Домашнее задание. Шаблонизация манифестов Kubernetes

Intro

В неофициальной домашней работе по занятию **"Компоненты Kubernetes"** вам было предложено зарегистрировать триальный аккаунт в GCP.

Если этого еще не сделано - сейчас наступило самое подходящее время. Домашнее задание предполагает выполнение в GKE кластере.

Обзор и развернутые инструкции по созданию кластера можно найти в официальной документации

Intro

Перед началом работы над домашним заданием вам необходимо:

1. Любым удобным способом (через **web console**, через **gcloud**, с использованием **terraform**) создать **managed** kubernetes кластер в облаке GCP
2. Настроить kubectl на локальной машине:

```
gcloud beta container clusters get-credentials ...
```

> ! В данной домашней работе конфигурация кластера не имеет принципиального значения, можно использовать параметры по умолчанию

Устанавливаем готовые Helm charts

Попробуем установить Helm charts созданные сообществом. С их помощью создадим и настроим инфраструктурные сервисы, необходимые для работы нашего кластера.

Будем использовать разные способы установки:

- **Helm 2** и **tiller** с правами `cluster-admin`
- **Helm 2** и **tiller** с правами, ограниченными namespace
- **Helm 2** с плагином **helm-tiller** (позволяет отказаться от использования **tiller** внутри кластера)
- **Helm 3**

Устанавливаем готовые Helm charts

Сегодня будем работать со следующими сервисами:

- nginx-ingress - сервис, обеспечивающий доступ к публичным ресурсам кластера
- cert-manager - сервис, позволяющий динамически генерировать Let's Encrypt сертификаты для ingress ресурсов
- chartmuseum - специализированный репозиторий для хранения helm charts
- harbor - хранилище артефактов общего назначения (Docker Registry), поддерживающее helm charts

Подготовка

Для начала нам необходимо установить клиент **Helm 2** на локальную машину.

Инструкции по установке можно найти по ссылке

Критерий успешности установки - после выполнения команды:

```
helm version --client
```

Ожидается следующий вывод:

```
Client: &version.Version{SemVer:"v2.14.3",
GitCommit:"0e7f3b6637f7af8fcfddb3d2941fcc7cbebb0085", GitTreeState:"clean"}
```

Подготовка

Создадим сервисный аккаунт `tiller` для корректной работы **tiller** в кластере. Примените манифест:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: tiller
  namespace: kube-system
```

Подготовка

Присвоим созданному аккаунту роль `cluster-admin`. Примените манифест:

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: tiller
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
  - kind: ServiceAccount
    name: tiller
    namespace: kube-system
```

! Давать сервисному аккаунту `tiller` права `cluster-admin` - плохая практика

Подготовка

Установим **tiller** в Kubernetes кластер и сконфигурируем локальное окружение с использованием команды helm init

```
helm init --service-account=tiller
```

Критерий успешности установки - после выполнения команды:

```
helm version
```

Ожидается следующий вывод:

```
Client: &version.Version{SemVer:"v2.14.3",
GitCommit:"0e7f3b6637f7af8fcfddb3d2941fcc7cbebb0085", GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.14.3",
GitCommit:"0e7f3b6637f7af8fcfddb3d2941fcc7cbebb0085", GitTreeState:"clean"}
```

Памятка по использованию Helm

Создание **release**:

```
$ helm install <chart_name> --name=<release_name>
$ kubectl get configmaps -n kube-system | grep <release_name>
<release_name>.v1                    1      26s
```

Обновление **release**:

```
$ helm upgrade <release_name> <chart_name>
$ kubectl get configmaps -n kube-system | grep <release_name>
<release_name>.v1                    1      64s
<release_name>.v2                    1      28s
```

Создание или обновление **release**:

```
$ helm upgrade --install <release_name> <chart_name>
$ kubectl get configmaps -n kube-system | grep <release_name>
<release_name>.v1                    1      64s
<release_name>.v2                    1      28s
<release_name>.v3                    1      5s
```

nginx-ingress

Создадим **release** nginx-ingress используя первый способ установки - **Helm 2** и **tiller** с правами `cluster-admin`

```
helm upgrade --install nginx-ingress stable/nginx-ingress --wait \
  --namespace=nginx-ingress \
  --version=1.11.1
```

Разберем используемые ключи:

- **--wait** - ожидать успешного окончания установки (подробности)
- **--timeout** - считать установку неуспешной по истечении указанного времени
- **--namespace** - установить chart в определенный namespace (будет создан, если не существует)
- **--version** - установить определенную версию chart

cert-manager

Следующим установим в кластер cert-manager. Будем использовать tiller, который установлен внутри namespace cert-manager и имеет соответствующие ограничения привилегий.

