Настройка интеграции Gitlab-CI и Kubernetes

Установка gitlab-runner

Установка репозитория и пакета из него:

```
curl -L https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.rpm.sh | sudo
 bash
yum -y install gitlab-runner
```

Регистрация проекта:

```
gitlab-ci-multi-runner register
```

Развертывание приложений в кластер Kubernetes

Конвеер *(pipeline)* состоит из пяти шагов (stage):

1. Сборка образа docker из Dockerfile, лежащего в корне проекта в Git

```
build:
  stage: build
  script:
    - docker build -t $CI_REGISTRY/$CI_PROJECT_NAMESPACE/$CI_PROJECT_NAME:$CI_COMMIT_REF_SLUG.
$CI_PIPELINE_ID .
```

2. Тесты

```
test:
  stage: test
  variables:
    GIT_STRATEGY: none
  script:
    - docker-compose up --abort-on-container-exit --exit-code-from app --quiet-pull
```

3. Очистка окружения после тестов

```
cleanup:
  stage: cleanup
  variables:
    GIT_STRATEGY: none
  script:
    - docker-compose down
  when: always
```

4. Загрузка (push) в *docker registry*

```
push:
  stage: push
  variables:
```

```
    GIT_STRATEGY: none
  before_script:
    - docker login -u gitlab-ci-token -p $CI_BUILD_TOKEN $CI_REGISTRY
  script:
    - docker push $CI_REGISTRY/$CI_PROJECT_NAMESPACE/$CI_PROJECT_NAME:$CI_COMMIT_REF_SLUG.$CI_
PIPELINE_ID
  only:
    - master
```

5. Развертывание (deploy) в рабочее окружение (production)

```
deploy_prod:
  stage: deploy
  environment:
    name: production
    url: https://exmple.slurm.io
  script:
    - docker run
      --rm
      -v $PWD/.helm:/.helm
      -e "K8S_API_URL=$K8S_API_URL"
      -e "K8S_CI_TOKEN=$K8S_CI_TOKEN_PROD"
      -e "CI_PROJECT_PATH_SLUG=$CI_PROJECT_PATH_SLUG"
      -e "CI_ENVIRONMENT_NAME=$CI_ENVIRONMENT_NAME"
      -e "CI_REGISTRY=$CI_REGISTRY"
      -e "CI_PROJECT_NAMESPACE=$CI_PROJECT_NAMESPACE"
      -e "CI_PROJECT_NAME=$CI_PROJECT_NAME"
      -e "CI_COMMIT_REF_SLUG=$CI_COMMIT_REF_SLUG"
      -e "CI_PIPELINE_ID=$CI_PIPELINE_ID"
      centosadmin/kubernetes-helm:v2.9
      /bin/sh -c
      'kubectl config set-cluster k8s --insecure-skip-tls-verify=true --server="$K8S_API_URL"
&&
      kubectl config set-credentials ci --token="$K8S_CI_TOKEN" &&
      kubectl config set-context ci --cluster=k8s --user=ci &&
      kubectl config use-context ci &&
      helm init --client-only &&
      helm upgrade --install "$CI_PROJECT_PATH_SLUG" .helm
        --set image="$CI_REGISTRY/$CI_PROJECT_NAMESPACE/$CI_PROJECT_NAME"
        --set imageTag="$CI_COMMIT_REF_SLUG.$CI_PIPELINE_ID"
        -f .helm/values.prod.yaml
        --wait
        --timeout 240
        --debug
        --namespace "$CI_PROJECT_PATH_SLUG-$CI_ENVIRONMENT_NAME"
        --tiller-namespace="$CI_PROJECT_PATH_SLUG-$CI_ENVIRONMENT_NAME" ||
      (helm history --max 2 --tiller-namespace="$CI_PROJECT_PATH_SLUG-$CI_ENVIRONMENT_NAME" "$
CI_PROJECT_PATH_SLUG" | head -n 2 | tail -n 1 | awk "{print \$1}" | xargs helm rollback --till
er-namespace="$CI_PROJECT_PATH_SLUG-$CI_ENVIRONMENT_NAME" "$CI_PROJECT_PATH_SLUG" && exit 1)'
  only:
    - master
```

Описание шагов

1. build

На данном этапе выполняется единственная команда (**docker build -t <IMAGE_NAME>**)
Само <IMAGE_NAME> составляется из встроенных в Gitlab переменных:

- **$CI_REGISTRY** - адрес Gitlab Docker Registry для данного проекта
- **$CI_PROJECT_NAMESPACE** - имя группы проекта
- **$CI_PROJECT_NAME** - имя самого проекта

