

# **Домашнее задание.**

## **Шаблонизация**

# **манифестов Kubernetes**

# Intro

Домашнее задание предполагает выполнение в GKE кластере.

Для доступа к Google Cloud Platform нужно активировать триальный аккаунт в GCP. Следующие несколько слайдов покажут как это сделать.

Обзор и развернутые инструкции по созданию кластера можно найти в [официальной документации](#)

# **Регистрация в Google Cloud Platform**

Необходимо зарегистрироваться в GCP по ссылке:

- <https://cloud.google.com/free/>

Для регистрации рекомендуем использовать новую и отдельную учетную запись Google

# Регистрация в Google Cloud Platform

## Google Cloud Platform Free Tier

Learn and build on GCP for free.

[Get started for free](#)



12 months

\$300 free credit to get started with any GCP product.



Always free

Free usage limits on participating products for eligible customers, during and after the free trial. Offer is subject to change.

# Регистрация в Google Cloud Platform

Try Google Cloud Platform for free

Step 1 of 2

Country

Russia



Terms of Service

- I have read and agree to the [Google Cloud Platform Free Trial Terms of Service](#).

Required to continue

AGREE AND CONTINUE

# Регистрация в Google Cloud Platform

## Step 2 of 2

### Customer info

 Account type 

Business

 Name and address 

Business name

Otus

Name

Otus Student

Address line 1

Moscow

Address line 2

Moscow

City

Moscow

Oblast

Moskva

Postal code

140000



# Регистрация в Google Cloud Platform

Во время регистрации Google запросит ввести данные платежной карты, это единственное платежное требование.

После окончания trial-периода Google **не будет** автоматически снимать средства с карты, **можно не волноваться**.

От предложений использовать карту после окончания trial-периода рекомендуется отказываться.

# Часто встречающиеся термины

- **Google Cloud Platform (GCP)** - облачная платформа от Google
- **Cloud console** - Веб интерфейс для управления GCP
- **Cloud shell** - Веб терминал, открывается в браузере. В нем установлены и настроены утилиты `gcloud` / `gsutil`
- **Serial Console** - консоль, которая позволяет подключиться к инстансу при сетевой недоступности и других проблемах
- **gcloud/gsutil** - утилиты командной строки для управления различными сервисами GCP из Cloud SDK, аналог `aws-cli` для Amazon AWS

# Основные элементы Cloud Console

Основное меню облачных сервисов

Выбор проекта

Поле поиска

Cloud shell, справочная информация, уведомления по учетной записи и настройки

Google Cloud Platform Infra

ПАНЕЛЬ УПРАВЛЕНИЯ АКТИВНОСТЬ НАСТРОЙТЬ

**Информация о проекте**

Название проекта Infra  
Идентификатор проекта infra-188718  
Номер проекта 719574837553

Перейти к настройкам проекта

**Ресурсы**

Данные недоступны.

Перейти к обзору API

**Трассировка**

За последние 7 дней нет данных трассировки

Начать работу с Stackdriver Trace

**API API**

Запросы (запросы/сек.)

Запросы: 0,9

Перейти к обзору API

**Состояние Google Cloud Platform**

Все сервисы в порядке

Открыть панель мониторинга

**Оплата**

Ориентировочная сумма платежа 0,00 USD  
За расчетный период 1–13 дек. 2017 г.

Просмотреть сведения о расходах

**Error Reporting**

Ошибка не обнаружено. Если вы ещё этого не сделали, настройте Error Reporting.

Как настроить Error Reporting

# Панель сервисов

В боковом меню слева открывается панель со всеми возможными сервисами:

- IAM и администрирование
- Compute Engine
- Kubernetes Engine
- Сеть VPC
- Сетевые сервисы
- Storage
- Stackdriver
- ...



gcp-panel

# Панель выбранного сервиса

Google Cloud Platform Infra

Compute Engine

Экземпляры ВМ

Вызван Compute Engine

Вызвано Экземпляры ВМ

ПОКАЗАТЬ ИНФОРМАЦИОННУЮ ПАНЕЛЬ

Столбцы

Ведите фильтр

Название	Зона	Рекомендация	Внутренний IP-адрес	Внешний IP-адрес	Подключиться
<input checked="" type="checkbox"/> reddit-app	europe-west3-a		10.156.0.2	35.198.152.68	SSH

Список доступных сущностей для Compute Engine

Список инстансов

Экземпляры ВМ

Группы экземпляров

Шаблоны экземпляров

Диски

Снимки

Образы

Скидки за обязательства ...