- Создайте namespace `cert-manager`
- Создайте сервисный аккаунт `tiller-cert-manager` в namespace `cert-manager`
- Создайте роль из следующего манифеста:

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: tiller-cert-manager
  namespace: cert-manager
rules:
- apiGroups: ["", "batch", "extensions", "apps"]
  resources: ["*"]
  verbs: ["*"]
```

cert-manager

Примените следующий манифест **roleBinding**:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: tiller-cert-manager
  namespace: cert-manager
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: tiller-cert-manager
subjects:
  - kind: ServiceAccount
    name: tiller-cert-manager
    namespace: cert-manager
```

Инициализируйте helm в namespace `cert-manager`:

```
helm init --tiller-namespace cert-manager --service-account tiller-cert-manager
```

cert-manager

Добавим репозиторий, в котором хранится актуальный helm chart cert-manager:

```
helm repo add jetstack https://charts.jetstack.io
```

Также для установки cert-manager предварительно потребуется создать в кластере некоторые **CRD** (ссылка на документацию по установке):

```
kubectl apply -f https://raw.githubusercontent.com/jetstack/cert-manager/release-
0.9/deploy/manifests/00-crds.yaml
```

Еще одна подготовка, описанная в документации:

```
kubectl label namespace cert-manager certmanager.k8s.io/disable-validation="true"
```

cert-manager

Теперь можно попробовать установить cert-manager. Для начала проверим, что tiller в namespace `cert-manager` действительно не обладает правами на управление объектами в других namespace (например, в `nginx-ingress`):

```
helm upgrade --install cert-manager jetstack/cert-manager --wait \
  --namespace=nginx-ingress \
  --version=0.9.0 \
  --tiller-namespace cert-manager
```

И получим примерно такую ошибку, говорящую об отсутствии прав:

```
Error: release cert-manager failed: namespaces "nginx-ingress" is forbidden: User
"system:serviceaccount:cert-manager:tiller-cert-manager" cannot get resource
"namespaces" in API group "" in the namespace "nginx-ingress"
```

cert-manager

Установим cert-manager в корректный namespace:

```
helm upgrade --install cert-manager jetstack/cert-manager --wait \
  --namespace=cert-manager \
  --version=0.9.0 \
  --tiller-namespace cert-manager
```

И столкнемся с одной из первых проблем helm. Это не баг:

```
WARNING: Namespace "cert-manager" doesn't match with previous. Release will be deployed
to nginx-ingress
```

cert-manager

А это - баг:

```
UPGRADE FAILED
Error: "cert-manager" has no deployed releases
Error: UPGRADE FAILED: "cert-manager" has no deployed releases
```

Одно из описаний проблемы

Решение для helm 2 (помимо удаления неудачного релиза)

cert-manager

Удалим release со статусом **FAILED**:

```
helm delete --purge cert-manager \
  --tiller-namespace cert-manager
```

И установим заново в правильный namespace, добавив ключ `--atomic` из упомянутого выше PR:

```
helm upgrade --install cert-manager jetstack/cert-manager --wait \
  --namespace=cert-manager \
  --version=0.9.0 \
  --tiller-namespace cert-manager \
  --atomic
```

Самостоятельное задание

- Новая ошибка должна подсказать вам, почему **в данном случае** установка tiller в namespace `cert-manager` и ограничение прав для него не имеет практического смысла
- Любым способом установите cert-manager в кластер
- Изучите документацию cert-manager, и определите, что еще требуется установить для корректной работы
- Поместите манифесты дополнительно созданных ресурсов в директорию `kubernetes-templating/cert-manager/`
- Проверить корректную работу cert-manager можно будет на последующих helm chart

chartmuseum

Попробуем обойтись без tiller внутри кластера. Будем использовать плагин helm-tiller, позволяющий запустить tiller локально