- **$CI_COMMIT_REF_SLUG** - имя ветки или тэга, написанное маленькими буквами, сокращенное до 63 байт, в котором все символы, кроме 0-9 и a-z, заменены на **-**
- **$CI_PIPELINE_ID** - уникальный ID текущего pipeline

На выходе мы получим, например, такое имя образа (image): **registry.slurm.io/slurm.io/example:master.45286**

2. test

На данном шаге мы поднимаем тестовое окружение в **docker-compose** и с помощью ключей **--abort-on-container-exit --exit-code-from app** указываем ему, что мы хотим следить только за кодом завершения контейнера с именем app, и в соответствии с этим кодом завершения оценивать успех pipeline.

Содержимое файла **docker-compose.yml** может быть таким:

```
version: '2.1'
services:
  app:
    image: ${CI_REGISTRY}/${CI_PROJECT_NAMESPACE}/${CI_PROJECT_NAME}:${CI_COMMIT_REF_SLUG}.${CI_PI
PELINE_ID}
    environment:
      DB_HOST: db
      DB_PORT: 5432
      DB_USER: postgres
      DB_PASSWORD: postgres
      DB_NAME: test
      DB_WAIT_TIMEOUT: 60
      SELENIUM_HOST: selenium
      SELENIUM_PORT: 4444
      RAILS_ENV: test
      RAILS_LOG_TO_STDOUT: 1
    command: /bin/bash -c 'bundle exec rake db:migrate && bundle exec rspec spec'
    depends_on:
      db:
        condition: service_healthy

  db:
    image: postgres:9.6
    environment:
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: test
    healthcheck:
      test: ["CMD", "pg_isready", "-U", "postgres"]
      interval: 1s
      timeout: 1s
      retries: 60
    logging:
      driver: none

  selenium:
    image: selenium/standalone-chrome-debug
    logging:
      driver: none
```

Пояснения:

- имя образа для разворачивания контейнера с приложением составляется из тех же переменных, что и при сборке;
- для ожидания запуска базы данных перед запуском самого приложения используется конструкция

```
    depends_on:
      db:
        condition: service_healthy
```

, а в описании БД есть healthcheck

```
    healthcheck:
      test: ["CMD", "pg_isready", "-U", "postgres"]
      interval: 1s
      timeout: 1s
      retries: 60
```

Таким образом, приложение не будет запущено, пока не поднимется инстанс БД.

- отключен вывод логов у БД и Selenium, для того, чтобы они не засоряли вывод CI. Это сделано с помощью конструкции

```
    logging:
      driver: none
```

В остальном это стандартный compose-файл.

3. cleanup

На этом шаге запускается команда, которая после завершения тестов гасит окружение docker-compose. Для того, чтобы данный этап выполнялся всегда (даже если предыдущий завершился с ошибкой), используется следующая конструкция в файле gitlab-ci

```
when: always
```

4. push

Здесь мы сначала логинимся в docker registry гитлаба для данного проекта:

```
    - docker login -u gitlab-ci-token -p $CI_BUILD_TOKEN $CI_REGISTRY
```

, при этом все используемые переменные стандартные (встроенные). Не требуется дополнительно создавать ключи доступа и выдавать CI права на пуш.

Использование секретов Kubernetes для приложений

Пароли и прочую sensitive data, которые нельзя хранить в репозитории, держим в *секретах Kubernetes*. Их значения будут переданы контейнеру с приложением в переменных окружения с помощью вот такой конструкции в описании контейнера (в файле **.helm/templates/deploy.yml**):

```
- name: LOGIN_PASSWORD
          valueFrom:
            secretKeyRef:
              key: login-password
              name: slurmio
```

, т. е. "секрете" **slurmio** мы ищем значение ключа **login-password**.

Создается secret таким образом:

```
kubectl create secret generic slurmio \
    --from-literal secret-key-base='XXX' \
    --from-literal db-password='XXX' \
    --from-literal db-user='XXX' \
    --from-literal login-password='xxx' \
    --from-literal login-sms='xxx' \
    --namespace slurm-io-production
```

После создания можно проверить, что значение хранится верно и туда не попали лишние символы -- например, кавычки:

```
kubectl get secret --namespace=slurm-io-production slurmio -o jsonpath='{.data.db-password}' | bas
e64 -d
```

