контроллер, создайте и удалите mysql-instance, проверьте, что все pv, pvc, svc и deployments удалились.

Актуальное состояние контроллера можно подсмотреть в [gist](#)

MySQL контроллер

Теперь добавим создание pv, pvc для backup и restore job. Для этого после создания deployment добавим следующий код:

```
# Создаем PVC и PV для бэкапов:
try:
    backup_pv = render_template('backup-pv.yml.j2', {'name': name})
    api = kubernetes.client.CoreV1Api()
    api.create_persistent_volume(backup_pv)
except kubernetes.client.rest.ApiException:
    pass

try:
    backup_pvc = render_template('backup-pvc.yml.j2', {'name': name})
    api = kubernetes.client.CoreV1Api()
    api.create_namespaced_persistent_volume_claim('default', backup_pvc)
except kubernetes.client.rest.ApiException:
    pass
```

MySQL контроллер

Конструкция try, except - это обработка исключений, в данном случае, нужна, чтобы наш контроллер не пытался бесконечно пересоздать pv и pvc для бэкапов, т к их жизненный цикл отличен от жизненного цикла mysql.

Далее нам необходимо реализовать создание бэкапов и восстановление из них. Для этого будут использоваться Job. Поскольку при запуске Job, повторно ее запустить нельзя, нам нужно реализовать логику удаления успешно законченных jobs с определенным именем.

...

MySQL контроллер

Для этого выше всех обработчиков событий (под функций `render_template`) добавим следующую функцию:

```
def delete_success_jobs(mysql_instance_name):
    api = kubernetes.client.BatchV1Api()
    jobs = api.list_namespaced_job('default')
    for job in jobs.items:
        jobname = job.metadata.name
        if (jobname == f"backup-{mysql_instance_name}-job"):
            if job.status.succeeded == 1:
                api.delete_namespaced_job(jobname,
                                           'default',
                                           propagation_policy='Background')
```

MySQL контроллер

Также нам понадобится функция, для ожидания пока наша backup job завершится, чтобы дождаться пока backup выполнится перед удалением mysql deployment, svc, pv, pvc.

Опишем ее:

```
def wait_until_job_end(jobname):  
    api = kubernetes.client.BatchV1Api()  
    job_finished = False  
    jobs = api.list_namespaced_job('default')  
    while (not job_finished) and \  
        any(job.metadata.name == jobname for job in jobs.items):  
        time.sleep(1)  
        jobs = api.list_namespaced_job('default')  
        for job in jobs.items:  
            if job.metadata.name == jobname:  
                if job.status.succeeded == 1:  
                    job_finished = True
```

Добавим запуск backup-job и удаление выполненных jobs в функцию

`delete_object_make_backup`:

```
name = body['metadata']['name']
image = body['spec']['image']
password = body['spec']['password']
database = body['spec']['database']

delete_success_jobs(name)
# Создаем backup job:
api = kubernetes.client.BatchV1Api()
backup_job = render_template('backup-job.yml.j2', {
    'name': name,
    'image': image,
    'password': password,
    'database': database})
api.create_namespaced_job('default', backup_job)
wait_until_job_end(f"backup-{name}-job")
```

Актуальное состояние контроллера [gist](#)

MySQL контроллер

Добавим генерацию json из шаблона для restore-job

```
restore_job = render_template('restore-job.yml.j2', {  
    'name': name,  
    'image': image,  
    'password': password,  
    'database': database})
```

Добавим попытку восстановиться из бэкапов после deployment mysql:

```
# Пытаемся восстановиться из backup  
try:  
    api = kubernetes.client.BatchV1Api()  
    api.create_namespaced_job('default', restore_job)  
except kubernetes.client.rest.ApiException:  
    pass
```

MySQL контроллер

Добавим зависимость restore-job от объектов mysql (возле других owner_reference):

```
kopf.append_owner_reference(restore_job, owner=body)
```

Вот и готово. Запускаем оператор (из директории build):

```
kopf run mysql-operator.py
```

Создаем CR:

```
kubectl apply -f deploy/cr.yml
```

Актуальное состояние контроллера [gist](#)