Установите плагин:

```
helm plugin install https://github.com/rimusz/helm-tiller
```

chartmuseum

Кастомизируем установку chartmuseum

- Создайте директорию `kubernetes-templating/chartmuseum/` и поместите туда файл `values.yaml`
- Изучите содержимое оригинального файла `values.yaml`
- Включите:
  - Создание ingress ресурса с корректным `hosts.name` (должен использоваться nginx-ingress)
  - Автоматическую генерацию Let's Encrypt сертификата

https://github.com/helm/charts/tree/master/stable/chartmuseum

chartmuseum

Файл `values.yaml` для chartmuseum будет выглядеть примерно следующим образом:

```
ingress:
  enabled: true
  annotations:
    kubernetes.io/ingress.class: nginx
    kubernetes.io/tls-acme: "true"
    certmanager.k8s.io/cluster-issuer: "letsencrypt-production"
    certmanager.k8s.io/acme-challenge-type: http01
  hosts:
    - name: chartmuseum.example.com
      path: /
      tls: true
      tlsSecret: chartmuseum.example.com
```

Вместо `example.com` укажите **EXTERNAL-IP** сервиса вашего nginx-ingress в формате `<IP-адрес.nip.io>`, например `1.1.1.1.nip.io`

chartmuseum

Установим chartmuseum:

```
helm tiller run \
  helm upgrade --install chartmuseum stable/chartmuseum --wait \
  --namespace=chartmuseum \
  --version=2.3.2 \
  -f kubernetes-templating/chartmuseum/values.yaml
```

Проверим, что tiller внутри кластера ничего не знает про release chartmuseum:

```
helm list
```

А локальный tiller знает:

```
helm tiller run helm list
```

chartmuseum

Дело в том, что плагин helm tiller по умолчанию хранит информацию о релизе в секретах (kubectl get secrets -n kube-system), а обычный tiller в configMap.

Можете попробовать переустановить chartmuseum указав helm tiller, что конфигурация должна храниться в configMap:

```
export HELM_TILLER_STORAGE=configmap
```

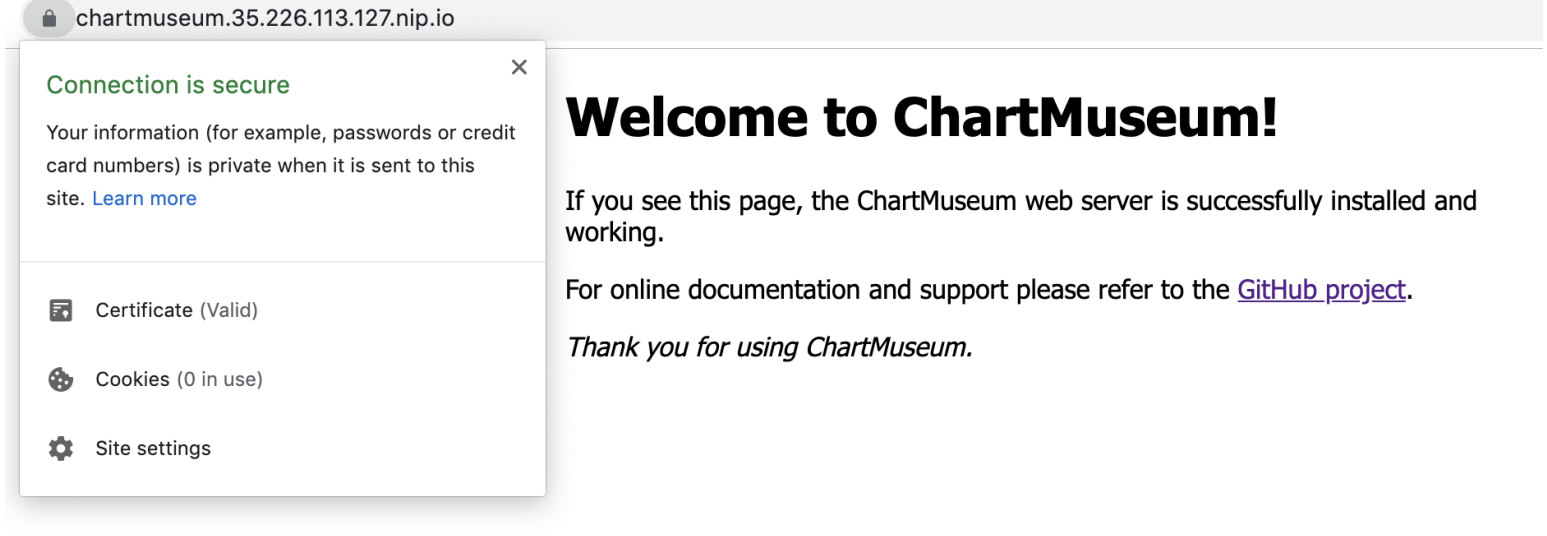
И убедиться, что tiller внутри кластера увидел release:

```
helm status chartmuseum
```

