Controller мы написали. В prod его запустили, но естественно время после всего этого написать тесты. Но на самом деле, я, конечно, иронизирую. Тесты желательно писать заранее. На этот счёт у нас есть мощная поддержка от самой тулзы. Тулза создает, соответственно, в папочке controller-ов, как в папочке, которая содержит хоть какую-то логику, создает сразу тесты. Естественно, в папочке API тестов у нас нет, потому что это по сути своей data структуры и в них особой логики у вас быть и не должно. Соответственно, Suite_test создает наши тестовые API-шки, уже сразу регистрируют к ним handler-ы для наших тестов, kubetest…, kubebuilder создает связку из envtest-а, который запускает GO-шный…, который запускает Kubernetes-овский сервер тестовый. Такой embedded сервер, который поддержит все Kubernetes-овское API. Своего рода поддерживает, сейчас мы увидим, как оно поддерживает. И использует библиотеку ginkgo, она же ginkgo вместе с gomega, которая является просто языком описания для теста, по сути. Вот это – это всё gomega. То есть некая попытка сделать, естественным языком описать всю логику.

Соответственно, ginkgo отвечает за эти suite-ы и за связку всего этого воедино. Как мы можем увидеть, здесь у нас у testEnv-а собирается config. Проверяем, что у testEnv-а нет никаких ошибок, что config не пустой, то есть всё соответственно, нормально стартануло. Вот это тоже ginkgo-вская фраза, которая просто нам позволяет понять, что происходит. По сути, такой подвид log-ирования внутри test-ового framework-а. И соответственно, дальше у нас добавляется наш batch в схемы. И проверяется, что нет никаких ошибок, создается client, который будет пользоваться… То есть API client, которым будем пользоваться наш тестовый controller, и соответственно, мы будем пользоваться в рамках тестирования.

Для запуска тестов у нас есть команда make test, уже обо всё подумано. В чем смысл этой команды? У нас для testEnv-а нужно определить KUBEBUILDER ASSETS, то есть нужно определить, где у нас лежат всякие вещи, нужные для

запуска этого testEnv-а, например, ETCD и прочие Kubernetes-овские приблуды. Если вы запустите test просто так, через Go test, он пожалуется, что не может найти. То есть если я сейчас сделаю go test controller-а, то он скажет, что: «Я не я, рыба не моя. У тебя вообще всё депрецировано. Не могу найти, не могу стартануть controlplane, потому что ETCD здесь нет. Потому что он ищет в другой папке». Соответственно, Kubebuilder assets должны определены быть, нам не нужно об этом думать за счёт того, что у нас есть make test.

Теперь давайте напишем test и по нему пройдемся, чтобы понять, как работает этот самый ginkgo и значит, gomega. И давайте посмотрим на новый код в suite_test. Suite_test организован таким образом, что вы добавляете тесты для controller-ов, добавляете controller-ы и тестируете их в своих файлах. Поэтому сюда добавляется cronjob controller_test, который мы сейчас рассмотрим.

Здесь что? Здесь мы создаем наш manager, тот самый manager, который управляет controller-ами и привязанный к нему Reconciler, наш cronge-овый Reconciler. С client-ом manager и схемы тоже от manager-а. Но manager, соответственно, скармливается config отсюда, но это и так понятно.

Затем мы в отдельной горутине запускаем наш менеджер, чтобы не мешать cleanup-у в отдельной горутине и её отменяем в clean up-е. Что здесь ещё можно сказать? Этот код вы можете, в принципе, увидеть, что он дико похож на то, что у нас происходит в main.go. У нас есть такой менеджер, но тут, правда, больше параметров, естественно. Здесь всякие литеры, здесь метрики и так далее. И точно также запускается наш Reconciler, и в случае ошибки мы выходим. Здесь в случае ошибки мы, соответственно, кидаем негативный assert.

И ещё я упоминал, что мы используем k8sClient для тестов. А вот k8sManager используют свой client, который мы и передаем в CronJobReconciler. Можно

передать сюда k8sClient, но у manager-ов в client-е существуют всякие фичи в духе всяких indexer-ов, cashing-а и прочего, чего нет в нашем client-е. Потому что наш client – он предназначен для тестирования с точки зрения пользователя, который не ожидает никаких cash-ей. Здесь мы хотим тестировать как в реальной жизни, поэтому k8sClient -а напрямую не передаем. Передаем всё от manager-а, это нам позволит тестировать просто прямо реально напрямую.

