Начнем мы разработку наших приложений с Kubebuilder-а. И начнем с того, что установим Kubebuilder. На Mac-е он устанавливается просто путём brew install kubebuilder. Соответственно, установится у нас версия Kubebuilder 3.2.0. Это, насколько я помню, актуальная версия Kubebuilder-а. Если посмотреть на релизы – да, это 3.2.0. Это latest версия, в brew она поддерживается.

Насколько я помню, (НРЗ 00:30) тоже существуют kubebuilder. Его там не надо качать из исходников, но, если очень хочется, соответственно, можно воспользоваться, путем установки, как, соответственно, показано здесь.

Ну, и давайте с использованием нашего kubebuilder-а, мы напишем кастомные ресурсы для cronjob-а, который будет поддерживать фишки cronjob-а. То есть, запускать джобы по расписанию, делать успешный, делать очистку джобов, как умеет стандартный cronjob. Ну, и, собственно говоря, пользоваться Jobtemplate-ами и запускать эти джобы. Поехали!

Сразу после установки нам доступен kubebuilder и, соответственно, доступны его команды. Команд не так много, здесь, по сути, нас интересовать будет, будут несколько всего команд. Есть команды alpha, которые являются тестовыми командами для всяких там экспериментальных фич. А есть команды, которые загружают (НРЗ 01:40), команда create, команда edit и команда init, по сути. Нас сейчас будет интересовать команда init, которая нам создаст новый проект.

Мы находимся сейчас в папочке lecture. Зайдём в папочку kubebuilder operator и выполним эту команду. При вызове, соответственно, help-а для для нашего kubebuilder-а, мы получаем его описание. Он инициализирует нам go.mod, он инициализирует нам описание project, которое нужно kubebuilder для хранения конфигурации проекта. Он нам генерирует Makefile, сразу генерирует нужные YAML-ики для Kubernetes-а. И файл main.go, который содержит в себе менеджера,

запускающего контроллеры проекта. Соответственно, давайте создадим, давайте создадим файлик init с доменом для нашей группы, например, slurm.io или лучше…, нет, пусть будет slurm.io. И, соответственно, репозиторием slurm.io пусть будет operator. Нет, пусть будет cronjob.

Итак, у нас проект (HP3 03:03). И соответственно, теперь у нас есть всякие файлики. Давайте откроем проект и посмотрим на них. Наш kubebuilder нам, соответственно, создал go.mod, смотрим, который мы указали в качестве репозитория проекта. Создал нам всякие библиотечки, подвязал. Сделал Makefile, который, ну, не очень, не очень прост, но в то же самое время не очень сложен. То есть, Makefile умеет генерировать манифесты для наших кастомных, кастомных ролей, для кастомных ресурсов и ролей, умеет генерировать контроллеры, умеет, соответственно, запускать go fmt и go vet. Умеет запускать тесты. Соответственно, тесты для контроллеров пишутся совершенно обычным образом. Мы на них посмотрим сегодня немножко.

Умеет билдить и запускать контроллеры – соответственно, делать docker-build и docker-push. Это важно. То есть, сразу генерируются Dockerfile таким образом, как я показывал. Потому что, соответственно, контроллеры у нас будут работать где-то в Kubernetes-е. То есть, их надо собрать. И в данном случае Docker не является опциональный вещью. То есть, Docker должен быть.

Соответственно, есть команда запуска kustomize. Kustomize – это такая тулза, которая позволяет из YAML-файликов собирать другие YAML-файлики. В данном случае она нужна для автоматизации подстановки значений в эти YAML-файлы и собирания их воедино. Мы тоже посмотрим, как она работает. Команда deploy, undeploy контроллера и дополнительные служебные команды для генерации, соответственно, для скачивания, controller-gen генератора, для kustomize и для envtest. Всё.