5. deploy

Доступ к kube-API через ingress

Если gitlab-runner запущен вне локальной сети кластера, может потребоваться открыть доступ к kube-API с помощью ingress. В этом ингрессе можно разрешить доступ только с IP гитлаб раннера:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  annotations:
    nginx.ingress.kubernetes.io/proxy-send-timeout: "300"
    nginx.ingress.kubernetes.io/proxy-read-timeout: "300"
    nginx.ingress.kubernetes.io/whitelist-source-range: 1.1.1.1/32,2.2.2.2/32
    nginx.ingress.kubernetes.io/secure-backends: "true"
  labels:
    app: kube-api
  name: kube-api
spec:
  rules:
  - host: k8s.slurm.io
    http:
      paths:
      - backend:
          serviceName: kubernetes
          servicePort: 443
        path: /
  tls:
  - hosts:
    - k8s.slurm.io
    secretName: k8s-slurm-io-tls
```

Кроме того, в случае, если мы выписываем сертификаты от LE через cert-manager, надо будет создать сертификат вручную, чтобы cert-manager создавал для него свой отдельный ингресс; по умолчанию он добавляет локейшен в ингресс выше и, так как там стоит ограничение по адресам, проверка домена не проходит

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Certificate
metadata:
  annotations:
  name: k8s-slurm-io-tls
spec:
  acme:
    config:
    - domains:
      - k8s.slurm.io
      http01:
        ingress: ""
        ingressClass: nginx
  commonName: k8s.slurm.io
  dnsNames:
  - k8s.slurm.io
  issuerRef:
    kind: ClusterIssuer
    name: letsencrypt
  secretName: k8s-slurm-io-tls
```

Доступ k8s к registry. Namespace и secret.

Нам следует обеспечить доступ kubernetes к docker-registry Гитлаба (для того, чтобы он мог скачивать оттуда образы). Для этого требуется создать в Gitlab пароль доступа к registry, а в kubernetes -- пространство имён (namespace) с именем **$CI_PROJECT_PATH_SLUG-$CI_ENVIRONMENT_NAME** (в случае со slurm.io это **"slurm-io-production"**), и в этом namespace нужно создать секреты с паролями.

Пароль для доступа к registry создается в Gitlab, как API токен с правами **read-registry** (Deploy Tokens): https://gitlab.slurm.io/slurm.io/example/settings/repository. На выходе получаем username / password, которые подставляем в **--docker-username** и **--docker-password** в следующем пункте.

Создадим секрет с паролями доступа к registry (подставляем на месте переменных их значения):

```
kubectl create secret docker-registry <$CI_PROJECT_PATH_SLUG>-gitlab-registry \
    --docker-username 'USER' \
    --docker-password 'PASSWORD' \
    --docker-server <$CI_REGISTRY> \
    --docker-email 'admin@slurm.io'
    --namespace <$CI_PROJECT_PATH_SLUG-$CI_ENVIRONMENT_NAME>
```

Указываем созданный imagePullSecret в описании деплоймента:

```
      imagePullSecrets:
        - name: <$CI_PROJECT_PATH_SLUG>-gitlab-registry
```

Затем создаем пользователя в Кубе для CI:

Можно воспользоваться вот таким скриптом:
https://github.com/centosadmin/slurm/blob/master/practice/ci-cd/setup.sh

- В нэймспэйсе например users создаем serviceaccount (заменяем переменные из значениями!)

```
kubectl create serviceaccount --namespace <$CI_PROJECT_PATH_SLUG-$CI_ENVIRONMENT_NAME> <$CI_PROJEC
T_PATH_SLUG-$CI_ENVIRONMENT_NAME>
```

- Создаем роль

```
    cat << EOF | kubectl create --namespace <$CI_PROJECT_PATH_SLUG-$CI_ENVIRONMENT_NAME> -f -
        apiVersion: rbac.authorization.k8s.io/v1
        kind: Role
        metadata:
          name: <$CI_PROJECT_PATH_SLUG-$CI_ENVIRONMENT_NAME>
        rules:
        - apiGroups: ["", "extensions", "apps", "certmanager.k8s.io"]
          resources: ["*"]
          verbs: ["*"]
EOF
```

- После этого в нэймспэйсе нашего проекта создаем rolebinding (не забываем о переменных!)

```
kubectl create rolebinding --namespace <$CI_PROJECT_PATH_SLUG-$CI_ENVIRONMENT_NAME> \
--serviceaccount <$CI_PROJECT_PATH_SLUG-$CI_ENVIRONMENT_NAME>:<$CI_PROJECT_PATH_SLUG-$CI_ENVIRONME
NT_NAME> \
--role <$CI_PROJECT_PATH_SLUG-$CI_ENVIRONMENT_NAME> \
 <$CI_PROJECT_PATH_SLUG-$CI_ENVIRONMENT_NAME>
```

- Далее получаем токен от нашего serviceaccount

```
kubectl get secret --namespace <$CI_PROJECT_PATH_SLUG-$CI_ENVIRONMENT_NAME> \
 <$CI_PROJECT_PATH_SLUG-$CI_ENVIRONMENT_NAME>-<TAB> -o jsonpath='{.data.token}' | base64 -d ; echo
```

- Копируем и создаем в проекте в Gitlab переменную K8S_CI_TOKEN со значением этого токена.

