В главе про Istio я показал красивую картинку, где у меня там, значит, pod-ы были, а к ним были прикручены proxy. Вот эти proxy – это пример так называемого паттерна Sidecar.

Смысл в том, что Pod – это несколько контейнеров, как я уже говорил. И у вас помимо одного контейнера, в котором само приложение, у вас могут быть еще дополнительные контейнеры, которые обслуживают этот контейнер. Почему? Потому что Kubernetes так же, как Linux так же, как Go, в принципе, исповедует вот тот принцип, что каждое приложение, каждый контейнер должен делать только одну работу. И делать её хорошо. И не распылятся на дополнительные функции. Например, если у вас есть приложение, которое, там, отдаёт сетевые запросы, и например, вам нужно сделать миграции, вы можете сделать Sidecar к этому приложению, почему бы и нет. Он отдельно запустит миграцию этого приложения. И соответственно, ваше приложение будет сконцентрировано только на том, чтобы себя запустить. У него не будет вот этих предварительных…, предварительных всяких scheduler-ов, предварительных каких-то команд для того, чтобы запустить миграцию. Оно просто запуститься, а миграции доедут через Sidecar.

Ну или, например, у нас было приложение, у которого статика обновлялась тоже через Sidecar. Sidecar проверял там определенный сайт, парсил и клал приложение. Приложение – у него основная задача была эту статику отдать. Соответственно, у Sidecar-а есть несколько вариаций. И часто используемая вариация – это паттерн Ambassador, это паттерн, например, паттерн клиентского proxy, то есть вы делаете запрос из своего приложения к чему-то, да, и вы делаете запрос не к этому конкретно, а к Ambassador-у этого сервиса. Например, если у вас есть какой-то (НРЗБ 01:56), какой-то кэширующий слой. И у вас есть несколько instance-ов этого слоя, то приложение не должно думать: так, а к какому слою мне сейчас обратится, какому там, какой мне выбрать? Как мне там определить какой

из них загружен, не загружен? Вы делаете Ambassador, приложение обращается к Ambassador-у, Ambassador уже сам определяет куда ему нужно сходить и какие данные получить.

Соответственно, третье, третий вид Sidecar-а, вернее второй вид Sidecar-а из часто используемых – это Adapter. Adapter конвертирует, как правило, ответ от вашего сервиса в ответ, ожидаемый другим сервисом. Самый простой пример: у вас был Prometheus, например, как приложение мониторинг, ожидал данные в определенном формате. Мы с ним уже познакомились в практических заданиях. Вы решили, что, блин, не подходит мне Prometheus, хочу Honeycomb, там, хочу как комплексное решение, Honeycomb не знает про метрики Prometheus-а. И что вы? Вы идёте ко всем разработчикам и говорите: «Блин, ребята, а сделайте мне там, вы там интегрировали библиотечку Prometheus-а, давайте вы теперь переделаете её, там, на библиотечку Honeycomb-а». Они скажут: «Блин, чувак, ну, тут три Story Point-а, тут два Story Point-а Task-е, ну, в общем – не круто». Что вы можете сделать? Вы можете повесить каждому контейнеру Adapter, и Adapter посмотрит на какие…, и Adapter будет точкой входа для вашего мониторинга. И он сам будет конвертировать. Он знает, что там, там Prometheus, тут Honeycomb, он все эти метрики сконвертирует и отдаст непосредственно в вашу тулзу мониторинга. Ну, как один из примеров.