Собственно говоря, описание проекта нас не особо интересует. Это вещь, принадлежащая kubebuilder-у. Нас будет интересовать файлик main.go. Ну, и заценим сконфигурированный для нас файлик. То есть у нас есть функция main. Main представляет из себя инициализацию менеджера. То есть, в терминологии нашего фреймворка, которой мы пользуемся, то есть, терминологии фреймворка, sigs controller-runtime, у нас существует менеджер, который запускает контроллеры. И каждый из контроллеров отвечает за свой ресурс и за свой так называемый gvk.

Gvk мы уже видели, на самом деле, группа версия, group version kind, группа версия вид. То есть, вот у нас есть api, который мы задали. Вот это его группа. Вот это его версия. Вот это его вид. И за каждую вот такую вот комбинацию у нас может отвечать свой контроллер. И менеджер отвечает за то, чтобы наши контролеры знали, за что они отвечают.

То есть, менеджер регистрирует контроллера для определенной группы и, соответственно, делает так называемый reconciliation loop. То есть, вызывает контроллер определенное, раз в определенное время. Контроллер отрабатывает и делает тот самый reconciliation, то есть, сравнивает то, что должно быть, с тем, что сейчас есть кластере, и приводит в необходимый вид. Нам нужно только реализовать ту часть, где мы сравниваем, приводим в необходимый вид, потому что всё остальное уже сделано за нас. То есть, ресурсы будут зарегистрированы. И соответственно, даже будут какие-то, даже будут предоставлены какие-то метрики.

В принципе, менеджер очень хорошо отражает то, как надо писать код контролеров. По любой ошибке мы пишем в логе. По критическим ошибкам мы сразу выходим, потому что проще закончить выполнение контейнера, чем

пытаться что-то там восстановить. Особенно, если у вас какие-то есть базовые вещи, типа там, кросс-чеки не добавляются.

И обратим внимание на вот эту вот пометку kubebuilder для скаффолдинга. Она нам пригодится позже. В папочке config у нас находятся папочки для сборки всех наших зависимостей, вернее, всех наших конфигураций для Kubernetes-а. Они находятся вот в этих самых папках kustomization. То есть, папочки, то есть, файлики вот эти нужны kustomization для того, чтобы эта утилита kustomization знала, как собирать из всех файликов какой-то единый файлик. С заменой уже, значит, с подменой значений, с заменой (НРЗ 07:52) их реальными значениями. И есть стандартные конфигурации для Deploy-я нашего менеджера. Вот тут можно увидеть, да, что у нас здесь есть config, который подгружается из папочки controller_manager_config. То бишь, отсюда.

И, соответственно, можно увидеть, что у нас генерируются файлики для deployment-а и для rbac-а. То есть, для тех самых базовых ролей, которые нужны, которые нужны менеджерам и для биндинга этих самых ролей. То есть, здесь уже на базовом уровне вам ничего делать не надо. У вас уже есть какая-то база для того, чтобы запустить менеджера. Есть вот 4 дополнительных файлика, которые ограничивают доступ к метрикам этого менеджера. Чтобы у нас мог, могли только определенные, значит, личности собирать метрики с этого менеджера. И больше…, и конфиги Prometheus-а.

И больше здесь нету ничего, потому что у нас на данный момент существует только менеджер, который непонятно, что делает. И нету никакого контроллера, который бы что-то делал, да, в нашем случае создавал бы cronjob-ы.

Давайте создадим этот контроллер. Ну, и как раз, когда мы создаем API, мы создаём gvk, группу batch, версию v1 и тип cronjob. Соответственно, он

спрашивает нас, создать ли нам ресурс. Да, создать для контроллера этого ресурса. И все. И сам добавит прекрасно все зависимости и сгенерирует нам новые файлики. Ну, и естественно, на них надо посмотреть.

Ну, во-первых, если начать с main-а, у нас тут добавилось куча всего, у нас тут добавился batchv1, который размещается в slurm.io/cronjob/api/v1. Добавили slurm.io/cronjob/controllers. У нас теперь есть контроллер и теперь есть api-шка с описанием всех типов. И, соответственно, в менеджере у нас теперь есть CronJobReconciler. То есть, здесь как раз создаётся наш контроллер, вот с помощью вот этой аннотации он у нас добавился. И теперь наш менеджер отвечает за, соответственно, этот CronJobReconciler. Назначает ему клиента для Kubernetes-овских ресурсов и схему, по которой наш Reconciler будет узнавать, что его касается, а что нет. Все это сгенерировано автоматически. Ничего я тут руками не делал, довольно-таки удобно.