chartmuseum

Критерий успешности установки

- Chartmuseum доступен по URL `https://chartmuseum.<IP>.nip.io`
- Сертификат для данного URL валиден



chartmuseum | Задание со ⭐

- Научитесь работать с chartmuseum
- Опишите в PR последовательность действий, необходимых для добавления туда helm chart's и их установки с использованием chartmuseum как репозитория

harbor

Для установки harbor попробуем использовать **helm 3**

На текущий момент он находится в статусе beta, соответственно можно ожидать, что он заработает без существенных проблем

Скачайте последний доступный binary файл для вашей OS и поместите его в `$PATH` (рекомендуется назвать файл `helm3` чтобы не было пересечений с helm 2)

В итоге команда `helm3 version` должна показать примерно следующий вывод:

```
version.BuildInfo{Version:"v3.0.0-beta.2",
GitCommit:"26c7338408f8db593f93cd7c963ad56f67f662d4", GitTreeState:"clean", GoVersion:
"go1.12.9"}
```

harbor

Для наглядности можно удалить deployment с tiller из кластера:

```
kubectl delete deployment tiller-deploy -n kube-system
```

И проверить, что **helm 2** сломался:

```
helm list
```

А **helm 3** продолжает работать:

```
helm3 list
```

Самостоятельное задание

- Установите harbor в кластер с использованием **helm3**
- Используйте репозиторий https://github.com/goharbor/harbor-helm и CHART VERSION **1.1.2**
- Требования:
  - Должен быть включен ingress и настроен host `harbor.<IP-адрес>.nip.io`
  - Должен быть включен TLS и выписан валидный сертификат
- Скопируйте используемый файл `values.yaml` в директорию `kubernetes-templating/harbor/`