А теперь разберемся с нашим cronjob_controller_test. Если у нас здесь было б несколько controller-ов, у нас был бы какой-нибудь cronjob_controller, у нас бы был batabase_controller и прочее. Соответственно, в этом тесте мы создадим cronejob-у и проверим, что controller при создании дополнительных job-ов записывает в эту cronejob-у в поле active эти активные job-ы. То есть мы проверим эту часть кода controller-а.

Кроме как интеграционных тестов я не вижу вообще смысла тестировать как-то controller. Но если у вас там какие-то есть простые служебные функции как у меня здесь есть вот эта функция, например, в которую можем job засунуть, а она time вернет. Но можно, наверное, описать Unitest-ы для неё. Unitest написать…, пишутся абсолютно как обычно, но, в основном у вас controller-ы будет основной Reconcile-яционный loop вот этот вот. И будет какая-то в нём логика зашитая глубоко на API Kubernetes-а. Кроме как каким-то embedded Kubernetes-ом её тестировать, наверное, нет смысла.

И давайте теперь разбираться по частям, что здесь у нас есть. Начнём с описания непосредственно конструкции нашего ginkgo. Конструкция Describe – это у нас такой блог, в который мы можем включить описания тестов. Несколько тестов – BeforeEach, after age. AfterEach – это всё методы, которые будут запускаться перед каждым тестом после каждого теста и непосредственно перед самим тестом. И It – это и будет описанием нашего теста. Внутри describe-а можно поместить другие

describe-ы, можно поместить Context-ы и When. Они абсолютно эквивалентные, но они обозначают разные вещи. То есть обычно пишется 1 Describe, ему передаются контексты и When. Мы это предпочитаем, я как…, в принципе, в прошлом Java-программист, предпочитаю об этом думать, как suit и непосредственно тесты в том же самом j-unit-е, например.

Здесь у нас есть вот такой же 1 Describe, который мы говорим, описываем job…, Kronejob controller, в нём мы задаем константы. Какие мы Kronejob-ы с какими мы задаем, в каком namespace, как у вас job имя и всякие служебные вещи типа timeout-ов, проверок, длительности проверок и интервалов проверок. Мы сейчас посмотрим для чего они нужны. И единственный тест, который у нас здесь есть – это наш контекст, когда update-ится статут Cronejob-ы и соответственно… Вернее, 1 блог и внутри него 1 тест, то есть мы говорим, что, когда update-иться Cronejob status – мы должны увеличить Cronejob status active. Здесь вы можете, например, 2-й тест добавить. Когда у нас fail-ится job-а, что будет происходить или когда у нас запустилось несколько job подряд с каким-то интервалом, какое должно быть имя…, какое должно быть значение в ласке job интервал? В общем, здесь понятно, это eat – это у нас непосредственно тест.

Но и внутри теста всё начинает с By, создание context-а и обращаю внимание на то, что нам для передачи, вот у меня тут в комментарии написано, что надо для передачи. Для создания cronejob-а в этом самом нашем embed-овском Kubernetes-овском API, нужно непосредственно создать структуру cronejob-ы со всеми её вложенными полями, то есть вот ту, которую мы как раз описали в наших spec-ах. То есть никаких CRD, никаких Yaml-овских конфигурационных файлов у вас не будет. Только Golang, только хардкор. Задаем всё руками, поэтому для простоты здесь мы будем использовать только обязательные поля.

Мы создаем нашу работу, передаем наш Kubernetes-овский кластер и с помощью вот этого самого k8sClient-а из нашего Test suite-а. Не manager-ского client-а, а именно вот этого без всяких cash-ей. Мы создаем cronejob-у, это обычная Kubernetes-овская API, которую вы можете засунуть в ваши параметры и ждём, что Kubernetes нам вернет, что всё хорошо.

После создания cronejob-ы мы будем…, мы её, соответственно, заберем из Kubernetes-овского API. С помощью того же самого client-а и проверим, что соответственно, мы её, во-первых, можем забрать, а во-вторых, её Schedule, например, вызывать каждую первую минуту каждого часа.

Соответственно, что здесь интересно? Здесь мы использовали функцию Expect, но здесь всё понятно – это прямая логика. Expect, какой-то линейный возврат, Should, и здесь должен быть результат.

Здесь мы используем функцию Eventually. Функция Eventually предполагает, что вот это у нас может не вернуться с первого раза, потому что job-е нужно какое-то время, чтобы создаться. Поэтому нужно это вызвать несколько раз. Вызывается, соответственно, несколько раз с каким-то интервалом. Он у нас интервал указан, интервал у нас 250 миллисекунд. И пока не превысит она будет вызываться, пока shoot не отработает верное значение. В shoot не появится, вернее, верное значение. Функция не вернет верное значение. Либо timeout не истечет, и соответственно тогда у нас тест провалится.