И, помимо того, что у нас изменился код, код основного клиента, у нас ещё добавилась папочка api, в которой у нас теперь лежит CronJobSpec. То есть, те самые Spec в полях, когда, которые у вас появляются в описании любого ресурса. Будь то deployment, statefulset или daemonset. Само описание ресурса CronJob, содержащее поля Spec и Status. И, соответственно, список этих самых CronJob-ов для возврата из api.

Есть также помимо CronJob-а и код контроллера, тут всякие у нас есть – тут все очень тяжко задокументированно. Вот здесь у нас есть как раз правило rbac для CronJob-ов для того, чтобы нам генерировать правила rbac для людей, которые будут получать доступы. И для самих, соответственно, администраторов кластера здесь у нас автоматически уже какие-то роли сгенерировались. И соответственно, метод Reconcile, который как раз отвечает за вот эту самую логику реконсиляции. И TODO-шка, чтобы мы еще здесь, здесь заполнили нашу логику.

И соответственно, также регистрируется batch CronJob в менеджере, для того чтобы мы понимали, какие ресурсы менеджер должен передавать к нам на reconciliation. И также у нас сгенерировалась, во-первых, папочка samples с нашим batch CronJob-ом для теста. То есть, уже все у нас сразу, и api версия, и kind, и все прочее сгенерировалось. И, соответственно, groupversion с builder для добавления их…, для регистрации их в нашем менеджере. То есть, у нас уже все, в принципе, готово, для того чтобы наш контроллер мог что-то с нашими ресурсами делать. Но, опять же, если вы посмотрите на ресурсы, вы увидите, что у нас нету как таковых CRD. У нас есть apiextensions с инжектированием менеджер-сертификатов и web-hook-и.

Соответственно, есть kustomization файл, но нет самих CRD. Их нет, потому что нам нужно будет в CronJob type-ах, соответственно, задать поля и эти поля будут использованы как раз генератором, генератором kubebuilder для того, чтобы создать уже непосредственно CRD. Потому что мы помним, да, что CRD – это, по сути своей, описание каких-то у нас ресурсов. Описание – вот это вот, есть group, names, kind - вот это вот всё. Вот этого вот всего, естественно, kubebuilder заранее не знает. И какой-то автоматический интерфейс для их заполнения у вас тоже не будет. Вы напишете код Go, запустите kubebuilder, и он сгенерирует вам спецификацию.

Давайте заимплементируем сначала, наверное, поля нашего api, для наших CronJob-ов. И потом сделаем контроллер. Соответственно, наши поля – самое главное поле, которое у нас есть в нашем классе – это поле CronJobSpec. Соответственно, содержит в себе расписание для Cron-а, содержит в себе дедлайны, содержит в себе ConcurrencyPolicy для конкурентного выполнения джобы.

Это все не очень, ну, то есть, если вы работали с CronJob-ами, тут все довольно-таки понятно. Код, в принципе, откомментирован. То есть, ничего такого. Что интересно – это вот эти вот атрибуты kubebuilder, которые добавляют дополнительную валидацию для полей. Например, вот эти комментарии у вас тоже попадут в Description-ы полей, для того чтобы пользователи вашего api, ваших CRD могли бы видеть описание этих полей. Соответственно, можно полям указать, они optional или не optinal. То есть, не optional у нас по умолчанию. И я здесь комменты написал, например, на русском – это довольно (НРЗ 14:41) практика. Так лучше не делать, потому что все остальные комменты, которые у вас будут для каких-то composed типов, типа, вот я здесь применяю тип batchv1 метод для JobTemplateSpec, который является частью стандартного Kubernetes api. Комменты там будут на английском и консистентности у вас уже не будет.

