Операторы, CustomResourceDefinition

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 небязательные (в заголовке слайда отражены знаком 🐍).

Подготовка

- Запустите kubernetes кластер в `minikube` версии 1.21 и выше
- Создадим директорию `kubernetes-operators/deploy`:

```
1  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:

```
1   apiVersion: otus.homework/v1
2   kind: MySQL
3   metadata:
4     name: mysql-instance
5   spec:
6     image: mysql:5.7
7     database: otus-database
8     password: otuspassword  # Так делать не нужно, следует использовать secret
9     storage_size: 1Gi
10  usless_data: "useless info"
```

> gist

CustomResource

Пробуем применить его:

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

Видим ошибку:

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

Ошибка связана с отсутсвием 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:

```yaml
apiVersion: apiextensions.k8s.io/v1beta1
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:

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

Добавим в спецификацию CRD (`spec`) параметры `validation` gist

Validation

```yaml
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

```
1   kubectl apply -f deploy/crd.yml
2   kubectl apply -f deploy/cr.yml
```

```
1  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:

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

Применяем:

```
1  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 обязательный 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 завершенные 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

```
1    import kopf
2    import yaml
3    import kubernetes
4    import time
5    from jinja2 import Environment, FileSystemLoader
```

🐍 MySQL контроллер

- В дирректории 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:

```
1    def render_template(filename, vars_dict):
2        env = Environment(loader=FileSystemLoader('./templates'))
3        template = env.get_template(filename)
4        yaml_manifest = template.render(vars_dict)
5        json_manifest = yaml.load(yaml_manifest)
6        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:

```
1       # Генерируем JSON манифесты для деплоя
2       persistent_volume = render_template('mysql-pv.yml.j2',
3                                           {'name': name,
4                                            'storage_size': storage_size})
5       persistent_volume_claim = render_template('mysql-pvc.yml.j2',
6                                                 {'name': name,
7                                                  'storage_size': storage_size})
8       service = render_template('mysql-service.yml.j2', {'name': name})
9
10      deployment = render_template('mysql-deployment.yml.j2', {
11          'name': name,
12          'image': image,
13          'password': password,
14          '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 конфигурацие 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:

```
1  kubectl delete mysqls.otus.homework mysql-instance
2  kubectl delete deployments.apps mysql-instance
3  kubectl delete pvc mysql-instance-pvc
4  kubectl delete pv mysql-instance-pv
5  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:

```
1       # Определяем, что созданные ресурсы являются дочерними к управляемому CustomResource:
2       kopf.append_owner_reference(persistent_volume, owner=body)
3       kopf.append_owner_reference(persistent_volume_claim, owner=body)  # addopt
4       kopf.append_owner_reference(service, owner=body)
5       kopf.append_owner_reference(deployment, owner=body)
6       # ^ 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:

```
1   @kopf.on.delete('otus.homework', 'v1', 'mysqls')
2   def delete_object_make_backup(body, **kwargs):
3       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:

```
1   def delete_success_jobs(mysql_instance_name):
2       api = kubernetes.client.BatchV1Api()
3       jobs = api.list_namespaced_job('default')
4       for job in jobs.items:
5           jobname = job.metadata.name
6           if (jobname == f"backup-{mysql_instance_name}-job"):
7               if job.status.succeeded == 1:
8                   api.delete_namespaced_job(jobname,
9                                             'default',
10                                            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:

```
1   def wait_until_job_end(jobname):
2       api = kubernetes.client.BatchV1Api()
3       job_finished = False
4       jobs = api.list_namespaced_job('default')
5       while (not job_finished) and \
6               any(job.metadata.name == jobname for job in jobs.items):
7           time.sleep(1)
8           jobs = api.list_namespaced_job('default')
9           for job in jobs.items:
10              if job.metadata.name == jobname:
11                  if job.status.succeeded == 1:
12                      job_finished = True
13
```

MySQL контроллер

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")
```

> Акутальное состояние контроллера gist

🐍 MySQL контроллер

Добавим генерацию json из шаблона для restore-job

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

Добавим попытку восстановиться из бэкапов после deployment mysql:

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

🐍 MySQL контроллер

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

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

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

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

Создаем CR:

```
1  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

MySQL контроллер

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:

```
1  FROM python:3.7
2  COPY templates ./templates
3  COPY mysql-operator.py ./mysql-operator.py
4  RUN pip install kopf kubernetes pyyaml jinja2
5  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):

```
1  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
```

Проверка | tree 🐍

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 комманды `kubectl get jobs` (там должны быть успешно выполненные backup и restore job)
- Приложетие вывод при запущенном MySQL:

```
1    export MYSQLPOD=$(kubectl get pods -l app=mysql-instance -o jsonpath="{.items[*].metadata.name}")
2    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:

```
1    Status:
2      Kopf:
3      mysql_on_create:
4        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