MySQL контроллер

Проверяем что появились pvc:

```
kubectl get pvc
```

NAME		STATUS	VOLUME	
CAPACITY	ACCESS MODES	STORAGECLASS	AGE	
backup-mysql-instance-pvc		Bound	pvc-22eace9a-89e6-4926-8949-cc62cb6489af	1Gi
RWO	standard	35s		
mysql-instance-pvc		Bound	pvc-b7d25705-15d7-49a5-97cb-aeccd938e611	1Gi
RWO	standard	35s		

MySQL контроллер

Проверим, что все работает, для этого заполним базу созданного mysql-instance:

```
export MYSQLPOD=$(kubectl get pods -l app=mysql-instance -o jsonpath="{.items[*].metadata.name}")
```

```
kubectl exec -it $MYSQLPOD -- mysql -u root -potuspassword -e "CREATE TABLE test ( id smallint unsigned not null auto_increment, name varchar(20) not null, constraint pk_example primary key (id) );" otus-database
```

```
kubectl exec -it $MYSQLPOD -- mysql -potuspassword -e "INSERT INTO test ( id, name ) VALUES ( null, 'some data' );" otus-database
```

```
kubectl exec -it $MYSQLPOD -- mysql -potuspassword -e "INSERT INTO test ( id, name ) VALUES ( null, 'some data-2' );" otus-database
```

MySQL контроллер

Посмотри содержимое таблицы:

```
kubectl exec -it $MYSQLPOD -- mysql -potuspassword -e "select * from test;" otus-  
database
```

id	name	
1	some data	
2	some data-2	

MySQL контроллер

Удалим mysql-instance:

```
kubectl delete mysqls.otus.homework mysql-instance
```

Теперь `kubectl get pv` показывает, что PV для mysql больше нет, а `kubectl get jobs.batch` показывает:

NAME	COMPLETIONS	DURATION	AGE
backup-mysql-instance-job	1/1	2s	2m39s

Если Job не выполнилась или выполнилась с ошибкой, то ее нужно удалять в ручную, т к иногда полезно посмотреть логи

Создадим заново mysql-instance:

```
kubectl apply -f deploy/cr.yml
```

Немного подождем и:

```
export MYSQLPOD=$(kubectl get pods -l app=mysql-instance -o jsonpath="{.items[*].metadata.name}")
kubectl exec -it $MYSQLPOD -- mysql -potuspassword -e "select * from test;" otus-
database
```

Должны увидеть:

id	name
1	some data
2	some data-2


MySQL контроллер

Мы убедились, что наш контроллер работает, теперь нужно его остановить и собрать Docker образ с ним. В директории build создайте Dockerfile:

```
FROM python:3.7
COPY templates ./templates
COPY mysql-operator.py ./mysql-operator.py
RUN pip install kopf kubernetes pyyaml jinja2
CMD kopf run /mysql-operator.py
```

Соберите и сделайте push в dockerhub ваш образ с оператором.

Деплой оператора

- Создайте в папке kubernetes-operator/deploy:
 - service-account.yml
 - role.yml
 - role-binding.yml
 - deploy-operator.yml
- Если вы делали задачи со  , то поменяйте используемый в deploy-operator.yml образ.

Деплой оператора

- Примените манифесты:
 - service-account.yml
 - role.yml
 - role-binding.yml
 - deploy-operator.yml

Проверим, что все работает

Создаем CR (если еще не создан):

```
kubectl apply -f deploy/cr.yml
```

Проверим, что все работает

Проверяем что появились pvc:

```
kubectl get pvc
```

NAME	STATUS	VOLUME	
CAPACITY	ACCESS MODES	STORAGECLASS	AGE
backup-mysql-instance-pvc	Bound	pvc-22eace9a-89e6-4926-8949-cc62cb6489af	1Gi
RWO	standard	35s	
mysql-instance-pvc	Bound	pvc-b7d25705-15d7-49a5-97cb-aeccd938e611	1Gi
RWO	standard	35s	