Соответственно, определяем также ConcurrencyPolicy, которая является Enum. Вот здесь вот можно посмотреть, как это работает. Соответственно, определяем статус для CronJob-ы и определяем общее поле для структуры CronJob-а. Эти поля сгенерированы. То есть мы заполнили только, по сути, дополнительные поля, основная структура у нас осталась, как была. Вот тут как раз у нас есть пометка от kubebuilder-а, что это основная структура.

Теперь давайте заимплементируем сам контроллер, который является, собственно говоря, основным строительным материалом нашего оператора, и будем использовать все эти поля для того, чтобы, как раз, реконсилировать все и запускать наши джобы. Обратим внимание, что у нас по сравнению с CronJobReconciler, который изначально был, содержал в себе два файла, добавилось новое поле Clock. Это поле для того, чтобы получать текущее время и зафейкать это время для тестов этого контроллера как раз в своих тестах. И обратим внимание на то, что у нас появились новые rbac-и, поскольку наш контроллер ещё и управляет джобами, то неплохо бы ему эти джобы видеть и все

эти действия с ними делать. И получили аннотации scheduled-at, которая, которая позволяет нам обращаться к scheduled-at конкретного batch-a CronJob-a. Соответственно, здесь я пока пропущу всякие функции, которые являются вспомогательными, и перейду в основной цикл Reconcile.

Итак, начинаем разбор нашего контроллера. Начнем мы с того, что получим все объекты для данного request, вот такого типа cronjob. То есть у нас создался, удалился, что-то изменилось в нашем CronJob-е, мы получим этот объект. И когда мы не можем получить CronJob-ы - я вот тут написал, что мы проигнорируем ошибку «не найдено» (русский язык меня покинул), поскольку их нельзя разрешить немедленным перезапуском. Что я имею в виду? В поле Result, которое мы должны вернуть из нашего Reconciliation контроллера есть вот такая вот штука – requeue. И соответственно, она нам говорит, то есть, по умолчанию мы, когда получаем контроллер, мы его, то есть, получаем какой-то запрос на reconcile, мы его получаем в результате каких-то действий над Kubernetes-овским ресурсом. И можем внутри контроллера сказать, что, типа, чуваки, вообще-то нам нужно перезапуститься, после того как мы что-то там сделали, и проверить ресурсы ещё раз. Ну, и вот на этот случай есть как раз Result.

У Result-а, по сути, есть 2 опции. Опция Requeue, которая просто скажет, что нам нужно этот ключик проверить ещё раз, этот запрос. И, соответственно, RequeueAfter, который проверит запрос спустя некоторое время. Запросов, опять же, на контроллер могут быть несколько. В самом запросе есть обычно только Namespace, (НРЗ 18:12). И у вас может быть такая логика применена для нескольких, например, перезапуска.

В целом думать насчет многопоточности не надо. Kubernetes думает об этом за нас, но блокировать никакие ресурсы при этом тоже, наверное, не стоит.

Соответственно, здесь вот мы не нашли никаких CronJob-ов, ну, и ладно. Значит, возможно, что она была удалена. Это не является как таковой ошибкой.

И следующее, что мы делаем, это получаем как раз ChildJob-ы, то есть, это уже непосредственно у нас стандартная структура JobList, который мы можем получить из нашего NameSpace и с матчинговыми ключами, с матчингом ключей. То есть, Owner Job-ы — это наше имя нашего запроса.

Ищем наши активные джобы. Для поиска активных джобов применим нашу функцию isJobFinished, которая проверит наш статус. Если он JobComplete или JobFailed и, соответственно, статус является true, то, соответственно, возвращаем true. В другом случае Job-а, Job-а не закончена и определяется в другую, в другую категорию. Здесь, на самом деле, стандартный, стандартный код. Если finishedType, значит у нас finished, то это активно… Если он не finished, то это, соответственно, активная джоба. Если он failed, то failed. Если complete, то это у нас успешные джобы.