Далее непосредственно переходим к описанию процесса деплоя из CI.

Для осуществления самого деплоя используется команда helm upgrade.

Мы запускаем образ centosadmin/kubernetes-helm:v2.9, в котором уже установлены kubectl и helm.
В нем выполняем стандартную настройку доступа к Kube-API.

```
      kubectl config set-cluster k8s --insecure-skip-tls-verify=true --server=$K8S_API_URL &&
      kubectl config set-credentials ci --token=$K8S_CI_TOKEN &&
      kubectl config set-context ci --cluster=k8s --user=ci &&
      kubectl config use-context ci
```

И далее запускаем обновление приложения

```
      helm upgrade --install $CI_PROJECT_PATH_SLUG .helm
        --set image=$CI_REGISTRY/$CI_PROJECT_NAMESPACE/$CI_PROJECT_NAME
        --set imageTag=$CI_COMMIT_REF_SLUG.$CI_PIPELINE_ID
        --wait
        --timeout 180
        --debug
        --namespace $CI_PROJECT_PATH_SLUG-$CI_ENVIRONMENT_NAME
        --tiller-namespace=$CI_PROJECT_PATH_SLUG-$CI_ENVIRONMENT_NAME
```

Тут мы подменяем переменные image и imageTag на сбилженный в этом пайплайне имадж.
И ждем (--wait) 180 секунд (--timeout) пока в кубернетисе не запустятся все объекты (поды станут ready, сервисы создадутся и т.д.)
--tiller-namespace нужен так как мы хотим обращаться к конкретному Tiller, который мы установили в нэймспэйсе приложения.

В случае если в течении timeout приложения так и не смогли запуститься, helm upgrade возвращает ненулевой код завершения и срабатывает команда отката.

```
|| (helm history --max 2 --tiller-namespace=$CI_PROJECT_PATH_SLUG-$CI_ENVIRONMENT_NAME $CI_PROJECT
_PATH_SLUG | head -n 2 | tail -n 1 | awk "{print \$1}" | xargs helm rollback --tiller-namespace=$C
I_PROJECT_PATH_SLUG-$CI_ENVIRONMENT_NAME $CI_PROJECT_PATH_SLUG && exit 1)
```

Она состоит из двух частей, так как сначала нам нужно получить версию предыдущей ревизии, а затем откатить деплой на нее.

Helm чарт

Сам чарт для приложения находится в директории .helm репозитория проекта.
В нем содержатся темплэйты для создания деплоймента, деплоймента sidekiq, сервиса, сертификата и ингрессов (один в www, второй без www, для осуществления редиректа www - > no www)

Поддерживаемые параметры для конфигурации:

```
image: адрес реджистри (будет в любом случае переопределена в CI)
imageTag: тэг имаджа (так же будет переопределен в CI)
imagePullSecret: данные для пула имаджей (нужно создать вручную в кластере перед деплоем)

env:
  NAME: value   любое количество переменных с их значениями по одному на строчку (будут добавлены
в деплоймент)

# This variables is taken from secret
# Value is secret name where variable value can be found
# Key in secret equals lowercased variable name with "_" replaced by "-"
# The secret should be created manually
envSecret:
  NAME: secret-name    см. описание выше

# Resources for app. Limits is the maximum number of resources that app cas use.
# And requests is resources that will be granted to the app at start time.
app:
  replicas: 2

  resources:
    limits:
      cpu: 200m
      memory: 256Mi
    requests:
      cpu: 200m
      memory: 256Mi
sidekiq:
  replicas: 2

  resources:
    limits:
      cpu: 200m
      memory: 256Mi
    requests:
      cpu: 200m
      memory: 256Mi

service:
  port: 80    порт на котором слушает приложение (будет добавлен в деплоймент, сервис и ингресс)

ingress:
  host: domain.com   хост приложения (будут созданы ингрессы domain.com www.domain.com и на эти им
ена будет получен LE сертификат)
```