Проверим, что все работает

Заполним базу созданного mysql-instance:

```
export MYSQLPOD=$(kubectl get pods -l app=mysql-instance -o jsonpath="{.items[*].metadata.name}")
kubectl exec -it $MYSQLPOD -- mysql -u root -potuspassword -e "CREATE TABLE test (
id smallint unsigned not null auto_increment, name varchar(20) not null, constraint
pk_example primary key (id) );" otus-database

kubectl exec -it $MYSQLPOD -- mysql -potuspassword -e "INSERT INTO test ( id, name
) VALUES ( null, 'some data' );" otus-database

kubectl exec -it $MYSQLPOD -- mysql -potuspassword -e "INSERT INTO test ( id, name )
VALUES ( null, 'some data-2' );" otus-database
```

Проверим, что все работает

Посмотри содержимое таблицы:

```
kubectl exec -it $MYSQLPOD -- mysql -potuspassword -e "select * from test;" otus-  
database
```

```
+----+-----+  
| id | name      |  
+----+-----+  
|  1 | some data |  
|  2 | some data-2 |  
+----+-----+
```

Проверим, что все работает

Удалим mysql-instance:

```
kubectl delete mysqls.otus.homework mysql-instance
```

Теперь `kubectl get pv` показывает, что PV для mysql больше нет, а `kubectl get jobs.batch` показывает:

NAME	COMPLETIONS	DURATION	AGE
backup-mysql-instance-job	1/1	2s	2m39s

Если Job не выполнилась или выполнилась с ошибкой, то ее нужно удалять в ручную, т к иногда полезно посмотреть логи

Проверим, что все работает

Создадим заново mysql-instance:

```
kubectl apply -f deploy/cr.yml
```

Немного подождем и:

```
export MYSQLPOD=$(kubectl get pods -l app=mysql-instance -o jsonpath="{.items[*].metadata.name}")
kubectl exec -it $MYSQLPOD -- mysql -potuspassword -e "select * from test;" otus-
database
```

Должны увидеть:

```
+-----+-----+
| id | name      |
+-----+-----+
|  1 | some data |
|  2 | some data-2 |
+-----+-----+
```

Проверка | tree



Содержимое папки kubernetes-operators, если вы не делали задачи с :

```
└─ deploy
    ├── cr.yml
    ├── crd.yml
    ├── deploy-operator.yml
    ├── role-binding.yml
    ├── role.yml
    └─ service-account.yml
```

Содержимое папки kubernetes-operators, если вы делали задачи с :

```
|— build
|   |— Dockerfile
|   |— mysql-operator.py
|   └─ templates
|       |— backup-job.yml.j2
|       |— backup-pv.yml.j2
|       |— backup-pvc.yml.j2
|       |— mysql-deployment.yml.j2
|       |— mysql-pv.yml.j2
|       |— mysql-pvc.yml.j2
|       |— mysql-service.yml.j2
|       └─ restore-job.yml.j2
└─ deploy
    |— cr.yml
    |— crd.yml
    |— deploy-operator.yml
    |— role-binding.yml
    |— role.yml
    └─ service-account.yml
```


Проверка

- Сделайте PR в ветку kubernetes-operators
- Добавьте label с номером домашнего задания
- Добавьте label с , если выполнили задания со 
- Добавьте в README вывод команды `kubectl get jobs` (там должны быть успешно выполненные backup и restore job)
- Приложение вывод при запуске MySQL:

```
export MYSQLPOD=$(kubectl get pods -l app=mysql-instance -o jsonpath="{.items[*].metadata.name}")
kubectl exec -it $MYSQLPOD -- mysql -potuspassword -e "select * from test;" otus-
database
```

Задание со (1)

- Исправить контроллер, чтобы он писал в `status subresource`
- Описать изменения в README.md (показать код, объяснить, что он делает)
- В README показать, что в status происходит запись
- Например, при успешном создании mysql-instance, `kubectl describe mysqls.otus.homework mysql-instance` может показывать:

Status:

Kopf:

mysql_on_create:

Message: mysql-instance created without restore-job

Задание со (2)

- Добавить в контроллер логику обработки изменений CR:
 - Например, реализовать смену пароля от MySQL, при изменении этого параметра в описании mysql-instance
- В README:
 - Показать, что код работает
 - Объяснить, что он делает