И дальше мы определяем последнее время запуска для джобы тоже с помощью функции. Тут мы просто получаем из аннотация этих джобов их время. У меня инстинкт, инстинкт автопоиском. И соответственно, после этого мы можем уже определить наиболее последние, наиболее нужное время, наиболее свежее время запуска и поместить его в статус нашей CronJob-ы для того, чтобы, ну, его можно было прочитать и, соответственно, увидеть.

Дальше мы получаем, значит, для всех активных джобов мы получаем ссылку на сам активный Job в нашей схеме и добавляем его в наши активные джобы. Потому что мы хотим их все видеть в нашем статусе. Выводим job count с более высоким приоритетом, потому что это важная информация.

И можем проапдейтить статус нашей, соответственно, нашего CronJob-а, то есть, обновить его в самом Kubernetes-е. Дальше мы проверяем наши удаленные работы, если у нас существует в спецификациях удалять работы. Ну, то есть, как вы видите, мы читаем просто спецификацию этой самой структуры CronJob-а. И из неё получаем всю информацию о том, что нам делать.

Ну то есть, по сути, вся бизнес-логика контроллера заключается в чтении структуры и применении каких-то усилий, приложении каких-то усилий соответственно её спецификации. Здесь мы сортируем failed Job-ы по их времени запуска и удаляем, соответственно, самые старые. Опять же, здесь best effort, если мы провалимся на, мы можем закончить обновление как-нибудь в другой раз.

Соответственно, если у нас successful Job-ы, мы с ними делаем то же самое. Если мы засаспендены, значит, мы просто возвращаемся из контроллера и ничего не делаем, пока не обновится спецификация для работы.

И теперь нам нужно понять, когда нам, соответственно, создавать следующие работы. Потому что мы там все удалили, все почистили, получили последнее время. Теперь бы нам надо, неплохо бы сделать основную работу, для которой мы сделали наш контроллер CronJob-а для…, которое позволит нам создать непосредственно работы.

Мы тут сразу создаем RequeueAfter. То есть, мы говорим, что нам нужно, нам нужно будет запустить эту работу ещё раз. Ничего удивительного. CronJob должен запускаться периодически, чтобы запустить работу. Поэтому, ну, в вашем контроллере это может и не быть. Соответственно здесь мы проверяем дедлайн. То есть, если мы, например, пропустили какие-то вещи, то – пропустили работы там, контроллер повис, что-то ещё случилось, Kubernetes его не дернул – то, ну, нам нужно убедиться, что мы вызовем только те работы, которые у нас,

соответственно, которые у нас, соответственно, должны быть вызваны. Понимаем, как, с какой Concurrency нужно запускать эти работы. И соответственно, пользуемся нашим методом для создания работы.

Это общий метод для создания чего угодно. У него есть мета и спецификация. То есть, вы просто берете объект Kubernetes-а. Он уже, как правило, существует в api Kubernetes-а в интерфейсах. И спокойненько, спокойненько с ним можно работать. Ну, и после того, как мы создали работу – это на самом деле ещё не все. Её нужно поместить в кластер с помощью команды r.Create. И соответственно, после этого мы запускаем работу. И потом мы schedule-рим следующий вывод для того, чтобы сделать то же самое.

Ну и напоследок сделаем 2 действия. Ну три, вернее. Соответственно, поставим реальные часы. И менеджеру, индексеру полей в менеджере мы скажем, что, соответственно, индексировать поля нужно по Owner-у и, соответственно, по имени этого самого Owner. То есть, чтобы быстро нам получать все эти списки, нам нужно, чтобы индексер знал, каким образом извлекается поле и какое поле извлекается. И ещё говорим, что, менеджеру, что наш, соответственно, контроллер own-ит джобы, владеет джобами. И, соответственно, менеджер таким образом сможет нам присылать update-ы, если, например, джоба изменилась или что-то ещё случилось.

На этом, в принципе, у нас с контроллером разбор закончен. Давайте попробуем это все собрать и запустить. Посмотрим, насколько это легко или сложно.