**Learning Kubernetes with CRD**

**CRD controller**

**Ресурсы в Kubernetes**

Этот раздел поможет понять как работают ресурсы внутри кластера и как на них влиять снаружи.

Работа кубернетов полностью построена на ресурсах. Ресурсы – часть ядра кубернетов, через которое пользователь взаимодействует с кластером, а так же происходит работа внутри кластера независимо от пользователя. Можно сказать, что ресурс – это минимальный компонент, некая элемантарная частица, а экосистема кубернетов построена на взаимодействии этих частиц. По аналогии с UNIX-подобными операционными системами, где всё есть файл, в кластере кубернетов – все есть ресурс.

Пользователь взаимодействует с кластером путем создания, обновления и удаления ресурсов. Но этот механизм прозрачен для пользователя, так как каждый ресурс описывает какую-либо конкретную сущность (ею может быть сервис, деплоймент и тд). Иными словами, пользователь создает сервис, не подозревая о создании ресурса.

Ресурс описывается с помощью YAML-файла, в котором указаны его параметры: тип (решает, как кластер будет обрабатывать ресурс), имя, пространство имен, спецификация (которая зависит от типа создаваемого ресурса), зависимые ресурсы и тд. В зависимости от типа, ресурс будет провалидирован кубернетами при создании и если его поля заполнены неправильно, то его создание отклоняется.

Пример:

Создавая деплоймент, мы на кластере создаем ресурс типа Deployment, который создает ресурс типа ReplicaSet, который в свою очередь создает ресурс типа Pod. Таким образом пользователь думал лишь об одной абстракции – Deployment, цель которой запусть приложение (что и прозойдет в итоге), однако кубернеты создали еще несколько ресурсов для своей внутренней работы.

**Можно ли создать свой тип ресурсов?**

Если есть желание – почему нет? Конечно можно! Для этого нужно создать YAML-файл с определением типа, который тоже будет ресурсом кубернетов, тип которого CustomResourceDefinition (потому что в кубернетах всё есть ресурс!), в котором можно указать имя типа и область видимости.

Сам по себе пользовательский тип и его объекты весьма безобидны и представляют собой лишь структуру данных. Но что, если мы хотим поведение подобное ReplicaSet, который следит за жизнью подконтрольных объектов и всегда держит их количество в определенном диапазоне?

Для этого нужно будет определить Custom Resource Definition Controller.

Что он умеет? Следить за созданием, обновлением и удалением подконтрольных ему ресурсов (уже после того, как это произойдет на кластере; если нужно следить за ресурсом на этапе его создания, с возможностью его модификации, то следует рассмотреть MutatingWebhook).

Где он живет? Где угодно: это может быть машина пользователя, но целесообразней запустить контроллер внутри кластера, так как он взаимодействует с кластером напрямую и внезапная недоступность контроллера (из-за фаервола или выключения пользовательской машины), при наличии на нем критически важных для работы кластера данных, может стать проблемной ситуацией. Поэтому в дальнейшем, мы рассмотрим запуск внутри кластера.

Как контроллер общается с кластером? Тем же образом, что и kubectl, а именно используя сущность kubeclient. Таким образом, мы снимаем с себя ответственность за безопасность и транспорт, делегируя эти обязанности на API, предоставленные самими кубернетами.

На выходе мы получим деплоймент, который будет разворачивать контроллер в кластере. Контроллеру нужно выдать права суперпользователя, чтобы он мог вычитать конфигурационные файлы из кластера. Он будет реагировать на события создания и удаления, а так же у него будет возможность создавать ресурсы на кластере через kubeclient.

Но мы же ничего не регистрировали на кластере, откуда CRD Controller узнает о новых ресурсах? Контроллер работает в режиме polling, поэтому узнает об изменении ресурсов пост фактум, периодически опрашивая кластер. Если же нужно работать непосредственно на создании (в режиме callback) стоит рассмотреть Mutating & Validating Webhooks.

**Создание пользовательских ресурсов и контроллера для них**

Prerequisits

1. Всё это делается, чтобы прокачать наш кластер, поэтому на этом этапе он уже должен существовать. Если его еще нет, то стоит обратиться к следующим инструкциям: [установка kubectl](https://kubernetes.io/docs/tasks/tools/install-kubectl/), [настройка кластера через kubeadm](https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/).
2. Нам потребуется установленный go и любая IDE на ваш вкус. Мне нравится VSCode. Golang лучше взять [отсюда](https://golang.org/doc/install). А VSCode можно найти [тут](https://code.visualstudio.com/).

Рассмотрим следующий случай: работая с кубернетами, я обнаружил, что среди возможностей кластера нет тех, которые удовлетворили мои потребности, а именно я хочу, чтобы с каждым новым комитом, проект собирался и выполнял автотесты. Такой себе CI на минималках, зато свой.

Для начала нужно определить тип. Для этого создадим YAML-файл следующего содержания:

apiVersion: apiextensions.k8s.io/v1beta1

kind: CustomResourceDefinition

metadata:

name: repos.myproject.com

spec:

group: myproject.com

versions:

- name: v1alpha1

served: true

storage: true

scope: Namespaced

names:

kind: Repo

plural: repos

singular: repo

subresources:

status: {}

Имя CustomResourceDefinition долдно иметь форму <plural>.<group>  
Я назвал тип нашего ресурса Repo. С помощью синонимов repo и repos мы сможем взаимодействовать с ресурсами через kubectl. Ресурсы будут создаваться в нейспейсах, так как они пользуются кубернетами, а расширяют их.

Теперь, когда у нас есть наш собственный тип ресурсов, мы можем приняться за создание самих ресурсов:

apiVersion: repos.myproject.com/v1alpha1

kind: Repo

metadata:

name: hello-world-repo

spec:

repoAddress: "path/hello-world"

dockerImage: "path/build-image" # this is used for build

buildCommands:

- "command1"

- "command2"

artifacts:

- "path/to/artifacts"

Ресурс создан. Разберем, за что он ответственен. По сути своей – это конфигурация CI. У нашего ресурса есть репозиторий, откуда будут загружены исходные файлы, а также докер образ, в котором будет происходить сборка проекта.

Но у нас нету контроллера, который бы умел правильно обрабатывать эту структуру данных.  
Поэтому нам остается только написать его.

Создадим проект для нашего контроллера по вашему усмотрению в $GOPATH. У меня это следующий путь: $GOPATH/github.com/kacejot/rep-controller**.** Для начала нужно описать структуру данных, которая будет представлять наш тип Repo в коде.

…

Затем необходимо сгенерировать для нее вспомогательный код. Можно написать его самому, но подготовленный генератор от создателей кубернетов сэкономит пару-тройку дней. Вспомогательный код может генерировать как функции, которые работают непосредственно с типами такие как DeepCopy, а есть, которые полностью реализуют логику polling, которая опрашивает кластер на предмет наличия наших ресурсов и добавляет их в кэш.

Вот [инструкция](https://itnext.io/how-to-generate-client-codes-for-kubernetes-custom-resource-definitions-crd-b4b9907769ba), которая говорит как нужно организовать структуру данных, чтобы к ней можно было применить кодогенерацию.