Метаданные

Проверки состояния

Зоны

Операции

Квоты

Настройки

42

11 / 72

Избавляем бизнес от ИТ-зависимости

# Язык интерфейса Cloud Console

The screenshot shows the 'Язык и регион' (Language and Region) settings page in the Google Cloud Platform. On the left, there's a sidebar with links: Пользовательские... (User), Оповещения (Notifications), Язык и регион (Language and Region, which is selected and highlighted in blue), and Персонализация (Personalization). The main content area has a title 'Язык и регион'. It contains four dropdown menus: 'Язык' (Language) set to 'русский' (Russian), 'Формат даты' (Date Format) set to 'По умолчанию для выбранного языка: 14 марта 2015 г.' (Default for selected language: March 14, 2015), 'Формат времени' (Time Format) set to 'По умолчанию для выбранного языка: 15:45' (Default for selected language: 15:45), and 'Формат чисел' (Number Format) set to 'По умолчанию для выбранного языка: 1 234,56' (Default for selected language: 1 234,56). A 'Сохранить' (Save) button is at the bottom. To the right, a vertical menu is open, showing options like Настройки (Settings), Быстрые клавиши (Keyboard shortcuts), Файлы для скачивания (Download files), Партнеры Google Cloud, Условия использования (Terms of service), Конфиденциальность (Privacy), and Открыть интерактивное руководство (Open interactive guide).

- По умолчанию, язык интерфейса соответствует языку системы
- При необходимости, в настройках можно переключить язык

# Инструменты управления GCP

- Dashboard (GUI)
- SDK/CLI (`gcloud` / `gsutil`)
- REST API и библиотеки + сторонние инструменты и сервисы для их использования

# 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.

Для установки будем использовать **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 **Helm 3** на локальную машину.

Инструкции по установке можно найти по [ссылке](#)

Скачайте [последний доступный](#) binary файл для вашей OS и поместите его в **\$PATH**

Критерий успешности установки - после выполнения команды:

```
helm version
```

Ожидается следующий вывод:

```
version.BuildInfo{Version: "v3.0.2",
GitCommit: "19e47ee3283ae98139d98460de796c1be1e3975f", GitTreeState: "clean",
GoVersion: "go1.13.5"}
```

# Памятка по использованию Helm

## Создание release:

```
$ helm install <chart_name> --name=<release_name> --namespace=<namespace>
$ kubectl get secrets -n <namespace> | grep <release_name>
sh.helm.release.v1.<release_name>.v1      helm.sh/release.v1    1    115m
```

## Обновление release:

```
$ helm upgrade <release_name> <chart_name> --namespace=<namespace>
$ kubectl get secrets -n <namespace> | grep <release_name>
sh.helm.release.v1.<release_name>.v1      helm.sh/release.v1    1    115m
sh.helm.release.v1.<release_name>.v2      helm.sh/release.v1    1    56m
```

## Создание или обновление release:

```
$ helm upgrade --install <release_name> <chart_name> --namespace=<namespace>
$ kubectl get secrets -n <namespace> | grep <release_name>
sh.helm.release.v1.<release_name>.v1      helm.sh/release.v1    1    115m
sh.helm.release.v1.<release_name>.v2      helm.sh/release.v1    1    56m
sh.helm.release.v1.<release_name>.v3      helm.sh/release.v1    1    5s
```

# Add helm repo

Добавьте репозиторий stable

По умолчанию в **Helm 3** не установлен репозиторий stable

```
helm repo add stable https://kubernetes-charts.storage.googleapis.com
```

```
helm repo list
NAME      URL
stable    https://kubernetes-charts.storage.googleapis.com
```

# nginx-ingress

Создадим **namespace** и **release** nginx-ingress

```
kubectl create ns nginx-ingress

helm upgrade --install nginx-ingress stable/nginx-ingress --wait \
--namespace=nginx-ingress \
--version=1.41.3
```

Разберем используемые ключи:

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

# 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 --validate=false -f https://github.com/jetstack/cert-manager/releases/download/v0.16.1/cert-manager.crds.yaml
```

# cert-manager

Установим cert-manager:

```
helm upgrade --install cert-manager jetstack/cert-manager --wait \
--namespace=cert-manager \
--version=0.16.1
```

# **cert-manager**

## **Самостоятельное задание**

- Изучите [документацию](#) cert-manager, и определите, что еще требуется установить для корректной работы
- Поместите манифестиы дополнительно созданных ресурсов в директорию `kubernetes-template/cert-manager/`
- Проверить корректную работу cert-manager можно будет на последующих helm chart

# 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"  
    cert-manager.io/cluster-issuer: "letsencrypt-production"  
    cert-manager.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:

```
kubectl create ns chartmuseum

helm upgrade --install chartmuseum stable/chartmuseum --wait \
--namespace=chartmuseum \
--version=2.13.2 \
-f kubernetes-templating/chartmuseum/values.yaml
```

Проверим, что release chartmuseum установился:

```
helm ls -n chartmuseum
```

## chartmuseum

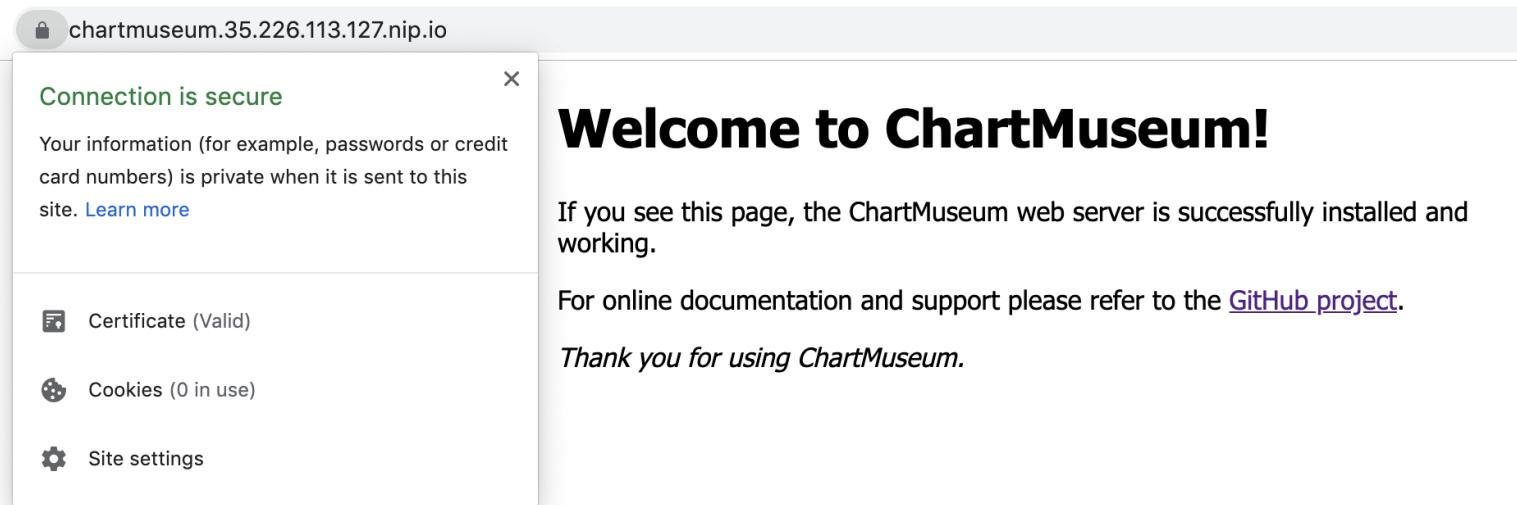
**helm 2** хранил информацию о релизе в configMap'ах (`kubectl get configmaps -n kube-system`).

А **Helm 3** хранит информацию в secrets (`kubectl get secrets -n 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

## 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-template/harbor/**

# 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
```

# harbor

## Критерий успешности установки

- Harbor запущен и работает
- Предъявленные требования выполняются

The screenshot shows the Harbor web interface at the URL `harbor.35.226.113.127.nip.io/harbor/projects`. A secure connection notice is displayed in the top left. The sidebar on the left includes links for Projects, Logs, Administrators, Users, Registries, Replications, Configuration, Tasks (which is selected), Vulnerability, and Garbage Collection. The main content area shows a table for managing projects:

<input type="checkbox"/>	Project Name	Access Level	Role
<input type="checkbox"/>	library	Public	Project Admin

## 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 манифесты для 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 [hipster-shop](#), представляющее собой типичный набор микросервисов.