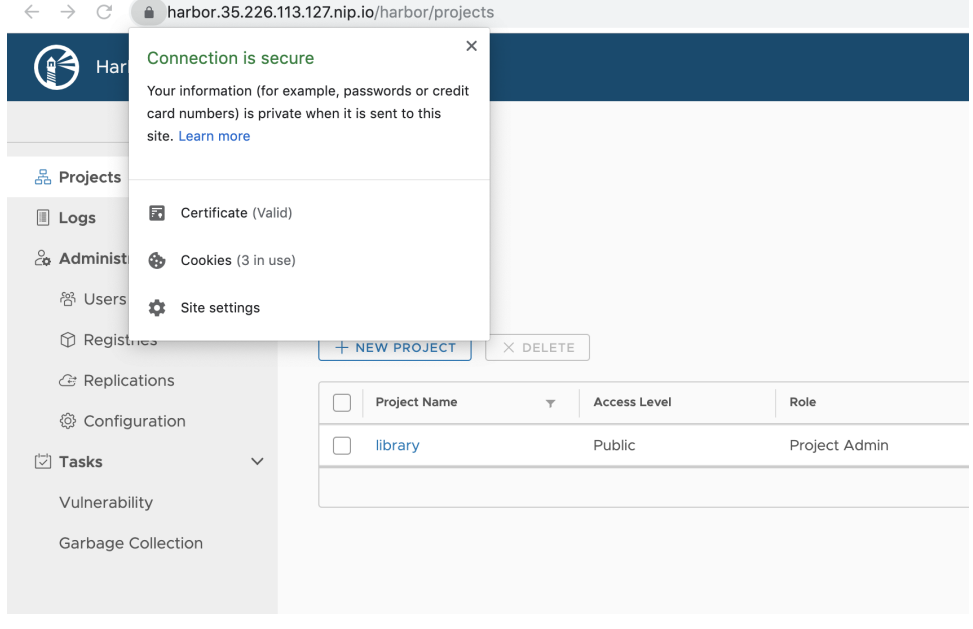
Tips & Tricks

- Формат описания переменных в файле `values.yaml` для **chartmuseum** и **harbor** отличается
- Helm3 не создает namespace в который будет установлен release
- Проще выключить сервис `notary`, он нам не понадобится
- Реквизиты по умолчанию - **admin/Harbor12345**
- `nip.io` может оказаться забанен в cert-manager. Если у вас есть собственный домен - лучше использовать его, либо попробовать `xip.io`, либо переключиться на staging ClusterIssuer

Обратите внимание, как helm3 хранит информацию о release:
`kubectl get secrets -n harbor -l owner=helm`

Критерий успешности установки

- Harbor запущен и работает
- Предъявленные требования выполняются



Используем helmfile | Задание со ⭐

Опишите установку **nginx-ingress**, **cert-manager** и **harbor** в `helmfile`

Приложите получившийся `helmfile.yaml` и другие файлы (при их наличии) в директорию `kubernetes-templating/helmfile`

Создаем свой helm chart

Создаем свой helm chart

Типичная жизненная ситуация:

- У вас есть приложение, которое готово к запуску в Kubernetes
- У вас есть для манифесты этого приложения, но вам надо запускать его на разных окружениях с разными параметрами

Возможные варианты решения:

- Написать разные манифесты для разных окружений
- Использовать "костыли" - sed, envsubst, etc...
- Использовать полноценное решение для шаблонизации (helm, etc...)

Создаем свой helm chart

Мы рассмотрим третий вариант. Возьмем готовые манифесты и подготовим их к релизу на разные окружения.

Использовать будем демо-приложение socks-shop от Weaveworks, представляющее собой типичный набор микросервисов.

Стандартными средствами helm инициализируйте структуру директории с содержимым будущего helm chart

```
helm create kubernetes-templating/socks-shop
```

Подробнее про архитектуру приложения

Создаем свой helm chart

Изучите созданный в качестве примера файл `values.yaml` и шаблоны в директории `templates`, примерно так выглядит стандартный helm chart.

Мы будем создавать chart для приложения с нуля, поэтому удалите `values.yaml` и содержимое `templates`.

После этого перенесите файл `all.yaml` из репозитория в директорию `templates`.

Создаем свой helm chart

В целом, helm chart уже готов, вы можете попробовать установить его:

```
helm upgrade --install socks-shop kubernetes-templating/socks-shop
```

После этого можно зайти в UI используя сервис типа NodePort (создается из манифестов) и проверить, что приложение заработало.

Создаем свой helm chart

Сейчас наш helm chart **socks-shop** совсем не похож на настоящий. При этом, все микросервисы устанавливаются из одного файла `all.yaml`

Давайте исправим это и первым делом займемся микросервисом **frontend**. Скорее всего он разрабатывается отдельной командой, а исходный код хранится в отдельном репозитории.

Поэтому, было бы логично вынести все что связано с frontend в отдельный helm chart.

Создадим заготовку:

```
helm create kubernetes-templating/frontend
```

Создаем свой helm chart

Аналогично чарту **socks-shop** удалите файл `values.yaml` и файлы в директории `templates`, создаваемые по умолчанию.

Выделим из файла `all.yaml` манифесты для установки микросервиса `frontend`.

В директории `templates` чарта `frontend` создайте файлы:

- **deployment.yaml** - должен содержать соответствующую часть из файла `all.yaml`
- **service.yaml** - должен содержать соответствующую часть из файла `all.yaml`
- **ingress.yaml** - должен разворачивать ingress с доменным именем `shop.<IP-адрес>.nip.io`

> Манифест для ingress необходимо написать самостоятельно

Создаем свой helm chart

После того, как вынесете описание `deployment` и `service` для **frontend** из файла `all.yaml` переустановите chart `socks-shop` и проверьте, что доступ к UI пропал и таких ресурсов больше нет.