И после того, как мы создали уже непосредственно работу в кластере, мы смотрим, что за какое-то определенное время: за 10 секунд в кластере новых работ незаschedule-лось. То есть самостоятельно у нас job-ы не создаются. И дальше мы можем создать нашу job-у.

Job-у мы создаем точно таким же образом, как мы до этого создавали наш cronejob. То есть мы прямо руками берем, создаем job-у. Ей заполняем поля, заполняем её статус, что Active: 2, значит, что pod-ы у нас все запущены. Потом мы получаем на неё ссылку и обращаю внимание, что нужна эта GroupVersion kind из-за нашего cronejob-а, потому что иначе мы не сможем установить ему owner-а.

Потом, значит, получаем ссылочку для controller-а и устанавливаем reference на owner-а. И если всё прикольно, всё у нас хорошо, то в нашем, значит…, в нашем cronejob-е должна появиться наша job-а и таким образом мы проверили, что наш controller всё правильно номерует, правильно создает и правильно update-ит наши cronejob-ы.

И в принципе, это такой путь тестирования controller-а, то есть я бы не стал писать тест, который тестирует сразу всё. То есть создает здесь job-ы с каким-то интервалом. Потом мы, значит, ставим часики, убеждаемся, что job-а создана. Потом убеждаемся, что job-ы у нас выполнены. Это, в принципе, слишком много логики для одного теста, поэтому тестируем изолированные кусочки. Можно, например, создать несколько job-ов, одну проваленную, одну выполненную. Привязать их к controller-у и посмотреть, правильно ли у нас определятся выполненные и проваленные job-ы. И так далее.

То есть каждый тест тестировать будет у вас какую-то часть вашего recancelation группа. У вас для этого будут все инструменты. И после того, как мы всё это написали, из главной папочки make test и наслаждаемся жизнью, вот она покажет наш coverage и будет работать как, в принципе, обычный стандартный Golang тест. И позволит нам протестировать интеграционно всю логику.

И в процессе записи видео я так довольно вольготно оперирую этими понятиями, всякие get-ы, листы и прочее. На самом деле ничего такого, то есть мы помним,

что у Kubernetes-а, у него все API – это Kubecut-ы get что-нибудь. Нет такого типа. Kubeclt, там get pods. Kubectl get namespace. Kubectl можно, например, describe namespace default. И собственно говоря, все эти вещи – это глаголы. То есть в данном случае Get и Describe разделены, но это, по сути, говоря, list и get. И здесь тоже самое: у нас разделены интерфейсы нашего API на тип reader и тип writer. Соответственно, тип reader в себе содержит глаголы get и list. Тип writer содержит в себе содержит глаголы создать, удалить, проапдейтить, пропатчить и так далее. И они все-все принимают объекты одного из ресурсов Kubernetes-а, которые, собственно говоря, тоже определены в API под различными директориями, под различными package-ми этого модуля.

И описание всех этих модулей – они уже непосредственно размещены внутри API, то есть здесь перечислены различные типы подтипы API. То есть это всё ресурсы. Всё те самые наши, значит, группы и версии. И если мы теперь зайдем в core v1, тут мы увидим дохринища чего, всякие к ЭВС-ам привязанные вещи. И увидим, например, типы нам уже знакомые. То есть типы, допустим, namespace. Тип node-а, тип node, адрес – это всё Down stream поля, то есть те поля, которые встроены внутри node, и прочее. То есть при хорошем, довольно частом употреблении Kubernetes-а здесь не должно возникнуть вообще никаких вопросов. Вы смотрите, к чему принадлежит ресурс и всё, опа, у вас уже есть зацепочка, куда лазить в этот K8test его API.

Ещё у всех модифицирующих и не модифицирующих, соответственно, всех кроме Get-а объектов есть опции, которые вам позволяют добавить, значит, Create к какому-то, например, другому create-у. Если у вас есть уже какие-то опции, вы можете их совместить.

Но и больше здесь, по сути, говоря, ничего особо нет. То есть всякие обработки ситуации. Довольно стандартная обертка на (НРЗ 16:44) API, которая ещё и

довольно-таки хорошо перекликается с тем, что есть в самом Kubernetes-е, поэтому здесь проблем возникнуть не должно.