Резонный вопрос: зачем нужны Sidecar-ы, если вы всё можете сделать в коде, вы и в библиотеке можете эти Prometheus-овские сделать. И подключить клиента того же Redis-а, которому можно просто указать список контейнеров Redis-а, и он будет с ними работать, ну и так далее. То есть это всё решается кодом. Здесь дело как раз в том, что контейнер должен делать только свою задачу. Это, опять же, удобно, когда разработчики не знают: разработчик может локально там со своим Redis-ом тестировать, он не знает какой у вас там (НРЗ 04:10) решение, сколько instance-ов, Redis-ов вы туда запихнули, какие у них обертки, и что вообще это…,

его не должно касаться. Это ваша задача, поставить нужный Ambassador и так далее. Поэтому здесь удобное разделение ответственности. Разработчик делает свою задачу, вы делаете свою задачу и ваши контейнеры тоже работают в разных парадигмах. И не делают общие задачи.

Ну, здесь то же самое. Задачи, не относящиеся к основной функции контейнера, в нём выполнятся не должны. Мы это уже, в принципе, обсосали. И инфраструктурные сетевые задачи не должны, опять же, быть точкой головной боли для разработчиков. Разработчик не должен знать, что у вас там istio, у вас там, я не знаю, traffic, у вас там что-то, что, о чем ему нужно иметь какое-то представление. Ему не надо. В том то и дело. Для этого у вас есть Kubernetes, с которым вы разделяете все эти (HP3 05:07).

Sidecar-ы – всё здорово, причем тут Go? Для дополнительных функций сервиса мы, конечно, будем использовать язык, на котором написан сервис. Если у нас какое-то веб-приложение с тремя точками входа, у него просто будет три Sidecar-а, вернее в приложении два Sidecar-а, которые будут что-то делать. Например, там, один скейджилит какие-то задания по расписанию, другой – там, не знаю, гоняет Daemon-ов для того, чтобы весело жилось с новым приложением, ну и так далее. Тут нет вопросов. Тут не нужен, ну, тут не нужно специально форсить разработчиков, делать на golang-е.

Но инфраструктурные контейнеры, какие-то ваши дополнительные контейнеры, ваши адаптеры, есть смысл делать так, чтобы они, опять же, занимали минимум «веса», имели минимум обвязок, минимум зависимостей для загрузки и работали, работали довольно быстро.

Ну и давайте посмотрим на парочку примеров Sidecar-ов. Первый пример Sidecar-а – это в нашем, соответственно, кластере из микросервисов (HP3 06:13) с

помощью istio можно увидеть, что…, ну, вернее, не с помощью istio, а с (НРЗ 06:16) istio можно увидеть, соответственно, Pod-ы, у которых два Pod-а в состоянии READY. Если мы сделаем describe любого Pod-а – мы, значит, увидим, что здесь раз контейнер наш с деталями, и два контейнер – контейнер istio- proxy. Вот этот контейнер istio- proxy – выступает как Sidecar, в данном случае, он выступает как proxy, ограничивает сетевые взаимодействия, вернее не ограничивает, а дополняет сетевые взаимодействия нашего контейнера, позволяя ему, собственно говоря, работать с помощью istio. И позволяя, соответственно, всякие различные istio-штуки. Нас интересует только что, значит, у нас есть один контейнер, и второй контейнер в Pod-е, каждый из которых выполняет какие-то разные вещи. Один выполняет основную функцию, в нашем случае details-контейнер выполняет функцию отдачи деталей по произведениям, по нашим книжкам, например. Второй контейнер выполняет чисто proxy работу со всей авторизацией, с сертификатами, с всякими различными адресами, сервисами, и всяким прочим.

Помимо этого описания Sidecar-а, вот, например, если у вас нет никакого istio, да, там, если вы просто что-то деплоити, есть такой классический, я бы сказал книжный практически пример, когда у вас Nginx пишет что-то логе, Nginx не умеет писать Stdout, по умолчанию он пишет всё в файлы. И у вас есть вот второй Sidecar, который всё пишет в Stdout Pod-ы, чтобы это всё можно было съесть значит, съесть fluentd решением каким-то логинговым решением. И скормить куда-то в elasticsearch.

Здесь, в принципе, всё. Ну, Sidecar – это очень простой концепт, главное понять принцип разделения зависимостей. Здесь больше ничего особо нет.