Установите chart **frontend** в namespace **socks-shop** и проверьте что доступ к UI вновь появился:

```
helm upgrade --install frontend kubernetes-templating/frontend --namespace socks-shop
```

Создаем свой helm chart

Пришло время минимально шаблонизировать наш chart **frontend**

Для начала продумаем структуру файла `values.yaml`

- Docker образ из которого выкатывается frontend может пересобираться, поэтому логично вынести его тег в переменную **frontend.image.tag**

В `values.yaml` это будет выглядеть следующим образом:

```
image:
  tag: 0.3.12
```

! Это значение по умолчанию и может (и должно быть) быть переопределено в CI/CD pipeline

Создаем свой helm chart

Теперь в манифесте `deployment.yaml` надо указать, что мы хотим использовать это переменную.

Было:

```
image: weaveworksdemos/front-end:0.3.12
```

Стало:

```
image: weaveworksdemos/front-end:{{ .Values.image.tag }}
```

> Попробуйте обновить chart и убедиться, что ничего не изменилось

Создаем свой helm chart

Аналогичным образом шаблонизируйте следующие параметры **frontend** chart

- Количество реплик в deployment
- **Port**, **targetPort** и **NodePort** в service
- Опционально - тип сервиса. Ключ **NodePort** должен появиться в манифесте только если тип сервиса - **NodePort**
- Другие параметры, которые на ваш взгляд стои шаблонизировать

> ! Не забывайте указывать в файле values.yaml значения по умолчанию

Создаем свой helm chart

Как должен выглядеть минимальный итоговый файл `values.yaml`:

```
image:
  tag: 0.3.12

replicas: 1

service:
  type: NodePort
  port: 80
  targetPort: 8079
  NodePort: 30001
```

Создаем свой helm chart

Теперь наш **frontend** стал немного похож на настоящий helm chart. Не стоит забывать, что он все еще является частью одного большого микросервисного приложения **socks-shop**.

Поэтому было бы неплохо включить его в зависимости этого приложения.

Для начала, удалите release **frontend** из кластера:

```
helm delete --purge frontend
```

Создаем свой helm chart

Файл `requirements.yaml` содержит список зависимостей helm chart (другие chart).

Нам нужно создать этот файл в директории `kubernetes-templating/socks-shop` и добавить туда chart **frontend** как зависимость

```
dependencies:
  - name: frontend
    version: 0.1.0
    repository: "file://../frontend"
```

Обновим зависимости:

```
helm dep update kubernetes-templating/socks-shop
```

Создаем свой helm chart

В директории `kubernetes-templating/socks-shop/charts` появился архив **frontend-0.1.0.tgz** содержащий chart **frontend** определенной версии и добавленный в chart **socks-shop** как зависимость.

Обновите release **socks-shop** и убедитесь, что ресурсы frontend вновь созданы.

Создаем свой helm chart

Осталось понять, как из CI-системы мы можем менять параметры helm chart, описанные в `values.yaml`.

Для этого существует специальный ключ `--set`

Изменим NodePort для **frontend** в release, не меняя его в самом chart:

```
helm upgrade --install socks-shop kubernetes-templating/socks-shop --namespace socks-shop --set frontend.service.NodePort=31234
```

Так как как мы меняем значение переменной для зависимости - перед названием переменной указываем имя (название chart) этой зависимости.

Если бы мы устанавливали chart frontend напрямую, то команда выглядела бы как `--set service.NodePort=31234`

Создаем свой helm chart | Задание со ⭐

Выберите один или несколько сервисов, которые можно установить как зависимости, используя community chart's.

Это могут быть **MongoDB**, **MySQL**, **RabbitMQ**.

Реализуйте их установку через `requirements.yaml` и обеспечьте сохранение работоспособности приложения.

Работа с helm-secrets | Необязательное задание

Разберемся как работает плагин **helm-secrets**. Для этого добавим в Helm chart секрет и научимся хранить его в зашифрованном виде.