Стандартными средствами helm инициализируйте структуру директории с содержимым будущего helm chart

```
helm create kubernetes-template/hipster-shop
```

# 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-hipster-shop.yaml` в директорию `templates`.

# Создаем свой helm chart

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

```
kubectl create ns hipster-shop  
  
helm upgrade --install hipster-shop kubernetes-template/hipster-shop --namespace  
hipster-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 **hipster-shop** совсем не похож на настоящий. При этом, все микросервисы устанавливаются из одного файла `all-hipster-shop.yaml`

Давайте исправим это и первым делом зайдемся 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 **hipster-shop** удалите файл `values.yaml` и файлы в директории `templates`, создаваемые по умолчанию.

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

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

- **deployment.yaml** - должен содержать соответствующую часть из файла `all-hipster-shop.yaml`
- **service.yaml** - должен содержать соответствующую часть из файла `all-hipster-shop.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-hipster-shop.yaml` переустановите chart `hipster-shop` и проверьте, что доступ к UI пропал и таких ресурсов больше нет.

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

```
helm upgrade --install frontend kubernetes-templateing/frontend --namespace hipster-shop
```

# Создаем свой helm chart

Пришло время минимально шаблонизировать наш chart **frontend**

Для начала продумаем структуру файла **values.yaml**

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

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

```
image:  
  tag: v0.1.3
```

! 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: gcr.io/google-samples/microservices-demo/frontend:v0.1.3
```

Стало:

```
image: gcr.io/google-samples/microservices-demo/frontend:{{ .Values.image.tag }}
```

Попробуйте обновить chart и убедитесь, 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: v0.1.3  
  
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 **hipster-shop**.

Поэтому было бы неплохо включить его в зависимости этого приложения.

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

```
helm delete frontend -n hipster-shop
```

# Создаем свой helm chart

В **Helm 2** файл `requirements.yaml` содержал список зависимостей helm chart (другие chart). В **Helm 3** список зависимостей рекомендуют объявлять в файле `Chart.yaml`.

При указании зависимостей в старом формате, все будет работать, единственное выдаст предупреждение. [Подробнее](#)

Добавьте chart **frontend** как зависимость

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

Обновим зависимости:

```
helm dep update kubernetes-template/hipster-shop
```

## Создаем свой helm chart

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

Обновите release `hipster-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 hipster-shop kubernetes templating/hipster-shop --namespace  
hipster-shop --set frontend.service.NodePort=31234
```

Так 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 сервисы, которые можно установить как зависимости, используя community chart's.

Например, это может быть **Redis**.

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 работе 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-template/frontend --namespace  
hipster-shop \  
-f kubernetes-template/frontend/values.yaml \  
-f kubernetes-template/frontend/secrets.yaml
```

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

# 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/hipster-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 **paymentservice** и **shippingservice**.

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

# Kubecfg

В итоге должно получиться четыре файла:

```
$ tree -L 1 kubecfg
kubecfg
├── paymentservice-deployment.yaml
├── paymentservice-service.yaml
├── shippingservice-deployment.yaml
└── shippingservice-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 `hipster-shop`, проверьте что микросервисы `paymentservice` и `shippingservice` исчезли из установки и магазин стал работать некорректно (*при нажатии на кнопку Add to Cart*)

# Kubecfg

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

```
$ kubecfg version

kubecfg version: v0.14.0
jsonnet version: v0.14.0
client-go version: v0.0.0-master+2d32dcd
```

# 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/52ba963ca44f7a4960aeae9ee0fbe44726e481f/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 hipster-shop
```

Через какое-то время магазин снова должен заработать и товары можно добавить в корзину

## Задание со

Выберите еще один микросервис из состава `hipster-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

Отпишите еще один (любой) микросервис из `all-hipster-shop.yaml` и самостоятельно займитесь его kustomизацией.

В минимальном варианте достаточно реализовать установку на два окружения - `hipster-shop` (namespace `hipster-shop`) и `hipster-shop-prod` (namespace `hipster-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 `kubernetes-templating` к вашему PR

## 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.