Начнем с того, что установим плагин и необходимые для него зависимости (здесь и далее инструкции приведены для MacOS):

```
brew install sops
brew install gnupg2
brew install gnu-getopt
helm plugin install https://github.com/futuresimple/helm-secrets --version 2.0.2
```

> В домашней работы мы будем использовать PGP, но никто не запрещает самостоятельно попробовать повторить задание с KMS 😊

Работа с helm-secrets | Необязательное задание

Сгенерируем новый PGP ключ:

```
gpg --full-generate-key
```

Ответьте на все вопросы. После этого командой gpg -k можно проверить, что ключ появился:

```
$ gpg -k
/Users/vegas/.gnupg/pubring.kbx
------------------------------
pub   rsa2048 2019-09-15 [SC]
      B086BD636EBD989F87399DD22B929BDEC3CFE7EC
uid           [ultimate] otusdemo <otusdemo@express42.com>
sub   rsa2048 2019-09-15 [E]
```

Работа с helm-secrets | Необязательное задание

Создадим новый файл `secrets.yaml` в директории `kubernetes-templating/frontend` со следующим содержимым:

```
visibleKey: hiddenValue
```

И попробуем зашифровать его:

```
sops -e -i --pgp <$ID> secrets.yaml
```

Примечание - вместо ID подставьте длинный хеш, в выводе на предыдущей странице это `B086BD636EBD989F87399DD22B929BDEC3CFE7EC`

Работа с helm-secrets | Необязательное задание

Проверьте, что файл `secrets.yaml` изменился. Сейчас его содержание должно выглядеть примерно так:

```
visibleKey:
ENC[AES256_GCM,data:N/ZmTE2PoaFn1qI=,iv:raG0p01sjoG/bSo9LM9NbzgxKuLCI3QMMfjsT1RYvbU=,tag
:2bI3zG8++m9Fo8d85caFaw==,type:str]
sops:
    kms: []
    gcp_kms: []
    azure_kv: []
    lastmodified: '2019-09-19T12:55:33Z'
    ...
```

Заметьте, что структура файла осталась прежней. Мы видим ключ visibleKey, но его значение зашифровано

Работа с helm-secrets | Необязательное задание

В таком виде файл уже можно коммитить в Git, но для начала - научимся расшифровывать его.

Можно использовать любой из инструментов:

```
# helm secrets
helm secrets view secrets.yaml

# sops
sops -d secrets.yaml
```

Работа с helm-secrets | Необязательное задание

Теперь осталось понять, как добавить значение нашего секрета в настоящий секрет kubernetes и устанавливать его вместе с основным helm chart.

Создайте в директории `kubernetes-templating/frontend/templates` еще один файл `secret.yaml`. Несмотря на похожее название его предназначение будет отличаться.

Поместите туда следующий шаблон:

```
apiVersion: v1
kind: Secret
metadata:
  name: secret
type: Opaque
data:
  visibleKey: {{ .Values.visibleKey | b64enc | quote }}
```

Работа с helm-secrets | Необязательное задание

Теперь, если мы передадим в helm файл `secrets.yaml` как values файл - плагин **helm-secrets** поймет, что его надо расшифровать, а значение ключа `visibleKey` подставить в соответствующий шаблон секрета.

Запустим установку:

```
helm secrets upgrade --install frontend kubernetes-templating/frontend --namespace
socks-shop \
  -f kubernetes-templating/frontend/values.yaml \
  -f kubernetes-templating/frontend/secrets.yaml
```

В процессе установки **helm-secrets** расшифрует наш секретный файл в другой временный файл `secrets.yaml.dec`, а после выполнения установки - удалит этот временный файл

Работа с helm-secrets | Необязательное задание

- Проверьте, что секрет создан, и его содержимое соответствует нашим ожиданиям
- Предложите способ использования плагина **helm-secrets** в CI/CD
- Про что необходимо помнить, если используем **helm-secrets** (например, как обезопасить себя от коммита файлов с секретами, которые забыл зашифровать)?
- Если вы попробовали использовать **helm-secrets** с KMS - опишите результаты своей работы

Проверка

Поместите все получившиеся helm chart's в ваш установленный harbor в публичный проект.

Создайте файл `kubernetes-templating/repo.sh` со следующим содержанием:

```
#!/bin/bash

helm repo add templating <Ссылка на ваш репозиторий>
```

После исполнения этого файла должен появляться репозиторий, из которого можно установить следующие helm chart's:

- **templating/frontend**
- **templating/socks-shop**

Kubecfg

Kubecfg

Представим, что одна из команд разрабатывающих сразу несколько микросервисов нашего продукта решила, что helm не подходит для ее нужд и попробовала использовать решение на основе **jsonnet** - **kubecfg**.

Посмотрим на возможности этой утилиты. Работать будем с сервисами `catalogue` и `payment`.

Для начала - вынесите манифесты описывающие **service** и **deployment** для этих микросервисов из файла `all.yaml` в директорию `kubernetes-templating/kubecfg`

> Манифесты описывающие создание баз данных выносить не надо

Kubecfg

В итоге должно получиться четыре файла:

```
$ tree -L 1 kubecfg
kubecfg
├── catalogue-deployment.yaml
├── catalogue-service.yaml
├── payment-deployment.yaml
└── payment-service.yaml
```

Можно заметить, что манифесты двух микросервисов очень похожи друг на друга и может иметь смысл генерировать их из какого-то шаблона. Попробуем сделать это.

Обновите release `socks-shop`, проверьте что микросервисы `catalogue` и `payment` исчезли из установки и магазин стал работать некорректно

Kubecfg

Установите kubecfg (доступна в виде сборок по MacOS и Linux и в Homebrew)

```
$ kubecfg version

kubecfg version: v0.12.5
jsonnet version: v0.12.0
client-go version: v0.0.0-master+0ed352f
```

Kubecfg

Kubecfg предполагает хранение манифестов в файлах формата `.jsonnet` и их генерацию перед установкой. Пример такого файла можно найти в официальном репозитории

Напишем по аналогии свой `.jsonnet` файл - `services.jsonnet`.

Для начала в файле мы должны указать `libsonnet` библиотеку, которую будем использовать для генерации манифестов. В домашней работе воспользуемся готовой от bitnami

Импортируем ее:

```
local kube = import "https://github.com/bitnami-labs/kube-libsonnet/raw/52ba963ca44f7a4960aeae9ee0fbee44726e481f/kube.libsonnet";
```

Kubecfg

Перейдем к основной части

Общая логика происходящего следующая:

1. Пишем общий для сервисов шаблон, включающий описание `service` и `deployment`
2. Наследуемся от него, указывая параметры для конкретных сервисов

> ! Рекомендуем не заглядывать в сниппеты в ссылках и попробовать самостоятельно разобраться с jsonnet
>
> В качестве подсказки можно использовать и готовый `services.jsonnet`, который должен выглядеть примерно следующим образом

Kubecfg

Проверим, что манифесты генерируются корректно:

```
kubecfg show services.jsonnet
```

И установим их:

```
kubecfg update services.jsonnet --namespace socks-shop
```

Магазин снова должен заработать

Задание со ⭐

Выберите еще один микросервис из состава `socks-shop` и попробуйте использовать другое решение на основе jsonnet, например Kapitan или qbec

Приложите артефакты их использования в директорию `kubernetes-templating/jsonnet` и опишите проделанную работу и порядок установки.

Kustomize

Kustomize | Самостоятельное задание

Отпилите еще один (любой) микросервис из `all.yaml` и самостоятельно займитесь его kustomизацией.

В минимальном варианте достаточно реализовать установку на два окружения - `socks-shop` (namespace `socks-shop`) и `socks-shop-prod` (namespace `socks-shop-prod`) из одних манифестов `deployment` и `service`.

Окружения должны отличаться:

- Набором labels во всех манифестах
- Префиксом названий ресурсов
- Ваш вариант...

Kustomize | Самостоятельное задание

Результаты вашей работы поместите в директорию `kubernetes-templating/kustomize`. Установка на выбранное окружение должна работать следующим образом:

```
kubectl apply -k kubernetes-templating/kustomize/overrides/<Название окружения>/
```

Проверка ДЗ

- Результаты вашей работы должны быть добавлены в ветку **kubernetes-templating** вашего GitHub репозитория `<YOUR_LOGIN>_platform`
- В **README.md** рекомендуется внести описание того, что сделано
- Создайте Pull Request к ветке **master** (описание PR рекомендуется заполнять)
- Добавьте метку `homework-10` к вашему PR
- В Assignees к PR добавьте преподавателя, который вел лекцию (GitHub логин **Avtandilko**)

Проверка ДЗ

Данное задание будет проверяться в полуавтоматическом режиме. Не пугайтесь того, что тесты в итоге завершатся неуспешно.

При этом смотрите в лог **Travis**, чтобы понять, действительно ли они дошли до "правильной ошибки", говорящей о том, что дальнейшая проверка будет производиться вручную.